Symbolic Languages
in Data Processing

SYMBOLIC LANGUAGES IN DATA PROCESSING

Proceedings of the Symposium
organized and edited by the
International Computation Centre
Rome, March 26-31, 1962

GORDON AND BREACH Science Publishers
New York London 1962

Science Publishers, Inc.
150 Fifth Avenue, New York 11, New York

Library of Congress Catalog Number 62-22085

For Great Britain and Europe:
Gordon and Breach, Science Publishers, Ltd.
171 Strand
London W.C. 2, England

Printed in the United States of America

PREFACE

During its provisional period, the International Computation Centre, whose mission is to promote international scientific co-operation, already organized four Symposia concerning various subjects of interest in the field of automatic computation and numerical analysis.

This present symposium is, however, the first scientific conference which the International Computation Centre has organized since it obtained its permanent status at the end of 1961.

Professor F. L. Bauer, Mainz, was the scientific coordinator.

Research in the field of symbolic languages being a subject of current scientific interest, this Symposium met with a large response from specialists throughout the world.

During the period 26-31 March, 1962, more than 270 participants assembled at the "Palazzo dei Congressi," Rome—EUR, close to the ICC headquarters. Fifty papers were presented, and six panel discussions were organized on relevant subjects. The present volume contains the full text of these papers, as well as summaries of the Panel Discussions.

With regard to these summaries, we should like to point out that it was impossible to submit to the numerous speakers the draft texts of their statements, transcribed to the best of our ability from a tape recorder. We cannot therefore guarantee that the text always reflects exactly what the speakers wanted to express.

Nevertheless, we believe that this section of the book will, taken as a whole, show, in one of the most rapidly developing fields of science, a true picture of the situation as it was in March 1962.

The various sessions were presided by Messrs. C. Böhm (Italy), S. Comét (Sweden), P. Dreyfus (France), S. Gorn (USA), R. Kogon (USA), L. Lukaszewicz (Poland), S. Moriguti (Japan), K. Samelson (Germany), H. R. Schwarz (Switzerland), A. van Wijngaarden (Netherlands), M. V. Wilkes (United Kingdom).

The names of the panelists, pre-selected for each discussion, are indicated at the beginning of the corresponding sections.

The reader will note that numerous speakers, not appearing on the panel, also took the floor during the discussions.

The opening session was honoured by the presence of Professor Giovanni Polvani, President of the *Consiglio Nazionale delle Ricerche*, Rome, and by Professor Aldo Ghizzetti, Director of the *Istituto Nazionale per le Applicazioni del Calcolo*, Rome, whose collaboration in organizing the Symposium was particularly appreciated.

Finally, we wish to thank the following Institutions and private firms who kindly offered donations towards the financing of this Symposium:

France: *Compagnie des machines BULL*

Cie générale de Télégraphe sans fil

IBM France

Société d'Electronique et d'Automatisme

Société europeénne pour le traitement de l'Information

Italy: *Istituto Nazionale di Alta Matematica*

Istituto Nazionale per le Applicazioni del Calcolo

IBM Italia

Olivetti

Remington Rand Italia

Stig Comét
Director, ICC

PREFACE

Alors que son statut était encore provisoire le Centre International de Calcul avait déjà organisé quatre Symposiums sur divers sujets intéressant le calcul automatique et l'analyse numérique.

Le Centre International de Calcul, dont la mission est de développer la coopération scientifique internationale, ayant acquis, fin 1961, son statut définitif, le présent Symposium en est la première manifestation publique officielle.

Le Proffesseur F. L. Bauer, de Mayence, en a été le coordonnateur scientifique.

La recherche dans le domaine des langages symboliques étant à la pointe de l'actualité scientifique, ce Symposium a recontré un très large écho parmi les spécialistes du monde entier.

Du 26 and 31 mars 1962, plus de 270 congressistes se réunirent au "Palazzo dei Congressi," Rome. Cinquante communications furent présentées, et six séances de discussions furent organisées sur les sujets à l'ordre du jour. Le présent volume contient le texte des communications et les comptes-rendues des discussions.

A propos de ces comptes-rendus de discussions nous devons signaler qu'il était impossible de soumettre aux nombreux orateurs le texte de leurs interventions, tel qu'il a dû être rédigé, tant bien que mal, d'après les enregistrements sur bande magnétique. Nous ne pouvons donc pas garantir que ce texte reflète toujours intégralment l'opinion que les congressistes avaient l'intention d'exprimer.

Tels quels, cependant, nous croyons que ces comptes-rendus constitueront dans leur ensemble un tableau valable des doctrines qui ont été actuelles en mars 1962, dans un des domaines les plus fluctuants de la science.

Les differentes séances ont été présidées tour à tour par MM. C. Böhm (Italie), S. Comét (Suède), P. Dreyfus (France), S. Gorn (USA), R. Kogon (USA), L. Lukaszewicz (Pologne), S. Moriguti (Japon), K. Samelson (Allemagne), H. R. Schwarz (Suisse), A. van Wijngaarden (Hollande), M. V. Wilkes (Royaume Uni).

Les noms des protagonistes ("panelists"), présélectionnés pour chaque discussion, sont indiqués en tête des chapitres correspondants. Le lecteur remarquera à ce propos que de nombreux orateurs ont pris la parole, qui ne faisaient pas partie du noyau primitif, mais improvisaient une intervention au cours de la discussion.

La séance d'ouverture fut honorée par la presence du Professeur Giovanni Polvani, Président du *Consiglio Nazionale delle Ricerche*, Rome, et du Professeur Aldo Ghizzetti, Directeur de l'*Istituto Nazionale per le applicazioni del Calcolo*, Rome, dont la collaboration à l'organisation du Symposium fut particulièrement appréciée.

Nous tenons enfin à rendre hommage aux Institutions et aux enterprises suivantes, dont l'aide généreuse a contribué au financement du Symposium:

France: *Compagnie des machines BULL*

Cie générale de Télégraphhe sans fil

IBM France

Société d'Electronique et d'Automatisme

Société européenne pour le traitement de l'Information

Italie: *Istituto Nazionale di Alta Matematica*

Istituto Nazionale per le Applicazioni del Calcolo

IBM Italia

Olivetti

Remington Rand Italia

Stig Comét
Director, ICC

CONTENTS

DESIGN OF PROBLEM ORIENTED LANGUAGES—
ALGOL AND COMPETITORS

DESIGN OF LANGUAGES–FOR COMMERCIAL PROBLEMS

DESIGN OF SPECIAL LANGUAGES

PROBLEMS OF PROGRAMMING SYSTEMS

THEORY OF LANGUAGES– SYNTACTICAL STRUCTURE AND META LANGUAGES

AN AXIOMATIC APPROACH TO PREFIX LANGUAGES*

Saul Gorn

University of Pennsylvania
Moore School of Electrical Engineering
Philadelphia, Pennsylvania, USA

INTRODUCTION

Some mechanical languages have a number of properties usually thought of as being characteristic of natural languages. Until recently the various notational systems in the various sciences and arts developed at such a leisurely pace and with seemingly so tenuous a relationship to the concepts and processes they were designed to simulate symbolically that, on the one hand, it was considered a rather dramatic, though interesting, analogy to call them languages at all, and, on the other hand, their growth and development was considered more an evidence of the ingenuity of a few scattered people than a necessary process resulting from mass-communication. The recent acceleration in the development of the sciences in general, and of the use of large scale, general purpose, digital computers in particular force us to look at the matter differently.

The most successful programming languages for machines have a seemingly simple structure, are more or less easy to learn, and do not seem to be too specific in their areas of application, all properties possessed by natural languages. The more successful they are in these respects, the more they tend to be applied to other areas, again like natural languages. The more extended this growing domain of application, the more the

*The background work on mechanical languages [see Gorn (3)] in this paper was originally supported by the U.S. Army Signal Corps from 1958 to 1960 under Contract DA-36-039-SC-74047 to the University of Pennsylvania's Institute for Cooperative Research and Moore School of Electrical Engineering. The developments in this paper were made possible by joint support from the National Science Foundation (NSF-G-14096) and the Air Force Office of Scientific Research (AF-49(638)-951).

programming language itself tends to grow, occasionally modifying basic elements in its structure to do so, and developing 'sub-languages' for use in broad groups of applications, even as the portions of natural languages in the individual arts and sciences seem to become jargons and dialects.

This moment in history is therefore an appropriate one to study such growing mechanical communication systems. Because of their accelerated growth we might be able to learn in various limited ways about the growth of those natural communication systems called languages, but, in any event, we must understand their structures and laws of growth if only to keep them properly controlled. Linguists might be able to use the insights developed from studying the growth of mechanical languages, but, right now, it is more urgent that mechanical linguists learn some insights from the natural linguists.

To begin to understand laws of growth of mechanical languages and their processors, we would like to find basic processors which make them grow under control, and to find basic processors which select desired sub-languages and construct their processors. Thus, on the one hand, we must seek properly to define appropriate processes yielding appropriate methods of extending mechanical languages into 'super-languages', and, on the other hand, we must seek to define the appropriate concept of 'sub-language', one which yields more structure than the mere concept of "any subset of the extent of a language". Beginning with the latter goal first, if we can develop an appropriate theory of the 'sub-languages' of a language, even if it is purely descriptive, extensional, formal, and non-constructive, in short, even if it is pure mathematics, we would then be in a position to study constructive procedures which extend sub-languages to yield more extensive sub-languages. These procedures would be specified and studied in a mixed descriptive and command language which would emphasize the intent of the languages as well as their extent, i.e. would constantly be concerned with their interpreting processors. In short the second goal, which is to obtain a constructive, intensional, nonformalized specification of these processes of extension in a mixed descriptive and command language, belongs to the computer and information sciences.

One class of mechanical languages of particular interest in the computer and information sciences is that of the 'complete prefix languages'. Because of standard scanning procedures, the

interpreters of such languages all proceed without the use of those 'control characters', known as 'parentheses', whose function is to delimit those substrings of 'well-formed expressions' forming the 'scopes' of the 'command characters' known as operators. The device whereby parentheses are made unnecessary was first introduced in symbolic logic by the Polish school; its utility is, however, considerably broader, as was pointed out by Curry and the school of combinatory logic (2). It has been applied to arithmetic notation, in various computer languages, and has affected the design of some recent digital computers.

This paper will characterize these 'complete prefix languages' by a number of properties as axioms. Omitting some of these axioms will yield a broader class of prefix languages, broader in several respects. In the first place, languages of the broader class will not be 'complete' in the sense that it will be possible to add new words to their extent without having to accept ambiguous possibilities in their decoding (more specifically, in their deconcatenation). In the second place, languages of the broader class will have a more flexible 'scope' structure in that some 'operational' characters may accept operands whose number varies with the context.

Before presenting such an axiom system, it is useful to summarize the development of the complete prefix languages in the manner of the computer and information sciences.

COMPLETE PREFIX LANGUAGES

The complete prefix language function $\wp$ is applicable to any 'simply-stratified alphabet'. The fundamental properties of complete prefix languages, $\wp\alpha$, are a result of application of a general theorem about 'doubly-stratified alphabets'; we requote it and related theorems from Gorn (3) and (4):

Theorem: If there is a pair of mappings of the alphabet α, finite or infinite, into the natural numbers or any representation language thereof, so that, for any character c of α two uniquely determined numbers may be generated which will be called the 'head' and 'tail' stratifications of c, $n_h c$ and $n_t c$, then the following recursive definition yields an extension of these mappings to head and tail stratifications of all finite strings of characters over α, $N_h\alpha$ and $N_t\alpha$, which are 'associatively invariant under concatenation'. Expressed in greater detail:

Let 1. $N_h c = n_h c$ for every c of $\mathfrak{a}$,

2. $N_t c = n_t c$ for every c of $\mathfrak{a}$,

3. $N_h c\,\alpha = n_h c + (N_h\alpha \dot{-} n_t c)$ for every c of $\mathfrak{a}$ and every string α over $\mathfrak{a}$,

4. $N_h c\alpha = n_h c + (n_t c \dot{-} N_h\alpha)$ for every c of $\mathfrak{a}$ and every string α over $\mathfrak{a}$,

where $x \dot{-} y = \begin{cases} x - y & \text{if } x \geq y, \\ 0 & \text{otherwise} \end{cases}$

Then 1. $N_h(\alpha\beta) = N_h\alpha + (N_h\beta \dot{-} N_t\alpha)$,

2. $N_t(\alpha\beta) = N_t\beta + (N_t\alpha \dot{-} N_h\beta)$,

3. $N_h((\alpha\beta)\gamma) = N_h(\alpha(\beta\gamma))$,

4. $N_t((\alpha\beta)\gamma) = N_t(\alpha(\beta\gamma))$,

for any strings of characters α, β, γ over $\mathfrak{a}$.

Theorem: Every double stratification of an alphabet $\mathfrak{a}$ determines an equivalence-relation in the extent of the complete string language, $\Sigma\mathfrak{a}$, of all finite strings of characters over $\mathfrak{a}$: α is equivalent to β if and only if $N_h\alpha = N_h\beta$ and $N_t\alpha = N_t\beta$. This equivalence-relation partitions the language $\Sigma\mathfrak{a}$ into (usually infinitely many distinct) decidable languages $\mathfrak{I}_{ij}\mathfrak{a}$ whose extents are those strings for which $N_h\alpha = i$ and $N_h\beta = j$, for any pair of natural numbers $i \geq 0$, $j \geq 0$.

Theorem: (The replacement theorem) If $\alpha, \beta_1, \beta_2, \gamma$ are four strings of $\Sigma\mathfrak{a}$, and if β_1 is equivalent to β_2, then $\alpha\beta_1\gamma$ is equivalent to $\alpha\beta_2\gamma$.

Definition: The 'extent' of a language $\mathfrak{L}\mathfrak{a}$ means the set of strings over $\mathfrak{a}$ belonging to $\mathfrak{L}\mathfrak{a}$. The extent of $\mathfrak{L}\mathfrak{a}$ is designated by the symbol '$\langle\mathfrak{L}\mathfrak{a}\rangle$'. The 'intent' of a language $\mathfrak{L}\mathfrak{a}$ at a given time is the set of processors for it. A set of strings of $\Sigma\mathfrak{a}$ without assigned processors is not called a language (some might choose to call it a 'code'). Thus every mechanical language has a non-null intent, and, more specifically, every one has at least a 'generator', a 'recognizer', or both. (The generator might be a translator from another language.) Furthermore, as new

processors are designed, the intent of a language increases while its extent remains fixed.

The (non-commutative, but associative) language operator called 'the concatenating cross-product' combines the generators of two language functions, $\mathfrak{L}_1$ and $\mathfrak{L}_2$ to form a generator of the language function '$\chi\,\mathfrak{L}_1\mathfrak{L}_2$' which, when applied to an alphabet $\mathfrak{a}$ in the intersection of the domains of $\mathfrak{L}_1$ and $\mathfrak{L}_2$ produces the extensional cross-product of $\langle\,\mathfrak{L}_1\mathfrak{a}\rangle$ and $\langle\,\mathfrak{L}_2\mathfrak{a}\rangle$ familiar in Backus Normal Form: $\langle\chi\,\mathfrak{L}_1\mathfrak{L}_2\mathfrak{a}\rangle = \langle\mathfrak{L}_1\mathfrak{a}\rangle\langle\,\mathfrak{L}_2\mathfrak{a}\rangle$. The difference in intent from the cross-product familiar to mathematicians is that the words of $\mathfrak{L}_1\mathfrak{a}$ and $\mathfrak{L}_2\mathfrak{a}$ are concatenated rather than paired, so that the component words may possibly be only ambiguously determined. As is usual for associative operators, we will write $\mathfrak{L}^n$ for an $n-1$-fold iterated product of $\mathfrak{L}$ with itself, where $n \geq 1$.

The (commutative and associative) language operator called 'the union' combines the generators and combines the recognizers of two language functions, $\mathfrak{L}_1$ and $\mathfrak{L}_2$, to form a generator and, if possible, a recognizer of the language function '$\cup\,\mathfrak{L}_1\mathfrak{L}_2$' which when applied to an alphabet $\mathfrak{a}$ in the intersection of the domains of $\mathfrak{L}_1$ and $\mathfrak{L}_2$ produces or recognizes the extensional union of $\langle\mathfrak{L}_1\mathfrak{a}\rangle$ and $\langle\mathfrak{L}_2\mathfrak{a}\rangle$ familiar in Backus Normal Form: $\langle\cup\,\mathfrak{L}_1\mathfrak{L}_2\mathfrak{a}\rangle = \langle\mathfrak{L}_1\mathfrak{a}\rangle|\langle\mathfrak{L}_2\mathfrak{a}\rangle$. In general we do not have an infinite iterated union, but do write $\mathfrak{L}^\omega$ for $\bigcup_{n=1}^{\infty}\mathfrak{L}^n$.

We also extend the intent of the symbols '$\subseteq$' and '$\subset$', so that '$\mathfrak{L}_1 \subseteq \mathfrak{L}_2$' and '$\mathfrak{L}_1 \subset \mathfrak{L}_2$' mean that the domain of alphabets of $\mathfrak{L}_1$ is contained in the domain of alphabets of $\mathfrak{L}_2$ and, for each such alphabet $\mathfrak{a}$, $\langle\mathfrak{L}_1\mathfrak{a}\rangle$ is contained in or is properly contained in $\langle\mathfrak{L}_2\mathfrak{a}\rangle$ respectively.

Finally we let Λ be the language function for generation and recognition of null words (cleared registers) over an alphabet, and we let $\mathfrak{s}$ be the language function generating and recognizing all single character strings from the alphabet $\mathfrak{a}$. Thus $\mathfrak{s}\,\mathfrak{a}$ is isomorphic in extent with $\mathfrak{a}$, and $\mathfrak{s}^\omega = \Sigma$.

Further theorems about doubly-stratified alphabets are then:

Lemma DS1: $\chi\,\mathfrak{d}_{ij}\,\mathfrak{d}_{kl} \subseteq \mathfrak{d}_{mn}$, where

$$\begin{cases} m = i + (k \dot{-} j), \\ n = l + (j \dot{-} k). \end{cases}$$

Thus, either $j > k$, in which case $m = i$ and $n > l$,
or $j = k$, in which case $m = i$ and $n = l$,

or $j < k$, in which case $m > i$ and $n = 1$.

In any case, no matter what the lengths of the strings α and β of $\Sigma\mathfrak{a}$, either $N_h(\alpha\beta) = N_h\alpha$, or $N_t(\alpha\beta) = N_t\beta$, or both. Finally, in scanning a word of $\Sigma\mathfrak{a}$ from left to right, N_h of the successive heads cannot decrease, and, in scanning from right to left, N_t of the successive tails cannot decrease.

Lemma DS2: If $n > 0$, $\mathfrak{z}^n_{ij} \subseteq \begin{matrix} \mathfrak{z}\, i,\, nj - (n-1)i \text{ if } i \leqslant j, \\ \mathfrak{z}\, ni - (n-1)j,\, j \text{ if } i \geqslant j. \end{matrix}$

Corollary DS1: If $n > 0$, $\mathfrak{z}^n_{ij} \subseteq \mathfrak{u}\mathfrak{z}_{ii}\,\mathfrak{z}_{jj}$ if and only if $i = j$.

Thus like Σ, $\mathfrak{z}^n_{ii} \subseteq \mathfrak{z}_{ii}$.

Corollary DS2: $\mathfrak{z}^n_{10} \subseteq \mathfrak{z}_{no}$.

Corollary DS3: $\bigcup_{n=0}^{\infty} \chi\, \mathfrak{z}_{1n}\, \mathfrak{z}^n_{10} \subseteq \mathfrak{z}_{10}$.

Application of these theorems to simply-stratified alphabets yields rapidly a number of properties of the complete prefix language function $\wp$.

Definition: If $\mathfrak{a}$ is a simply-stratified alphabet, and the stratification of a character c of $\mathfrak{a}$ is designated by $n_t c$, then the 'complete prefix language' over $\mathfrak{a}$, $\wp\mathfrak{a}$, is the one specified by the following generator: if $n_t c = n$ and $\alpha_1, \ldots, \alpha_n$ are words of $\wp\mathfrak{a}$, then so is the word '$c\alpha_1 \ldots \alpha_n$'. When $n_t c = 0$, this is interpreted to mean that the one character string 'c' is also a word of $\wp\mathfrak{a}$.

If no character of $\mathfrak{a}$ has stratification 0, the extent of $\wp\mathfrak{a}$ is null: $\mathfrak{a}^{(o)} = \Lambda \Rightarrow <\wp\mathfrak{a}> = <\Lambda\mathfrak{a}>$. If $\mathfrak{a}$ possesses at least one character of stratification 0, but no character of any other stratification number, then the extent of $\wp A$ is merely the set of single character strings over $\mathfrak{a}$, i.e. is isomorphic to $\mathfrak{a} : \mathfrak{a} = \mathfrak{a}^{(o)} \neq \Lambda \Rightarrow <\wp\mathfrak{a}> = <s\mathfrak{a}>$.

These two cases are considered trivial examples of $\wp$. In any case, let us make a doubly-stratified alphabet out of $\mathfrak{a}$ by taking $n_h c = 1$ for every c in $\mathfrak{a}$. Let us also designate by $\mathfrak{a}^{(j)}$ all characters of $\mathfrak{a}$ for which $n_t c = j$. Thus $s\mathfrak{a}^{(j)} \subseteq \mathfrak{z}_{ij}\mathfrak{a}$, and $\mathfrak{a} = \bigcup_{j=o}^{\infty} \mathfrak{a}^{(j)}$. Also, while $\mathfrak{z}_{oo} = \Lambda$, $\mathfrak{z}_{oj}\mathfrak{a}$ is null for $j > 0$. When N_h and N_t are determined by extension to $\Sigma\mathfrak{a}$ from a simply-stratified alphabet by taking $n_h \equiv 1$ in this manner, we will call

them the head and tail deficiency of a word over α, $\delta_h\alpha$ and $\delta_t\alpha$; except for $\delta_h\alpha = 1$, this designation will be justified shortly.

The recursive definition of the complete prefix language function then reads:

$$\wp = \bigcup_{n=0}^{\infty} \chi\, \alpha^{(n)}\, \wp n$$

Lemma SS1: If $\chi\, \mathfrak{z}_{ij}\, \mathfrak{z}_{kl} \subseteq \mathfrak{z}_{10}$ (where $i > 0$), then $i = 1$, $j = k$, and $l = 0$; Thus a head of any word of $\mathfrak{z}_{10}\alpha$ has $\delta_h\alpha = 1$, and a tail of any such word has $\delta_t\alpha = 0$.

Corollary SS1: If $\chi\, \mathfrak{z}_{10}\, \mathfrak{z}_{kl} \subseteq \mathfrak{z}_{10}$, then $k = l = 0$; i.e. no word of $\mathfrak{z}_{10}\alpha$ is proper head of another ($\mathfrak{z}_{10}$ is simply-analyzable), and therefore $\mathfrak{z}_{10}$ is uniquely deconcatenable.

Theorem SS1: $\wp \subseteq \mathfrak{z}_{10}$, i.e. if α is a word of $\wp\alpha$, then $\delta_h\alpha = 1$ and $\delta_t\alpha = 0$.

The proof is by recursion, since $s\alpha^{(0)} \subseteq \mathfrak{z}_{10}\ \alpha$, $s\alpha^{(n)} \subseteq \mathfrak{z}_{1n}\alpha$, whence the productions of all words of $\wp\alpha$ leave the 'property' $\mathfrak{z}_{10}$ invariant by corollary DS3.

Corollary SS2: In any complete prefix language, no word is a proper head of another; thus every complete prefix language is simply-analyzable, and therefore uniquely deconcatenable.

Lemma SS2: $\chi\, \mathfrak{z}_{ij}\, \mathfrak{z}_{1l} \subseteq \mathfrak{z}_{mn}$, where $\begin{cases} m = i + (1 \dot{-} j), \\ n = l + (j \dot{-} 1), \end{cases}$ so that either $j = 0$ and $m = i + 1$, or $j > 0$ and $m = i$. In particular, in scanning any word of $\Sigma\alpha$ from left to right, as each character is added, the head deficiency can increase at most by 1, and that only when the tail deficiency has reached 0.

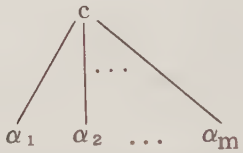
Corollary SS3: Every word of $\mathfrak{z}_{n0}\alpha$ is uniquely deconcatenable into n words of $\mathfrak{z}_{10}\alpha$.

The fact that it is deconcatenable at all follows from lemma SS2; the unicity follows corollary SS1.

Corollary SS4: $\mathfrak{z}_{n0} = \mathfrak{z}_{10}^{n}$ and $\mathfrak{z}_{10} = \bigcup_{n=0}^{\infty} \chi\, \mathfrak{z}_{1n}\, \mathfrak{z}_{10}^{n}$; in greater detail, if c is the head character of the word α of $\mathfrak{z}_{10}\alpha$ and $\delta_t c = n$, then α is uniquely deconcatenable into $\alpha = c\alpha_1\alpha_2 \ldots \alpha_n$ where α_i belongs to $\mathfrak{z}_{10}\alpha$.

The first equality follows from corollary DS2 and corollary SS3, the last statement from lemma SS1 with $i = 1$ and $j = n$. The second equality is then obvious in view of corollary DS3.

Definition: We define the depth of a word of $\mathfrak{z}_{10}\mathfrak{a}$ (and therefore of $\mathfrak{p}\mathfrak{a}$ by theorem SS1), depth α by recursion. If α is a single character word of $\mathfrak{s}\,\mathfrak{a}^{(o)} \subseteq \mathfrak{z}_{10}\mathfrak{a}$, we let depth $\alpha = 0$. Otherwise, by corollary SS4, α is uniquely representable in the form $\alpha = c\alpha_1\alpha_2 \ldots \alpha_m$ where c is a character and α_i are words of $\mathfrak{z}_{10}\mathfrak{a}$; in this case we define depth $\alpha = 1 + \max_i$ depth α_i. In effect corollary SS4 permits translation of any word of $\mathfrak{z}_{10}\mathfrak{a}$ into a tree, where, recursively $\alpha = c\alpha_1 \ldots \alpha_m$ translates into



and one repeats the process with each α_i until only end-points, labelled by characters of $\mathfrak{a}^{(o)}$ are left. The depth of α is then the depth of this tree in the sense that it is the maximum number of descending branches possible from the root to an end-point.

The definition of depth α is unique by corollary SS4.

Theorem SS2: $\mathfrak{p} = \mathfrak{z}_{10}$.

In view of theorem SS1, we have only to prove that $\mathfrak{z}_{10} \subseteq \mathfrak{p}$. This is done by induction on depth. For words of depth 0, i.e. $\mathfrak{s}\,\mathfrak{a}^{(o)}$, this is a result of the production rule for $\mathfrak{p}$. If it is true that every word of $\mathfrak{z}_{10}\mathfrak{a}$ of depth $\leq n$ is a word of $\mathfrak{p}\mathfrak{a}$, then by corollary SS4, if α in $\mathfrak{z}_{10}\mathfrak{a}$ is of depth $n + 1$, $\alpha = c\alpha_1 \ldots \alpha_m$, where c is in $\mathfrak{a}^{(m)}$, $\mathfrak{s}\,\mathfrak{a}^{(m)} \subseteq \mathfrak{z}_{1m}\mathfrak{a}$, and α_i for $i = 1, \ldots, m$ are words of $\mathfrak{z}_{10}\mathfrak{a}$ of maximum depth n; by the induction hypothesis they are words of $\mathfrak{p}\mathfrak{a}$, whence, by the production for $\mathfrak{p}$, so is α. q.e.d.

Lemma SS3: If $\alpha c \beta$ is a word of $\mathfrak{p}\mathfrak{a}$, where α and β are strings of $\Sigma\mathfrak{a}$ and c is a character of $\mathfrak{a}$, then β is uniquely deconcatenable into strings β_1 and β_2 (either or both might be the null string) such that $c\beta_1$ is also a word of $\mathfrak{p}\mathfrak{a}$; in short, every character of a word of $\mathfrak{p}\mathfrak{a}$ is the head of a uniquely determined connected substring of that word which is also a word of $\mathfrak{p}\mathfrak{a}$.

For, by lemma SS1, $c\beta$ belongs to $\mathfrak{z}_{n_0}$ for some $n > 0$; by corollary SS3, there is a unique head of $c\beta$, $c\beta_1$ belonging to

$\mathfrak{z}_{10}\mathfrak{a} = \mathfrak{p}\mathfrak{a}$. The extreme cases are β_1 the null string if c belongs to $\mathfrak{a}^{(o)}$, and $\beta_1 = \beta$ if $n = 1$. q.e.d.

Definition: The unique connected substring of lemma SS3 beginning with the character c in a particular position of a word of $\mathfrak{p}\mathfrak{a}$ is called the *'scope of that character in that word at that position'*. It represents the unique subtree of the tree which has that c as its root. By the *'height of the character'* c in a particular position of the word α of $\mathfrak{p}\mathfrak{a}$ we mean the depth of its scope in α at that position. The characters at height 0 are precisely those belonging to $\mathfrak{a}^{(o)}$ in that word, i.e. the end-points of the tree. A character c_2 in a particular position of the word α of $\mathfrak{p}\mathfrak{a}$ is said to be *'a descendent'* of the character c_1 in a particular position of α if it is within the scope of that c_1. Under the same conditions, that c_1 is called a *'tree predecessor'* of that c_2. That c_1 is called *'the immediate tree predecessor'* of that c_2 if it is the first character to the left of that c_2 in α within whose scope that c_2 may be. (Only the head character has no tree predecessor.) In this last case that c_2 is called *'an immediate tree descendent'*. By the *depth of a character* c in a particular position of the word α of $\mathfrak{p}\mathfrak{a}$ we mean the number of its tree predecessors, i.e. the length of the unique chain connecting it to the root of the tree. Clearly the depth of a word α of $\mathfrak{p}\mathfrak{a}$ is the depth of its deepest characters. Thus, the height of a particular character of a word α of $\mathfrak{p}\mathfrak{a}$ is the length of the longest chain of descendents in α.

Corollary SS5: Every character in a particular position in a word α of $\mathfrak{p}\mathfrak{a}$ is the head of a unique scope in α.

Corollary SS6: Every word of $\mathfrak{p}\mathfrak{a}$ is a scope.

Proof: By corollary SS5, the head character of any word determines a scope. By corollary SS1, it must be the whole word.

Corollary SS7: If σ_1 and σ_2 are two scopes of the same word α of $\mathfrak{p}\mathfrak{a}$, then, either one is a part of the other, or they have no characters in common; i.e. the pairs of end-characters of two scopes do not 'separate' each other. (Roughly speaking—scopes do not overlap.)

Proof: If $\beta\gamma\delta$ is a part of the word α such that $\beta\gamma = \sigma_1$ is a scope of α and $\gamma\delta = \sigma_2$ is also a scope of α, then $\beta\gamma$ and $\gamma\delta$ belong to $\mathfrak{z}_{10}\mathfrak{a}$ by theorem SS2. By two applications of lemma SS1, γ belongs to $\mathfrak{z}_{n0}\mathfrak{a}$ and to $\mathfrak{z}_{1m}\mathfrak{a}$, while δ belongs to $\mathfrak{z}_{m0}\mathfrak{a}$. Thus $n = 1$

and m = 0. It follows that δ, as an element of $\mathfrak{z}_{00}\mathfrak{a}$, must be the null string. This means that σ_2 is part of σ_1. q.e.d.

Corollary SS8: Every scope of every word of $\mathfrak{P}\mathfrak{a}$ is a word of $\mathfrak{P}\mathfrak{a}$.

Corollary SS9: No scope is proper head of another. In particular, no scope of any word is proper head of another word.

Corollary SS10: If $\mathfrak{a}^{(o)}$ is not null, every character is head of some word of $\mathfrak{P}\mathfrak{a}$. For the characters of $\mathfrak{a}^{(o)}$ itself, these words are exactly those of $\mathfrak{s}\mathfrak{a}^{(o)} = \mathfrak{P}\mathfrak{a}^{(o)}$.

Theorem SS3: Any word of $\mathfrak{P}\mathfrak{a}$ remains a word of $\mathfrak{P}\mathfrak{a}$ when any of its scopes is replaced by any word of $\mathfrak{P}\mathfrak{a}$.

This is an immediate consequence of the replacement theorem for doubly-stratified alphabets in view of corollary SS8 and theorem SS2. For all words of $\mathfrak{z}_{10}$ are equivalent (it is an equivalence-class).

Corollary SS11: (Word Expansion) If $\alpha c \gamma$ is a word of $\mathfrak{P}\mathfrak{a}$, c is a character of $\mathfrak{a}^{(o)}$, and σ is a word of $\mathfrak{P}\mathfrak{a}$, then $\alpha\sigma\gamma$ is a word of $\mathfrak{P}\mathfrak{a}$.

Corollary SS12: (Word Contraction) If σ is a scope of the word $\alpha\sigma\gamma$ of $\mathfrak{P}\mathfrak{a}$, and if c is a character of $\mathfrak{a}^{(o)}$, then $\alpha c \gamma$ is a word of $\mathfrak{P}$a.

Theorem SS4: Every word of $\mathfrak{z}_{ij}\mathfrak{a}$, α, is uniquely deconcatenable into i words, $\alpha = \alpha_1\alpha_2\ldots\alpha_{i-1}\alpha_i$, such that α_k is a word of $\mathfrak{P}\mathfrak{a} = \mathfrak{z}_{10}\mathfrak{a}$ for $k = 1, \ldots, i-1$, and α_i belongs to $\mathfrak{z}_{1j}\mathfrak{a}$; thus

$$\mathfrak{z}_{ij} = \chi\, \mathfrak{z}_{10}^{i-1}\, \mathfrak{z}_{1j}.$$

Proof: By lemma SS2, by scanning α from left to right, there must be $i-1$ places at which $\delta_t = 0$ in order that δ_h may become i. This yields the head $\alpha_1\alpha_2\ldots\alpha_{i-1}$ belonging to $\mathfrak{z}_{i-1,0}$ by corollary SS4. Let α_i belong to $\mathfrak{z}_{kl}\mathfrak{a}$. By lemma DS1, $i = i - 1 + k$, and $j = 1$; i.e. $k = 1$ and $l = j$. q.e.d.

Corollary SS13: If $\mathfrak{a}^{(o)} \cup \mathfrak{a}^{(1)} \subset \mathfrak{a}$, then every string α over $\mathfrak{a}$ is part of some word of $\mathfrak{P}\mathfrak{a}$, $\beta\alpha\gamma$.

Proof: Let $\delta_h\alpha = i$ and $\delta_t\beta = j$. We need only select words β and γ belonging to $\mathfrak{z}_{1i}\mathfrak{a}$ and $\mathfrak{z}_{j0}\mathfrak{a}$ respectively. By hypothesis, there is a character of $\mathfrak{a}$ whose stratification number is greater than 1; let it be c, with $n_t c = n > 1$. Also let c_0 belong to $\mathfrak{a}^{(o)}$. We can then construct a string, β_1, belonging to $\mathfrak{z}_{12}\mathfrak{a}$ as follows: if

$n = 2$, take $\beta_1 = c$; if $n > 2$, take $\beta_1 = cc_0 \ldots c_0$, repeating c_0 $n-2$ times. Applying lemma DS2, we then find β belonging to $\mathfrak{z}_{1i}\alpha$ by taking $\beta = \beta_1 \ldots \beta_1$, repeating β_1 $i - 1$ times. For γ we merely take $c_0 \ldots c_0$, repeating c_0 j-times. q.e.d.
Note: corollary SS13 explains the terminology 'tail-deficiency' for $\delta_t\alpha$ and 'head-deficiency' when $\delta_h\alpha > 1$.

Corollary SS14: If α is not trivially stratified, $\wp\alpha$ is saturated in $\Sigma\alpha$ with respect to unique deconcatenability, i.e. if the word α of $\mathfrak{z}_{ij}\alpha$ does not belong to $\wp\alpha = \mathfrak{z}_{10}\alpha$, and if $\beta = c_0 \ldots c_0$, where c_0 is a character of $\alpha^{(\omega)}$ repeated j-times (β is the null string if $j = 0$), then $\alpha\beta$ is ambiguously deconcatenable into words of $\wp\alpha$ and α.

For $\alpha\beta = \alpha_1 \ldots \alpha_{i-1}\alpha_i c_0 \ldots c_0$ has the ambiguous deconcatenations

1. α, and $\alpha_1, \ldots, \alpha_i$ if $j = 0$
2. $\alpha\beta$, and $\alpha_1, \ldots, \alpha_{i-1}, \alpha_i c_0 \ldots c_0$ if $j \neq 0$ but $i > 1$. q.e.d.

AXIOMS FOR GENERALIZED PREFIX LANGUAGES

Corollaries SS-2, and SS5-12 inclusive can all be stated independently of the generative structure of the complete prefix languages. We will see that a number of them actually characterize such languages. In all of them, the key words are: 'word (of the language)', 'scope (of a character in a particular position of a particular word)', 'character (of the alphabet of the language)', and 'replacement (of a scope by an 'equivalent' scope) in a word'. Let us use these as undefined words, and develop the prefix languages in the axiomatic manner.

Let α be an alphabet whose elements will be called 'characters' (it may be infinite because α is a decidable language over a prior alphabet). Let $\mathfrak{L}\alpha$ be a linear sequential language over α; its extent, $\langle \mathfrak{L}\alpha \rangle$, will therefore be a set of finite strings of characters chosen from α, each of which we will call a 'word'. Among the finite class of all connected substrings of each word α we designate a subclass called 'the scopes in α'. If σ_1 is a scope in the word $\alpha = \alpha_1\sigma_1\beta_1$, then there may be many words with head α_1 and tail β_1 with a scope in between; all such scopes are said to be equivalent modulo the context $\alpha_1\sigma\beta_1$, and the string $\alpha_1\sigma\beta_1$ with the scope variable character, σ, determines a 'scope type'. Thus if σ_1 is a scope of the word $\alpha' = \alpha_1\sigma_1\beta_1$ and σ_2 is a scope of

the word $\alpha'' = \alpha_1\sigma_2\beta_1$, then σ_1 and σ_2 belong to a scope type which is an equivalence-class modulo the 'context' $\alpha_1\sigma\beta_1$.

Definition: $\mathfrak{L}\mathfrak{A}$ is called a (generalized) prefix language if the following three axioms hold:

Axiom 1: No scope of any word is a proper head of any word (SS9).

Axiom 2: Any character in any position of any word is the head character at that position of a unique scope in that word. It is called the scope for that word of that character in that position (SS5).

Axiom 3: If a character of a word is within a scope of that word, its scope in that word is a substring of that scope (monotonicity, or non-overlapping, of scopes; SS7).

Some immediate consequences of these axioms are the following:

Lemma GP1: The scope determined by the first character of any word is the whole word. Briefly, every word is a scope (SS6).

Follows from axioms 1 and 2.

Lemma GP2: No word is proper head of another. Thus every language fulfilling axioms 1 and 2 is simply-analyzable, and hence uniquely deconcatenable (SS2).

This follows from lemma 1 and axiom 1.

Lemma GP3: Every word translates into a unique tree with one of its characters at each node; the inverse translation recovers the word by reading the nodes of the tree in prefix fashion (compare with corollary SS4).

Proof: Let c be the head character of the word α. We define the depth of c in α to be 0. Let the second character of α be c_1 and let its scope in α be σ_1. Thus $c\sigma_1$ is a head of α and c_1 is head of σ_1; if it is a proper head, let c_2 be the next character of α, and let σ_2 be its scope. Thus $c\sigma_1\sigma_2$ is a head of α, and c_2 is head of σ_2. Continuing in this way we find that $\alpha = c\sigma_1\sigma_2 \ldots \sigma_{n_0}$, where σ_j has head character c_j. (If $\alpha = c$, $n_0 = 0$). We now define the stratification of c as head of α to be n_0, and the depths of c_j in α to be 1.

Note that so far we have used only lemma 1 and axiom 2, i.e. axioms 1 and 2. It is in the further analysis of the σ_j that axiom 3 becomes involved.

Let the second character of σ_j (reading from left to right) be c_{j_1}, and let σ_{j_1} be its scope in α. By axiom 3, $c_j\sigma_{j_1}$ is a head of σ_j if it is a proper head, let the next character to the right in σ_j (and hence in α) be c_{j_2} and its scope in α be σ_{j_2}. By axiom 3, σ_{j_2} is also part of σ_j. Continuing in this way we find that $\sigma_j = c_j\sigma_{j_1} \ldots \sigma_{jn_j}$, where each σ_{jk_j} has c_{jk_j} at its head. We now define the stratification of c_j in α to be n_j and the depths of c_{jk_j} to be 2 for $1 \leq j \leq n_0$, $1 \leq k_j \leq n_j$ (If $\sigma_{jk_j} = c_{jk_j}$, we take $n_j = 0$.)

Continuing in this manner until all scopes are single characters, α has been translated into a tree and each character in each position in α has defined for it three things:

1. A vector of subscripts which we will call its 'tree-address in α'. It is a generalized Dewey decimal naming system for the nodes of the tree of α, called canonical code in Gorn (3).
2. A depth in α, which is the number of branches from the root c to that character.
3. A stratification number for that position of that character in α, which is the number of branches of the tree emanating from it. If that character in that position is an end-point, its stratification number in α at that position is 0.

The lemma is now obvious.

Definitions: The proof of lemma GP3 has given us the definitions of 'tree-address of a character in a certain position in α', stratification in α of a character, and depth of a character in α. It is now clear how we can define 'immediate descendents of a character in α', 'descendents of a character in α', 'the immediate tree predecessor of a character in α', and 'tree predecessors of a character in α'. The depth of a character in α is clearly the number of numbers in its tree-address in α. Also character 'a' in a certain position in α is a tree predecessor of character 'b' if and only if the tree-address of 'a' is a proper head of the tree address of 'b' in α; it is the immediate predecessor if its address is obtained by dropping the right-most character of the address of 'b'. Similar remarks follow for descendents. The depth of a scope σ of α is the difference between the number of characters in the address of its deepest character and the number of characters in the address of its head character (c is assumed to have no address). The height of a character in α is then defined to be the depth of its scope. Every scope in α is a subtree of the α-tree. Charac-

ters appearing at height 0, i.e. end-points, are said to be terminal in their contexts.

Note: All these definitions and results are local to the particular words in the language. In particular, the stratification numbers of the characters may vary, even for different occurrences in the same word. The axioms so far given permit local analysis of each word, as we have seen, but are much too weak, in spite of the unique deconcatenability inherent in axiom 1, to relate words to one another. We cannot even guarantee that some words can appear as scopes (not at the head, by axiom 1) in others. For the languages to have stronger global structures, as we might expect, some replacement properties of words and scopes are necessary.

We will put off examples and counter-examples to the next section, and will now consider such replacement properties.

Definitions: A scope σ_1 with head character c is said to be a kernel in context $\alpha_1 \sigma \beta_1$ if it has minimum depth among all scopes of head c in that scope type. We designate by $\mathfrak{K}^{(n)}$ the kernels of depth n. Among the complete prefix languages, the only non-null kernel classes are $\mathfrak{K}^{(0)} = \mathfrak{s}\mathfrak{a}^{(0)}$, and $\mathfrak{K}^{(1)} = \bigcup_{n=1}^{\infty} \chi \mathfrak{a}^{(n)} \mathfrak{a}^{(0)} n$. In any case, the head characters of $\mathfrak{K}^{(0)}$ have local stratification number 0 in their contexts, while the head characters of $\mathfrak{K}^{(1)}$ have contextual stratification numbers equal to the number of characters following, all in $\mathfrak{K}^{(0)}$, in these kernels.

A prefix language is called 'contractible' if it fulfills axiom 4.

Axiom 4: (Contractibility) Every scope type contains a single character scope (a kernel of depth 0).

Lemma GP4: In a contractible prefix language every scope of more than one character has a contextually equivalent kernel of depth 1 with the same head character.

Proof: One has merely to replace the scopes of the characters at depth 1 successively by contextually equivalent kernels of depth 0. Even though these contexts change with each replacement, the context of the original scope type remains unchanged. q.e.d.

A prefix language is called 'homogeneous at depth 0' if:

Axiom 5: (Depth 0 Homogeneity) All single character scopes are equivalent in every context.

A prefix language is called 'expandible' if:

Axiom 6: (Expandibility) There is a single character scope whose replacement by any word in any context produces a word.

Note, however, that, hitherto, although all words are equivalent in null context, they may still be inequivalent in other contexts. Scope types overlap because they are equivalence-classes for different equivalence-relations. The effect of axioms 4, 5, and 6 is to make all words equivalent in all contexts. For example, a kernel is a kernel only in its context; it is only a word if the context of its scope type is null (i.e. α_1 and β_1 are null).

A prefix language is called 'locally uniform' if it fulfills:

Axiom 7: (Local Uniformity) Every scope is a word.

A prefix language is called 'uniform' if:

Axiom 8: (Uniformity) The stratification number of each character is the same in every context.

Lemma GP5: Every contractible prefix language which is homogeneous at depth 0 must have the following property: the stratification number of the head character of any word must be the minimum of all the stratification numbers of that character.

Proof: If a word has c as its head character, by lemma GP4, there is a kernel word of depth 1 with that character as its head. If a scope with c at its head permitted a smaller stratification number, the equivalent kernel scope of depth 1 would have smaller length than the kernel word. By axiom 5, the scope and the word could be taken with all end-points the same, and this would contradict axiom 1. q.e.d.

Lemma GP6: Every locally uniform, contractible prefix language which is homogeneous at depth 0 is uniform (i.e. axioms 1, 2, 3, 4, 5, 7 imply axiom 8).

This is an immediate consequence of lemma GP5 and axiom 7.

Actually, the effect of axioms 4, 5, 6, and 7 together is to identify all scope types, so that words and scopes are identical and any word may replace any other in any word to produce a new word. Without axiom 6 we therefore obtain sub-languages of complete prefix languages, and with axiom 6 we obtain the complete prefix languages.

Theorem GP1: Every locally uniform, contractible prefix language which is homogeneous at depth 0 may have its alphabet stratified so as to become in extent a subset of a complete prefix language.

Note: In point of fact one may show that the following properties result from axioms 1, 2, 3, 4, 5, 7:

a. No word is proper head of another,
b. Every character in every word is proper head of a connected segment which is also a word (unique by 'a'—it is the scope of that character in that word).
c. If the set of terminal characters of all words is called $\mathfrak{a}^{(o)}$ (they are words, by 'b') and if c is any such character, and if σ_1 and $\alpha_1\sigma_1\beta_1$ are words, then so is $\alpha_1 c \beta_1$ (contraction).

Theorem GP1 follows from these properties alone.

Theorem GP2: Every locally uniform prefix language which is homogeneous at depth zero and both contractible and expandible may have its alphabet stratified so as to become a complete prefix language.

Proof: By theorem GP1, after determination of the stratification numbers of the characters; we have only to prove that every word of $\wp\mathfrak{a}$ is a word of the language. The proof is by induction with respect to depth. Every character of $\mathfrak{a}^{(o)}$ is a word as remarked in item c. If $\sigma_1, \ldots, \sigma_n$ are words of depth $\leq m$, and assumed to be in $\wp\mathfrak{a}$, and if c is a character of stratification number n, then by lemma GP4 and axiom 5 $cc_0 \ldots c_0$ is a scope, and therefore a word (axiom 7), where c_0 is in $\mathfrak{a}^{(o)}$, is the character of axiom 6 and is repeated n-times. Repeated application of axiom 6 to this kernel of depth 1, using the words $\sigma_1, \ldots, \sigma_n$, shows that $c\sigma_1 \ldots \sigma_n$ is a word. But every word of $\wp\mathfrak{a}$ is of this form. q.e.d.

EXAMPLES AND COUNTER-EXAMPLES

Example 1: Our first example is a complete prefix language specifically designed to name all trees [see Gorn (3) & (4)]. Let $\mathfrak{N}$ be any ordinal alphabet, i.e. any number representation system for the integers ≥ 0. Let it be stratified by: $n_{\mathfrak{t}}n = n$. Then $\wp\mathfrak{N}$ names all trees. Any subset of $\mathfrak{N}$, $\mathfrak{a}$, with the same stratification yields a complete prefix language whose words designate all trees whose nodes are restricted to those stratifications only. For example, if $\mathfrak{D}=\{0, 1\}$, then $\wp\mathfrak{D}$ designates all trees with no ramifications at all, i.e. the ordinal numbers themselves: 0, 10, 110, ... We could therefore take advantage of the unique deconcatenability of $\wp\mathfrak{D}$ to be able to do without

'character' separators in $\mathfrak{P}\mathfrak{A}$; take $\mathfrak{A} = \mathfrak{P}\mathfrak{D}$, so that $\mathfrak{P}\mathfrak{A} = \mathfrak{P}\mathfrak{P}\mathfrak{D}$. (This device has been used by Smullyan to simplify Gödel numbering.) If, however, we take $\mathfrak{a} = \{0, 2\}$, then $\mathfrak{P}\mathfrak{a}$ names all binary trees.

Example 2: Our second example seems to have tree structure, but fails to be a prefix language, failing axioms 1 and 2, so that we cannot even define scopes uniquely. We introduce it as a sublanguage of a phrase structure language of type 2, presented in Backus Normal Form (1). Let $\mathfrak{a} = \{1, 2, 3\}$, and let the words of $\mathfrak{L}_1\mathfrak{a}$ be called i-words if they begin with i (i = 1, 2, 3), and be designated by ⟨i⟩. Then

$\langle \mathfrak{L}_1\mathfrak{a} \rangle ::= \langle 1 \rangle \mid \langle 2 \rangle \mid \langle 3 \rangle$
$\langle 1 \rangle ::= 1 \mid \langle 1 \rangle \langle 2 \rangle \langle 3 \rangle$
$\langle 2 \rangle ::= 2 \mid \langle 2 \rangle \langle 3 \rangle \langle 1 \rangle$
$\langle 3 \rangle ::= 3 \mid \langle 3 \rangle \langle 1 \rangle \langle 2 \rangle$

$\mathfrak{L}_1\mathfrak{a}$ is clearly obtained, beginning with the words 1, 2, and 3, by replacing at will, and as often as one pleases, $1 \rightarrow 123$, $2 \rightarrow 231$, and $3 \rightarrow 312$. (We might write $\langle \mathfrak{L}_1\mathfrak{a} \rangle = (1 \mid 2 \mid 3;\ 1 \rightarrow 123,\ 2 \rightarrow 231,\ 3 \rightarrow 312.)$) Clearly, we could consider 1, 2, and 3 as trees of depth 0, and 123, 231, and 312 as trees of depth 1 read in prefix fashion; the language $\mathfrak{L}_2\mathfrak{a}$, a sublanguage of $\mathfrak{L}_1\mathfrak{a}$, could begin with these trees, replace any end-point by the tree of depth 1 beginning with the same character, continue this process indefinitely, and read the prefix name of the resulting tree.

First of all $\langle \mathfrak{L}_2\mathfrak{a} \rangle \subset \langle \mathfrak{L}_1\mathfrak{a} \rangle$; for example, 12323 is obtained from 123 by replacing 1 by 123. Thus 12323 belongs to $\langle \mathfrak{L}_1\mathfrak{a} \rangle$ but not to $\langle \mathfrak{L}_2\mathfrak{a} \rangle$.

It might seem from its generation that $\langle \mathfrak{L}_2\mathfrak{a} \rangle$ is a prefix language; corresponding to each binary tree there is exactly one word each in ⟨1⟩, ⟨2⟩, and ⟨3⟩. However, some words have ambiguous derivations, belonging to more than one tree. For example, 123123123 derives from both binary trees 220202000 and 202020200; the first character has stratification 2 in each case, and the last two characters have stratification 0; all the other characters have one stratification in the first derivation and the other one in the second. Thus, although $\mathfrak{L}_2\mathfrak{a}$ may be defined by three translation procedures from the complete prefix language $\mathfrak{P}\mathfrak{a}$, where $\mathfrak{a} = \{0, 2\}$, it fails axioms 1 and 2. In fact it is obvious that *every* word is head of another, and 'scopes' are not uniquely determined in such 'ambiguous' words.

Example 3: Our third example is another language which names all trees. It is a prefix language with variable stratification numbers. Let $\mathfrak{A}$ be an ordinal alphabet. $\mathfrak{L}_3\mathfrak{A}$ may be specified by the following translation procedure from trees which is recursive by height:

a. label each end-point 0.

b. when all nodes at height $< n$ have been labelled, label each node at height n by adding the stratification number of the node to the sum of the labels of the immediate descendents.

c. when all nodes have been labelled, read the labels in prefix order.

One can give a generative specification of the words of $\mathfrak{L}_3\mathfrak{A}$, without reference to trees, as follows:

1. '0' is a word; its 'rank' is 0.

2. If $\alpha_1, \alpha_2, \ldots, \alpha_n$ are words of rank $r_1, r_2, \ldots, r_n$, and $r = n + r_1 + r_2 + \ldots + r_n$, then '$r\alpha_1\alpha_2\ldots\alpha_n$' is a word of rank r.

Thus the head character of every word is its rank; as a matter of fact, the length of every word is one more than its rank.

A table of all words through rank 4 versus the stratification number of the first character is as follows:

	0	1	2	3	4
0	0				
1		10			
2		210	200		
3		3210	3010	3000	
		3200	3100		
4		43210	42100	41000	40000
		43200	40210	40100	
		43010	42000	40010	
		43100	40200		
		43000	41010		

We see, then, that the character n can have the stratification number 1, 2, ..., n.

This language therefore satisfies axioms 1, 2, 3, 5, 7, but fails 4, 6, and 8: it is locally uniform, and homogeneous at depth 0, but is non-uniform, hence non-contractible; it is also nonexpandible. As a matter fact, the scope types are independent of context; two words are equivalent if and only if they have the same rank. No replacement ever changes the size of a word. $\mathfrak{L}_3\mathfrak{A}$ has the useful property, therefore, that each character states exactly how many characters to its right are within its scope. It can therefore be used as a control language which,

scanned from left to right, predicts storage requirements and maximum depth of recursion.

Example 4: Let $\mathfrak{a} = \{p, a\}$, and let the stratification number of each character depend upon the character immediately to the right of it as indicated in the following table:

		a	p	null -right
left	a	0	0	0
	p	1	2	-

These stratification rules then determine a language $\mathfrak{L}_4\mathfrak{a}$ in which the words at the first four depths are:

0: a
1: pa
2: ppaa, ppapa
3. pppaaa, pppaapa, pppaappaa,
pppaappapa, pppapaa, pppapapa,
pppapappaa, pppapappapa,

Note that 'papa' is not a word.

$\mathfrak{L}_4\mathfrak{a}$ satisfies axioms 1, 2, 3, 5, 6, 7; it is therefore locally uniform and expandible, homogeneous at depth 0, but not uniform, and hence not contractible.

Example 5: In the previous example, if we introduce the new character 'b', always replacing 'pa' by 'b', we have traded homogeneity for uniformity. The effect is to change the depths; the language, however, is incomplete because of the lack of homogeneity: $\mathfrak{a} = \{p, a, b\}$, $n_t p = 2$, $n_t a = n_t b = 0$, and the generating rule is the following- $\langle \mathfrak{L}_5 \mathfrak{a} \rangle$ is composed of those and only those words of $\wp\mathfrak{a}$ in which there is no substring 'pa'. Examples are

0: a, b
1: pba, pbb
2: ppbaa, ppbba, ppbapba,
ppbapbb, ppbba, ppbbb,
ppbbpba, ppbbpbb

$\mathfrak{L}_5\mathfrak{a}$ therefore fulfills axioms 1, 2, 3, 4, 6, 7, 8. It is contractible, but not homogeneous; nevertheless it is uniform.

Example 6: Let $\mathfrak{a}_0 = \{0, 1, 2\}$, $\mathfrak{a}_1 = \{0_1, 1, 2_1\}$, $\mathfrak{a}_2 = \{0_2, 1, 2_2\}$, and $\mathfrak{a} = \mathfrak{a}_0 \cup \mathfrak{a}_1 \cup \mathfrak{a}_2 = \{0, 0_1, 0_2, 1, 2, 2_1, 2_2\}$.

Let, further 0, 0_1, and 0_2 have stratification number 0, 2, 2_1, and 2_2 have stratification number 2, and 1 have stratification number 1. The complete prefix languages $\mathfrak{P}\mathfrak{a}_0$, $\mathfrak{P}\mathfrak{a}_1$, and $\mathfrak{P}\mathfrak{a}_2$ are therefore completely specified. Let us now define the following syntactic properties of the characters '0' in the words of $\mathfrak{P}\mathfrak{a}_0$:

A character '0' in a particular position in a word α of $\mathfrak{P}\mathfrak{a}_0$ is called:

1. 'left-free' if its immediate tree predecessor is '2' and it appears as the first scope of that '2', reading from left to right,
2. 'right-free' if its immediate tree predecessor is '2' and it appears as the second scope of that '2',
3. 'bound' otherwise.

Thus '0' is bound in itself, and in the word '221002010', the first and fourth '0' reading from left to right are bound, the second is right-free, and the third is left-free.

Now define $\mathfrak{L}_6\mathfrak{a}$ as follows: any word of $\mathfrak{L}_6\mathfrak{a}$ is obtained from a word of $\mathfrak{P}\mathfrak{a}_0$ by replacing each left-free '0' by a word of $\mathfrak{P}\mathfrak{a}_1$, and each right-free '0' by a word of $\mathfrak{P}\mathfrak{a}_2$. The replacements are obligatory, so that no word of $\mathfrak{L}_6\mathfrak{a}$ can have the character '0' appearing as an immediate descendent of a character '2'.

$\mathfrak{L}_6\mathfrak{a}$ then fulfills axioms 1, 2, and 3; it is therefore a prefix language. Furthermore, it is uniform (axiom 8) but is not locally uniform (axiom 7) because the words of $\mathfrak{P}\mathfrak{a}_1$ and $\mathfrak{P}\mathfrak{a}_2$ are scopes in $\mathfrak{P}\mathfrak{a}$ but are not words in $\mathfrak{P}\mathfrak{a}$. It is, however, expandible (axiom 6) because '0_1' and '0_2' may be replaced by any words in $\mathfrak{P}\mathfrak{a}_1$ and $\mathfrak{P}\mathfrak{a}_2$ respectively; the replacement conditions of '0' are, however, not obvious. $\mathfrak{L}_6\mathfrak{a}$ is not contracting (axiom 4); the replacement of the scope '10' is context dependent. Also $\mathfrak{L}_6\mathfrak{a}$ is not homogeneous at depth 0 (axiom 5), because '0', '0_1', and '0_2' cannot replace one another. Thus $\mathfrak{L}_6\mathfrak{a}$ fulfills axioms 1, 2, 3, and 6; it fails in axioms 4, 5, and 7; nevertheless it is uniform (axiom 8).

Suppose, now, that we removed the uniformity from $\mathfrak{L}_6\mathfrak{a}$, by replacing all characters '1' appearing in $\mathfrak{P}\mathfrak{a}_1$ and $\mathfrak{P}\mathfrak{a}_2$ scopes by the character '2'. In other words, let $\mathfrak{a}_1' = \{0_1, 2, 2_1\}$ and $\mathfrak{a}_2' = \{0_2, 2, 2_2\}$, where the stratification of '2' is to be 1 in $\mathfrak{a}_1'$ and $\mathfrak{a}_2'$, while it is 2 in $\mathfrak{a}_0$, and let the generation rule of $\mathfrak{L}_7\mathfrak{a}$ be exactly the same as that of $\mathfrak{L}_6\mathfrak{a}$ except that we use $\mathfrak{P}\mathfrak{a}_1'$ and $\mathfrak{P}\mathfrak{a}_2'$ instead of $\mathfrak{P}\mathfrak{a}_1$ and $\mathfrak{P}\mathfrak{a}_2$. In this case we have lost the property of simple analyzability (axiom 1); for example the word '220_10_2', in which the second character has stratification 1, is a proper head of the word '$220_10_20_2$', in which the second character has stratification number 2.

REFERENCES

1. Backus, J. W. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference, ICIP Proceedings, Paris, June 1959.
2. Curry and Feys. Combinatory Logic, Vol. I, Amsterdam, The Netherlands, North Holland, 1958.
3. Gorn, S. Common Programming Language Task, Final Reports AD59UR1 and AD60UR1, U.S. Army Signal Corps Contract No. DA-36-039-SC75047, Part I, 1959, 1960.
4. Gorn, S. The Treatment of Ambiguity and Paradox in Mechanical Languages, Proceedings of Symposia in Pure Mathematics, Vol. V, Recursive Function Theory, August 1962.

A TRANSLATION TECHNIQUE FOR LANGUAGES WHOSE SYNTAX IS EXPRESSIBLE IN EXTENDED BACKUS NORMAL FORM*

Peter Zilahy Ingerman

University of Pennsylvania
Moore School of Electrical Engineering
Philadelphia, Pennsylvania, USA

I. BACKUS NORMAL FORM

Introduction

Backus Normal Form (abbreviated BNF) is the metalanguage used to define the syntax of Algol (16). In the Algol report BNF is defined first by example:

<ab> : : = (|[|<ab>(|<ab><d>

Sequences of characters enclosed in the brackets <> represent metalinguistic variables whose values are sequences of symbols. The marks : : = and | (the latter with the meaning of *or*) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted.

This description can be made rigorous. We assert a set of "basic symbols"

$$B = \{b_1, b_2, - - -, b_m\} \tag{1.0}$$

where b_1 is the null string, and a set of "names"

$$N = \{n_1, n_2, - - -, n_p\} \tag{1.1}$$

*The work reported in this paper was sponsored jointly by the National Science Foundation under Grant G-14096 and the Air Force Office of Scientific Research under Contract AF-49(638)-951.

where n_1 is the unique name of b_1, and require that

$$N \cap B = \text{the null set} \tag{1.2}$$

We require that N and B both be finite sets, and we require that the strings of elements of $(N \cup B)$ be uniquely deconcatenable into the constituent elements. The first requirement is met in any of the languages which we shall consider. The second requirement, although facilitated in practice by the introduction of an artifice such as the metalinguistic brackets of Algol, can be met without such a device.

We shall use the symbols from symbolic logic, together with juxtaposition to show concatenation, as an informal metametalanguage.* The rules for the formation of rules in BNF can then be written as:

$$n_i \vee b_j \vee e_{s_1} e_{s_2} - - - e_{s_n} \Rightarrow e_k \tag{1.3}$$

$$n_j ::= e_i \Rightarrow r_{ij} \tag{1.4}$$

$$n_1 ::= \Rightarrow r_{11} \tag{1.5}$$

where e_i is an element from a set E of expressions, r_{ij} an element from a set R of rules in BNF, and "::=" is a single symbol here without meaning. In a rule r_{ij} we shall refer to the b_k's and n_p's which (ancestrally) form e_i as *metacomponents* and shall refer to n_j as a *metaresult*.

In the Algol report a notational convenience is introduced:

$$(n_i ::= e_j) \ \& \ (n_i ::= e_k) \equiv (n_i ::= e_j \mid e_k) \tag{1.6}$$

In the metalanguage (BNF) the symbol "::=" is read "names the string". 1.6 in the metametalanguage has the effect of introducing "|" into the metalanguage with the meaning (in the metalanguage) of "or the string".

Rules 1.3, 1.4, and 1.5 give a structural description of well-formed rules in BNF.

BNF As a Phrase Structure Grammar

Note that 1.3 and 1.4 without 1.5 define a Phrase Structure Grammar (PSG) of Type 2, by the classification introduced by Chomsky (3), and can generate only grammars of Type 2. This

---

*Since juxtaposition will be used for concatenation, we shall use "&" for "and".

can be seen as follows: Chomsky defines PSG's as grammars satisfying certain restrictions.

Restriction 1 requires that the rules of the grammar all be of the form

$$e_i n_j e_k ::= e_i e_p e_k \qquad (1.7)$$

where e_p is neither b_1 nor n_1. Such a rule asserts that $n_j ::= e_p$ in the context $e_i - - - e_k$ (which may be null). Restriction 2 requires that the limiting context indeed be null; that is, that the rules all be of the form $n_j ::= e_p$, and that each such rule may be applied independently of the context in which n_j appears. Restriction 3 limits the rules to the form $n_i ::= b_h n_k$ or $n_j ::= b_h$. A grammar satisfying restriction i (i = 1, 2, 3) also satisfies restriction h > i, and is a PSGi. Rules 1.3 and 1.4 together satisfy restriction 2, and the languages whose grammars may be expressed by the use of 1.3 and 1.4 are also PSG2. The addition of rule 1.5 does not per se invalidate this result in cases where the null string can be eliminated by rewriting. Algol is such a case.

The Rules of a Language

We have so far considered the well-formedness of individual r_{ij}'s; we now consider the well-formedness of R(L), the set of rules which are the grammar of some language L with names N(L), basic symbols B(L), and expressions E(L).

We state a restriction on R(L) which we call *enterability*: for every element of N(L) there is at least one string of elements of B(L) which can be generated by a processor designed to generate words in a language given a name and a grammar. This restriction eliminates superfluous elements of N(L); such superfluous names would not affect the processor to be discussed later in this paper, but the names would require space in the tables which the processor must use, and hence would decrease the available memory space of a computer.

A *rule-chain* is defined as a sequence of rules, all elements of R(L), such that the metaresult of the k^{th} rule is one of the metacomponents (or the sole metacomponent) of the $(k+1)^{st}$ rule. Note that 1.3, 1.4, and 1.5 do not prevent a rule-chain from being cyclic (and since rules can have several metacomponents each, the complexity of the cycles that may be formed is quite great).

A slightly different restriction, which we call *cyclic non-*

nullity, is necessary to guarantee that the processor using the grammar R(L) does not cycle non-productively. Cyclic non-nullity requires that if there exist a rule-chain with a loop in it then there is at least one metacomponent of one of the rules in the loop which cannot be (ancestrally) b_1. This restriction asserts, therefore, that on each recursion through a recursive definition at least one non-null character is generated (if generating) or is picked up from the input string (if recognizing).

Last, we define the set of *heads* of a language as the subset of N(L) which appear as metaresults but not as metacomponents. Without loss of generality (since additional rules can be introduced) we shall consider only languages which are *single-headed*.

Finally, a well-formed grammar must be

a) enterable,
b) cyclically non-null, and
c) single-headed

in addition to the general restrictions that the null string have a unique name and that names and basic symbols be pairwise distinguishable. Interestingly enough, Algol fails to satisfy restriction c) and actually has two heads (<basic symbol> and <number>), neither of which seems appropriate for the language; the head which is given implicitly in the text [Naur, AB14] is

<program>::=<statement> (1.8)

and the other two heads are apparently for pedagogical purposes.

The restrictions in the preceding paragraph are sufficient to guarantee the recognizability (in a yes-no sense) of a string, or to generate strings. They do not guarantee that strings are unambiguously recognizable, in the sense that for each recognizable input string there is a unique production tree which was "used" to generate the string originally. In the recognition scheme to be discussed below a grammatical string will cause its rule-tree (in a modified prefix form) to be generated. Ambiguity in the sense of more than one possible rule-tree for any given string will be resolved by selecting the first possible tree encountered by a systematic inspection of the list of rules. (Gorn discusses this problem from a slightly different point of view.) In what follows, we shall assume that no ambiguities of this type exist.

II. THE GRAMMAR IN PROCESSOR FORM

Introduction

In this chapter we shall discuss the algorithms used for deriving the processor form of the grammar from R(L), N(L), and B(L). The grammar in processor form consists of the Syntax Matrix, S, and the Transition Matrix, T. These matrices are used in a processor which will inspect them in a systematic manner, together with the input string, to determine whether the imput string is grammatical.

It is convenient to write the rules in BNF in a slightly altered form to eliminate the symbol "::="; we do this by using, instead of 1.4,

$$e_i n_j \Rrightarrow r_{ij} \tag{2.1}$$

For convenience we define

$$P(L) \equiv B(L) \cup N(L) \tag{2.2}$$

We shall here treat a rule

$$r_{ij} = e_i n_j = e_{s_1} - - - e_{s_n} n_j \tag{2.3}$$

where

$$(\forall i)[e_{s_i} \epsilon P(L)] \tag{2.4}$$

and shall refer to n_j as the metaresult of the rule and e_{s_i} as the i^{th} metacomponent, as we did in the preceding chapter. We define

$$b_i = e_i \text{ for } 1 \leq i \leq m \tag{2.5}$$

$$n_j = e_{m+j} \text{ for } 1 \leq j \leq p \tag{2.6}$$

where m and p are implicitly defined by 1.0 and 1.1. We note that

$$P(L) = \{b_1, - - -, b_m, n_1, - - -, n_p\}$$
$$= \{e_1, - - -, e_m, e_{m+1}, - - -, e_{m+p}\} \tag{2.7}$$

The Formation of the First-Component Path Matrix

We associate e_k with the k^{th} row and column of an $(m+p)$-by-$(m+p)$ matrix whose elements $m_1[i, j]$ are true if e_i is the first metacomponent of a rule whose metaresult is e_j;* the elements of M_1 are false otherwise.

We may note that M_1 is a path matrix in the same sense that

*In the strict sense of the definition, a metaresult must be an element of N(L); when we say, informally, that it is an element of E(L), we mean to imply that, because of 2.6, this particular element of E(L) has a subscript $m + 1 \leq x \leq m + p$.

a rule denotes a path from the first metacomponent to the meta-result. If e_i has been identified in the input string and $m_1[i, j]$ is true, there is a single rule which may allow e_i (and perhaps some other e's) to be rewritten as e_j.

We define the product of Boolean matrices A and B to form a matrix C by

$$c_{ij} = \bigvee_{r=1}^{n} a_{ir} \,\&\, b_{rj} \tag{2.8}$$

Since M_1 is a square matrix, we can form powers of it by 2.8. From switching theory we note that if K is a single-step transition matrix, then k^n is an n-step transition matrix; we shall therefore form

$$B_k = \bigvee_{i=1}^{k} M_1^i \tag{2.9}$$

Warshall (20) gives a proof of a much simpler way of forming B, where B is defined as B_k for some k such that $B_k \equiv B_{k+1}$; this algorithm can be stated as:

```
for j := 1 step 1 until n do                    (2.10)
for i := 1 step 1 until n do
if a[i, j] then
for k: = 1 step 1 until n do
a[i, k] := a[i, k] ∨ a[j, k];
```

which has the advantage of not requiring large numbers of temporary storage locations if implemented for a computer. We apply 2.10 to M_1 to form M_1^*. (Note that e_k is still associated with the k^{th} row and column of M_1^*.)

The processor to be described in the following chapter is *goal-oriented*, in the sense that at each stage of its execution it has available both the particular n_h which it is trying to identify in the input string and the e_k that it has currently identified. The matrix M_1^* (actually, a matrix derived from M_1^*; see below) is used by the processor to answer the question, "Given a rule with e_k as its first metacomponent, is there a rule-chain beginning with that rule and ending with n_h?". If the answer is "yes", $m_1^*[k, h]$ is true; otherwise it is false.

The Formation of the Multi-Component Path Matrices

We shall also form matrices M_2 through M_y where y is the maximum number of metacomponents occurring in any rule

which is an element of R(L). We associate e_k with the k^{th} column of M_2 through M_y. With the rows of M_i we associate the first i metacomponents of rules with at least i metacomponents; analogously to M_1, $m_i[r, s]$ is true if the set of metacomponents associated with the r^{th} row appears as the first i metacomponents of a rule whose metaresult is e_s.

We now form M_i^* from M_i (for $i > 1$). Let q_i be the number of rows in M_i. Then we produce M_i^*:

```
for h: =1 step 1 until q_i do                                    (2.11)
begin for k: =1 step 1 until m + p do
m*_i[h, k]: = m_i[h, k];
for k: =1 step 1 until m + p do
begin if m_i[h, k] then
for j: =1 step 1 until m + p do
m*_i[h, k]: = m*_i[h, k] ∨ m*_1[k, j] end end;
```

and we repeat 2.11 for each of the M_i's.

The reasoning behind 2.11 is simple. If a rule has some n_h as its metaresult, that rule can be considered as the first link in a rule-chain, and the remainder of the rule-chain is that which was derived by considering n_h itself to be the first link. But M_1^* is a matrix which lists all paths from first metacomponents. Hence 2.11 follows.

The multi-component path matrices generated from 2.11 are used by the processor in a manner analogous to the use of M_1^*; if the first k metacomponents of a rule have been identified, inspection of the appropriate element of M_k^* will show whether a path still exists to the desired goal.

The Construction of T

We now adjoin the M_i^*'s, formed by the application of 2.11 to the bottom of M_1^*, in numerical order by i. This matrix, which we shall call M^*, could be used directly for the transition matrix T. However, we are considering the use of T in a processor to be used in a digital computer with a finite memory, and a few additional manipulations may reduce the size of M^* without loosing the information needed by the processor.

First, we note that, as long as the e_i's originally associated with the rows remain so associated, the rows themselves may be rearranged; the same is true for columns. Second, we note that the rows and columns can be arranged into equivalence

classes, by associating with a single row all of the e_i's associated with duplicated rows, and eliminating the duplicated rows; similarly for columns. Finally, we may strike from M^* those rows and columns whose elements are all **false**. The resulting matrix is T.

An Example

As an example, we consider the set of rules in TABLE 2.12. We use lower case roman letters as elements of N(L), all other symbols as elements of B(L) and present R(L) in the form specified by 2.1.

B(L) ={A, - - -, Z, 0, - - - 9, ↑, ¬, +, *, /, $_{10}$, ., (,), :=}
N(L) ={a, - - -, s}

Figure 2.13 through 2.18 give the development of T from R(L) for this example; note the "abbreviations" λ for A---Z and δ for 0---9 are based on the second reduction rule of the previous section. Figures 2.13 and 2.14 are the matrices M_1, M_2, and M_3

TABLE 2.12.1
A sample set of rules

A k	Y k	e g q
B k	Z k	e q
C k	0 f	f r
D k	1 f	g q
E k	2 f	h ↑ o h
F k	3 f	h p
G k	4 f	i k i
H k	5 f	i s
I k	6 f	k i
J k	7 f	l m
K k	8 f	m l m
L k	9 f	m b c
M k	¬ r j	o h
N k	¬ a	p n h p
O k	+ r j	p b
P k	+ a	q o
Q k	* n	r f r
R k	/ n	r d e
S k	$_{10}$ j g	r e
T k	. r d	r j
U k	(b) o	s := l
V k	a p b	s o
W k	b a p b	
X k	d e	

Name	Abbreviation
<adding operator>	a
<arithmetic expression>	b
<assignment statement>	c
<decimal fraction>	d
<decimal number>	e
<digit>	f
<exponent part>	g
<factor>	h
<identifier>	i
<integer>	j
<letter>	k
<left part>	l
<left part list>	m
<multiplication operator>	n
<primary>	o
<term>	p
<unsigned number>	q
<unsigned integer>	r
<variable>	s

FIG. 2.12.2. Abbreviations Used in Table I.

derived directly from the rules in 2.12. Figures 2.15 and 2.16 give the result of applying 2.10 to M_1 and then 2.11 to M_2 and M_3. Figure 2.17 is the result of the elimination of false rows and columns from M*, and Figure 2.18 is the final matrix T formed by the reduction by equivalence classes. Note that the matrix formed from M_1^*, M_2^*, and M_3^* by adjoining rows was 85-by-65, or 5525 elements, while T is only 19-by-18, with 343 elements. In Figure 2.18 the rows and columns have been numbered for later convenience.

The direction of scan enters into the construction of this matrix, at the point where the original M_i's are formed, by considering the left-most metacomponents and the metaresults. If the direction of scan were to have been right-to-left, the original matrices would have been formed by considering the rightmost metacomponents and the metaresult.

The Formation of the Syntax Matrix from the Rules of the Language

We must now consider the formation of the Syntax Matrix S, from the rules of the language. We shall, as in the derivation of T, consider the direction of scan to be left to right. The preceding chapter gave criteria for the well-formedness of R(L).

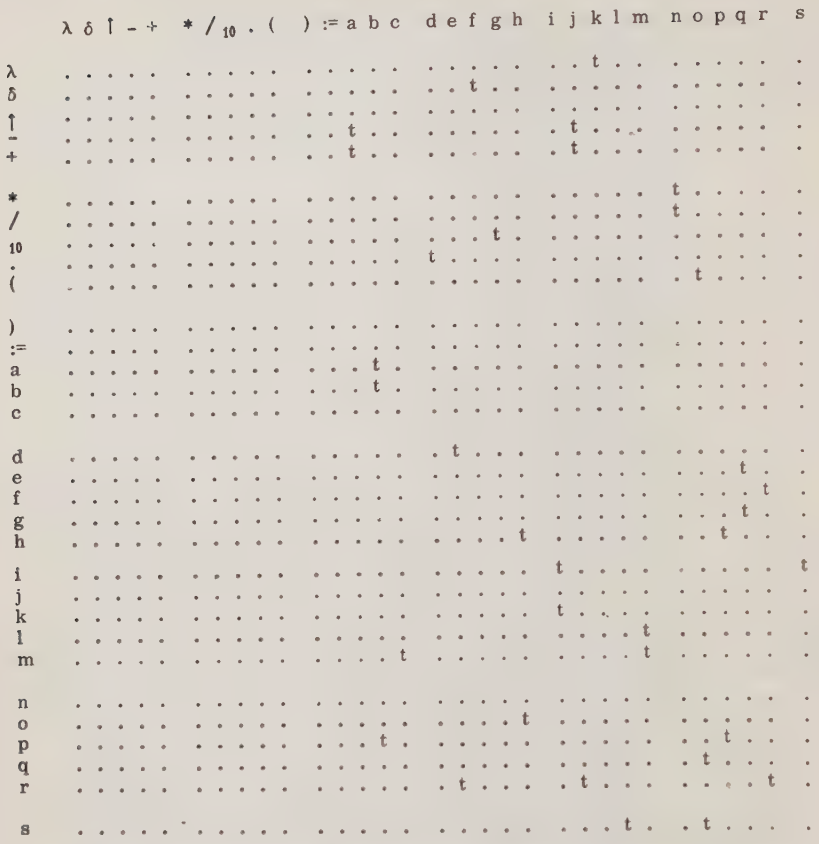


Fig. 2.13. The matrix M_1. Note that in Figures 2.13 through 2.18 we represent **true** by "t" and **false** by "." to make the matrices easier to read.

We shall impose an ordering on R(L) which will result in greater efficiency of use of storage in the processor; we are entitled to do this because we have assumed that no ambiguity can be introduced through the ordering of the rules.

In 2.1 each rule was considered to be made up of a string of i metacomponents followed by a metaresult. There is therefore a rule with a maximal number of metacomponents. We follow each shorter rule with an arbitrary symbol not an element of P(L)

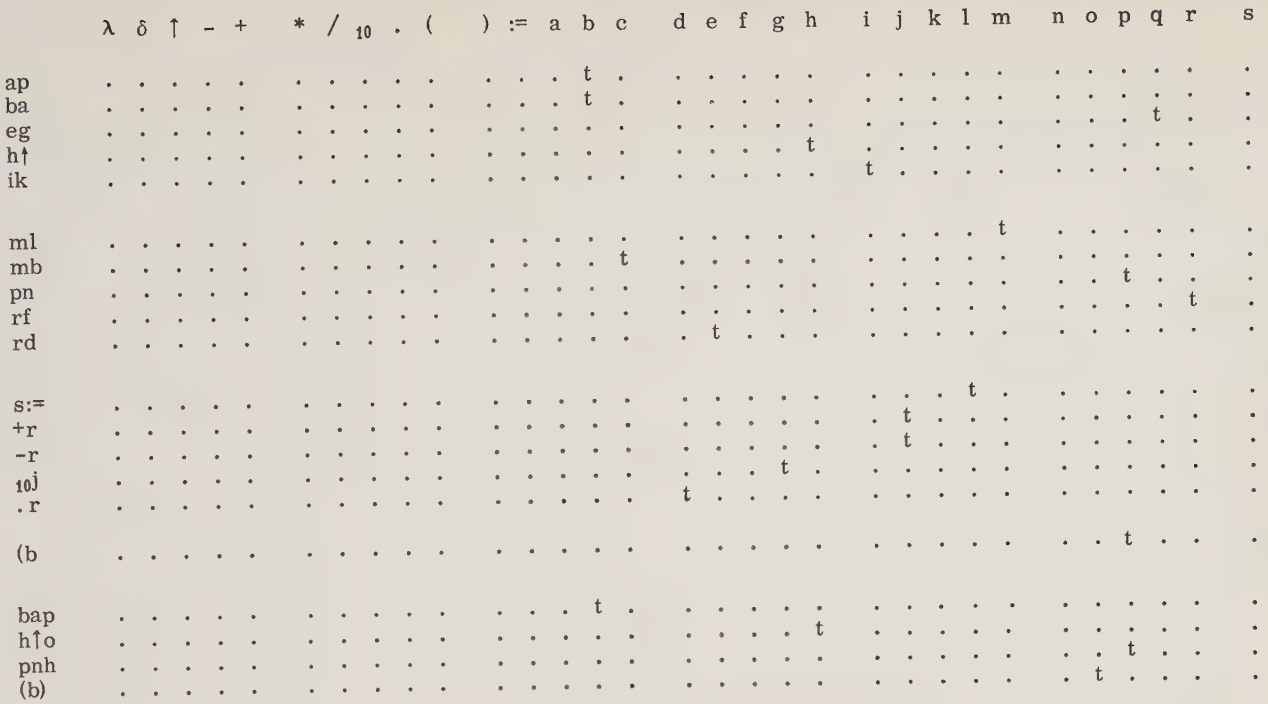


FIG. 2.14. The matrices M_2 and M_3.

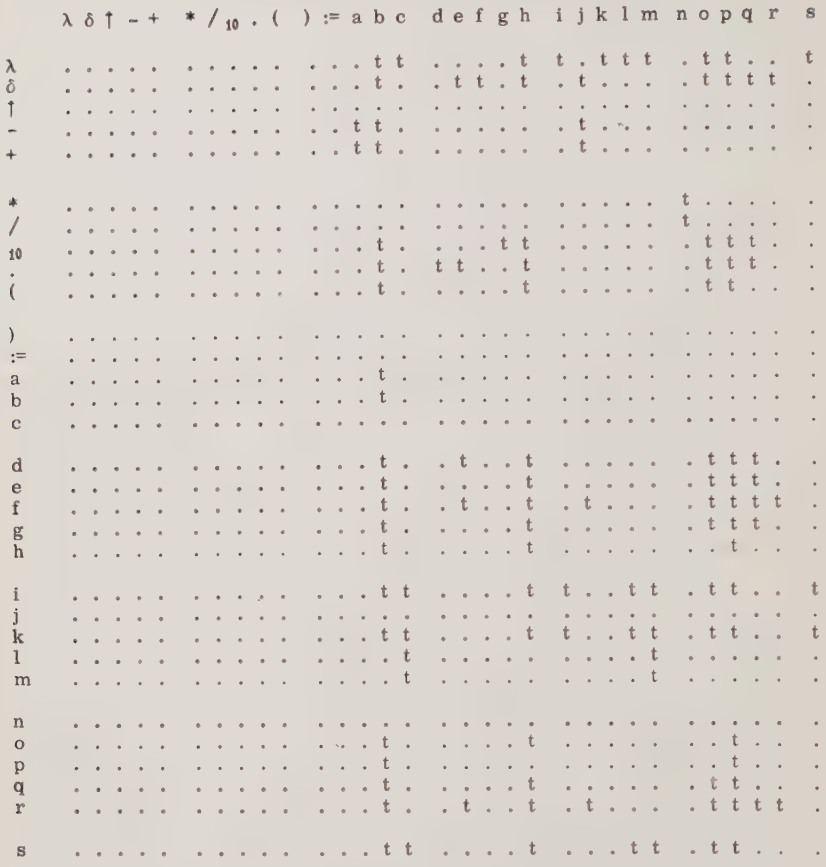


FIG. 2.15. The matrix M_1^*.

and consider R(L) to form an m-by-n array, where m is the number of rules and n is the length of the maximally long rule. (The arbitrary symbol which we shall use will be a space.) TABLE 2.12 can therefore be considered as a 70-by-4 array.

We impose an ordering rule on the rows of this array; if any two rows have the same first k elements, then any rows which fall between the two selected rows must also have the same first k elements. This ordering rule will not, in general, produce a unique ordering; any ordering which it produces is acceptable.

We now assign numbers to the elements of P(L) in a slightly different way than was implied by 2.7. We first number those

	λ	δ	↑	–	+	*	/	10	.	(	)	:=	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
ap	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
ba	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
eg	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	t	.	.	.	.	.	.	t	t	t	.	.
h↑	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	t	.	.	.	.	.	.	.	t	.	.	.
ik	.	.	.	.	.	.	.	.	.	.	.	.	.	t	t	.	.	.	.	.	t	t	.	.	t	t	.	t	t	.	.	t
ml	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.
mb	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
pn	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.
rf	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	t	.	.	t	.	t	.	.	.	.	.	t	t	t	t	.
rd	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	t	.	.	t	.	.	.	.	.	.	.	t	t	t	.	.
s:=	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	.	.	.	t	t	.	.	.	.	.	.
+r	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	.	.	.
–r	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	.	.	.
10j	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	t	t	.	.	.	.	.	.	.	t	t	t	.	.
.r	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	t	t	.	.	t	.	.	.	.	.	.	.	t	t	t	.	.
(b	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	t	.	.	.	.	.	.	t	t	.	.	.
bap	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
h↑o	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	t	.	.	.	.	.	.	.	t	.	.	.
pnh	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.
(b)	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	t	.	.	.	.	.	.	t	t	.	.	.

FIG. 2.16. The Matrices M_2^* and M_3^*.

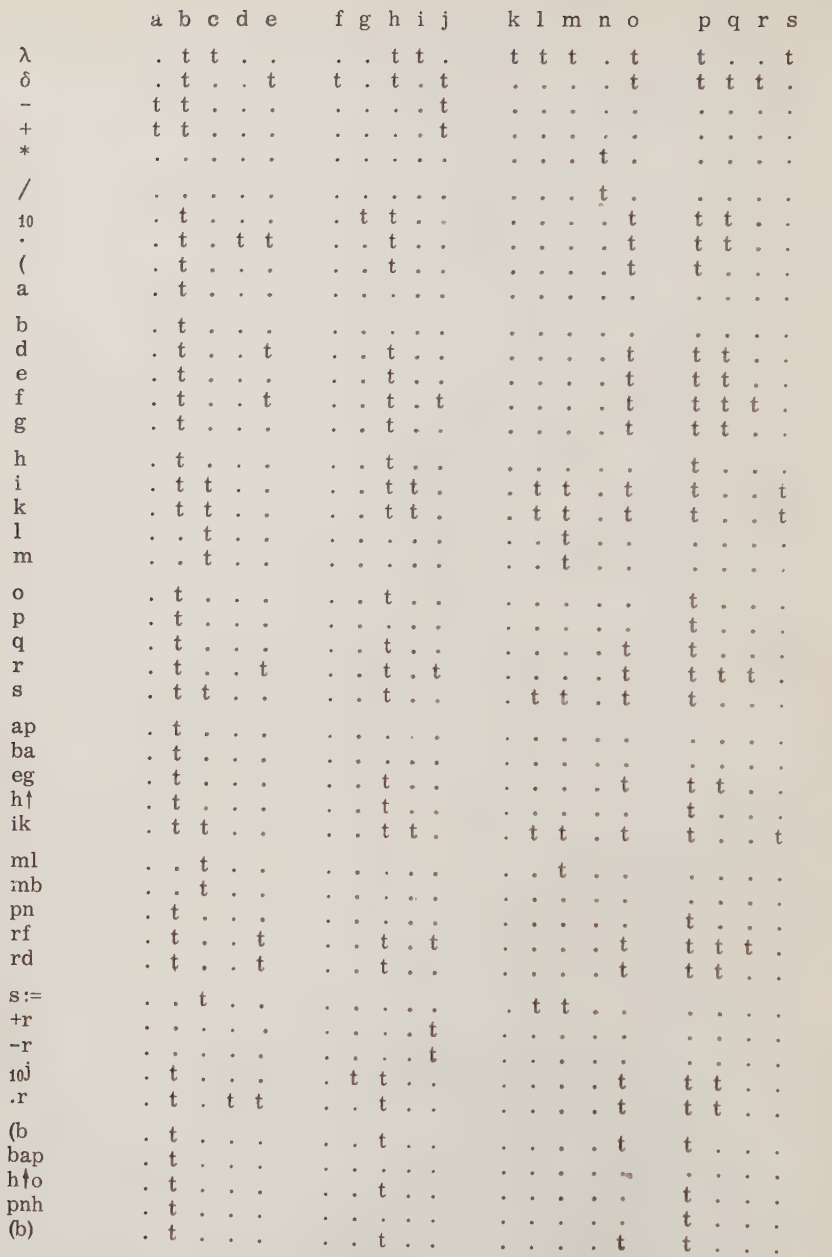


FIG. 2.17. The matrix M*, with false rows and columns removed.

		0 1	0 2	0 3	0 4	0 5	0 6	0 7	0 8	0 9	1 0	1 1	1 2	1 3	1 4	1 5	1 6	1 7	1 8
		a	b	c	d	e	f	g	h	i s	j	k	l	m	n	o	p	q	r
*/	1	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.	.	.
+r -r	2	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.	.	.	.
mb	3	.	.	t	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
l m ml	4	.	.	t	.	.	.	.	.	.	.	.	.	t	.	.	.	.	.
s:=	5	.	.	t	.	.	.	.	.	.	.	.	t	t	.	.	.	.	.
a b ap ba bap	6	.	t	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
p pn pnh	7	.	t	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	.
h o h↑ h↑o	8	.	t	.	.	.	.	.	t	.	.	.	.	.	.	.	t	.	.
(q (b (b)	9	.	t	.	.	.	.	.	t	.	.	.	.	.	.	t	t	.	.
e g eg	10	.	t	.	.	.	.	.	t	.	.	.	.	.	.	t	t	t	.
$10\ 10^j$	11	.	t	.	.	.	.	t	t	.	.	.	.	.	.	t	t	t	.
d rd	12	.	t	.	.	t	.	.	t	.	.	.	.	.	.	t	t	t	.
f r rf	13	.	t	.	.	t	.	.	t	.	t	.	.	.	.	t	t	t	t
δ	14	.	t	.	.	t	t	.	t	.	t	.	.	.	.	t	t	t	t
. . r	15	.	t	.	t	t	.	.	t	.	.	.	.	.	.	t	t	t	.
s	16	.	t	t	.	.	.	.	t	.	.	.	t	t	.	t	t	.	.
i k ik	17	.	t	t	.	.	.	.	t	t	.	.	t	t	.	t	t	.	.
λ	18	.	t	t	.	.	.	.	t	t	.	t	t	t	.	t	t	.	.
+ −	19	t	t	.	.	.	.	.	.	.	t	.	.	.	.	.	.	.	.

FIG: 2.18. The matrix T.

b_i's which appear as first metacomponents; second, the n_j's which appear as first metacomponents; third, the remaining n_k's; and finally, the remaining b_h's. Figure 2.19 gives the numbering (one of several possible) which we shall use.

A	1	1	N	14	14	0	27	27	/	40	41	l	53	58
B	2	2	O	15	15	1	28	28	10	41	42	m	54	59
C	3	3	P	16	16	2	29	29	.	42	43	o	55	61
D	4	4	Q	17	17	3	30	30	(	43	44	p	56	62
E	5	5	R	18	18	4	31	31	a	44	47	q	57	63
F	6	6	S	19	19	5	32	32	b	45	48	r	58	64
G	7	7	T	20	20	6	33	33	d	46	50	s	59	65
H	8	8	U	21	21	7	34	34	e	47	51	c	60	49
I	9	9	V	22	22	8	35	35	f	48	52	j	61	56
J	10	10	W	23	23	9	36	36	g	49	53	n	62	60
K	11	11	X	24	24	-	37	38	h	50	54	↑	63	37
L	12	12	Y	25	25	+	38	39	i	51	55	)	64	45
M	13	13	Z	26	26	*	39	40	k	52	57	:=	65	46

FIG. 2.19. First column is an element of P(L). Second column is the number assigned according to the text. Third column is the subscript assigned according to 2.7.

The syntax matrix S is divided into two parts. The first part (the "head") contains entries for all first metacomponents and all metaresults which do not appear as first metacomponents. (These correspond to the first three categories in the numbering scheme of the preceding paragraph.) Each row in the head of the S matrix represents a different first metacomponent (or metaresult not used as a first metacomponent). The numbering scheme of the preceding paragraph allows us to make the numbers assigned in Figure 2.19 correspond to the row numbers in the head of S; this avoids table-look-up in the processor.

The second part of S consists of packets of rows. Each packet is linked from a row in the first part which represents a first metacomponent, and the packet represents all rules with that same first metacomponent. Since a given first metacomponent can begin several rules, we must have some way of indicating this in the packet. Each row in a packet corresponds to a single element of P(L), as it appears in a rule. A given rule is represented by consecutive rows in the packet, the first representing the second metacomponent and the last the metaresult (the first metacomponent appeared in the head of S). Each row contains as one element the address of another row in the packet which represents the alternative rules which have this particular metacomponent (or metaresult) as the point of difference between the two rules. If there is no such alternative rule, the "alternative choice" is set equal to zero.

TABLE 2.20 gives the descriptions of the six different types of rows. Columns one of S identifies the type of row for the processor. Row types 0, 1, and 2 are in the first part of S; types

TABLE 2.20

	Column in S			
1	2	3	4	5
0	(not used)	row in T	link to packet	(not used)
1	col in T	row in T	link to packet	(see Chapter IV)
2	col in T	0	(not used)	(see Chapter IV)
3	symbol	alternative choice	(not used)	(not used)
4	metacomponent	alternative choice	row in T	(not used)
5	metaresult	alternative choice	row in T	rule number

3, 4, and 5 in the second part. Row types 0 and 3 indicate that the associated metacomponent is a basic symbol; row types 1 and 4, that the associated name is a metacomponent; and row types 2 and 5, that the associated name is a metaresult. In row types 0, 1, and 5, the "row in T" is the one which has associated with it the same element of P(L) as the row in S. In row type 4, the "row in T" is the one which has associated with it the same concatenation of metacomponents as were necessary to get to the row in S. By "column in T" in row types 1 and 2 we mean the column in T which has associated with it the same element of P(L) as the row is S. The "rule number" of row type 5, column 5 is a number associated with rule. It is this number which is put into the output by the processor as part of the prefix form of the rule tree. Hence, if it is desired to have any particular information about the rules that were applied, these numbers can be placed in row type 5 for each rule.

In TABLE 2.20 we have made the least significant digit of each rule equal to the stratification number of the rule, and the most significant digits are the row number from 2.12.1. The stratification number of a rule is equal to the number of metacomponents of the rule which are elements of N(L). We shall justify this assertion when we consider an example of the output of the processor to be discussed in the following chapter.

TABLE 2.21 is the matrix S derived from 2.12. Rows 1 through 62 constitute the head of S; rows 63 through 152 constitute the remainder. We use a minus sign to indicate entries which are not used by the processor. A zero entry for "row T" in row types 0, 1, and 5 indicates a reference to the (fictictous) false row of T, and is treated accordingly by the processor. A zero entry in the third column of row types 3, 4, and 5 indicates that there is no alternative rule available if the "present" rule is not applicable; this was discussed above.

We shall add to the matrix S a row 0 whose elements are

row	1	2	3	4	5
0	1	0	0	–	–

(2.23)

(2.23 is in the same form as the row headings and a typical row of TABLE 2.21.) This row has a dual purpose. First, it provides the information needed by the processor when a request is made for a character from the input string and the input string is exhausted; second, it is used when the grammar does not contain

TABLE 2.21.

row	1	2	3	4	5	row	1	2	3	4	5
1	0	–	18	63	–	39	0	–	1	105	–
2	0	–	18	64	–	40	0	–	1	106	–
3	0	–	18	65	–	41	0	–	11	107	–
4	0	–	18	66	–	42	0	–	15	109	–
5	0	–	18	67	–	43	0	–	9	111	–
6	0	–	18	68	–	44	1	1	6	114	2440
7	0	–	18	69	–	45	1	2	6	116	2450
8	0	–	18	70	–	46	1	4	12	119	2460
9	0	–	18	71	–	47	1	5	10	120	2470
10	0	–	18	72	–	48	1	6	13	123	2480
11	0	–	18	73	–	49	1	7	10	124	2490
12	0	–	18	74	–	50	1	8	8	125	2500
13	0	–	18	75	–	51	1	9	17	129	2510
14	0	–	18	76	–	52	1	11	17	132	2520
15	0	–	18	77	–	53	1	12	4	133	2530
16	0	–	18	78	–	54	1	13	4	134	2540
17	0	–	18	79	–	55	1	15	8	138	2550
18	0	–	18	80	–	56	1	16	7	139	2560
19	0	–	18	81	–	57	1	17	9	143	2570
20	0	–	18	82	–	58	1	18	13	144	2580
21	0	–	18	83	–	59	1	9	16	150	2590
22	0	–	18	84	–	60	2	3	0	–	2600
23	0	–	18	85	–	61	2	10	0	–	2610
24	0	–	18	86	–	62	2	14	0	–	2620
25	0	–	18	87	–	63	5	52	0	17	10
26	0	–	18	88	–	64	5	52	0	17	20
27	0	–	14	89	–	65	5	52	0	17	30
28	0	–	14	90	–	66	5	52	0	17	40
29	0	–	14	91	–	67	5	52	0	17	50
30	0	–	14	92	–	68	5	52	0	17	60
31	0	–	14	93	–	69	5	52	0	17	70
32	0	–	14	94	–	70	5	52	0	17	80
33	0	–	14	95	–	71	5	52	0	17	90
34	0	–	14	96	–	72	5	52	0	17	100
35	0	–	14	97	–	73	5	52	0	17	110
36	0	–	14	98	–	74	5	52	0	17	120
37	0	–	19	99	–	75	5	52	0	17	130
38	0	–	19	102	–	76	5	52	0	17	140

n_1. These will both be discussed in the next section, which deals with the processor.

Table 2.21 (cont'd)

row	1	2	3	4	5	row	1	2	3	4	5
77	5	52	0	17	150	115	5	45	0	6	462
78	5	52	0	17	160	116	4	44	0	6	–
79	5	52	0	17	170	117	4	56	0	6	–
80	5	52	0	17	180	118	5	45	0	6	473
81	5	52	0	17	190	119	5	47	0	10	481
82	5	52	0	17	200	120	4	49	122	10	–
83	5	52	0	17	210	121	5	57	0	9	492
84	5	52	0	17	220	122	5	57	0	9	501
85	5	52	0	17	230	123	5	58	0	13	511
86	5	52	0	17	240	124	5	57	0	9	521
87	5	52	0	17	250	125	3	63	128	–	–
88	5	52	0	17	260	126	4	55	0	8	–
89	5	48	0	13	270	127	5	50	0	8	532
90	5	48	0	13	280	128	5	56	0	7	541
91	5	48	0	13	290	129	4	52	131	17	–
92	5	48	0	13	300	130	5	51	0	17	552
93	5	48	0	13	310	131	5	59	0	16	561
94	5	48	0	13	320	132	5	51	0	17	571
95	5	48	0	13	330	133	5	54	0	4	581
96	5	48	0	13	340	134	4	53	136	4	–
97	5	48	0	13	350	135	5	54	0	4	592
98	5	48	0	13	360	136	4	45	0	3	–
99	4	58	101	2	–	137	5	60	0	0	602
100	5	61	0	0	371	138	5	50	0	8	611
101	5	44	0	6	380	139	4	62	142	7	–
102	4	58	104	2	–	140	4	50	0	7	–
103	5	61	0	0	391	141	5	56	0	7	623
104	5	44	0	6	400	142	5	45	0	6	631
105	5	62	0	0	410	143	5	55	0	8	641
106	5	62	0	0	420	144	4	48	146	13	–
107	4	61	0	111	–	145	5	58	0	13	652
108	5	49	0	10	431	146	4	46	148	12	–
109	4	58	0	15	–	147	5	47	0	10	662
110	5	46	0	12	441	148	5	47	149	10	671
111	4	45	0	9	–	149	5	61	0	0	681
112	3	64	0	–	–	150	3	65	152	–	–
113	5	55	0	8	451	151	5	53	0	4	691
114	4	56	0	6	–	152	5	55	0	8	701

Some Further Comments on S

The matrix S is a representation of R(L) in the form of a LISP-type list (14). This configuration is merely convenient when no

changes are anticipated in the grammar; when it is anticipated that the grammar may be extended or changed (as is the case when sub-languages may have their grammar checked out before being incorporated into the main grammar of a language) this LISP-type configuration means that only the incrementally new information need be added to S. In addition, the ordering rule given for the rows of R(L) guarantees that S will have the minimum number of rows, since there is no unnecessary duplication.

III. THE BNF PROCESSOR

Introduction

Fig. 3.1 gives in Algol a procedure called RAVEL which will examine a string of characters to determine their grammaticalness. RAVEL is called by a procedure statement of the form:

RAVEL (AP1, AP2, AP3, AP4, AP5, AP6, AP7, AP8) (3.2)

where: AP1 = the number (in the sense of 2.19) assigned to the single head of the language,
AP2 = the number assigned to the name of the null string (or zero if the grammar does not contain n_1),
AP3 = the label of the statement to which transfer is to be made if the string is ungrammatical,
AP4 = the label to which transfer is to be made if the string is grammatical,
AP5 = the identifier of the array whose elements are the numerical equivalents of characters in the input string (in the sense of 2.19),
AP6 = the array identifier of the Syntax Matrix,
AP7 = the array identifier of the Transition Matrix, and
AP8 = the procedure identifier of the output procedure.

The array AP5 is a vector whose subscript is the position of the character in the input string. The character positions are numbered in the direction of scan, beginning with zero. The value of AP5 for a subscript not referring to a character in the input string is defined to be zero. There is nothing in RAVEL which prevents AP5 from being considered the procedure identifier of a function designator which serves as an input routine; it may therefore be convenient to replace A[h] by A(h) in any particular implementation.

```
procedure RAVEL (name, empty, BAD, GOOD, A, S, T, Write);
value name, empty;
integer name, empty;
Boolean array T;
integer array A, S;
label BAD, GOOD;
procedure Write;
   begin comment block 1;
   integer W0, W1, W2, W3, W4, h, i, z, y, L, q, SJ;
   Boolean AL;
   Boolean procedure Nogo (P, Q);
    value P, Q; integer P, Q; Nogo: = P = 0 ∨ ¬T [P, Q];
   L1:  W1: = S[empty, 2];
        W2: = S[name, 2];
        W3: = S[A[0], 1];
        W4: = S[A[0], 3];
        AL: = W3 ≠ 0 ∨ Nogo (W4, W2) ∧ Nogo (W1, W2);
        if AL then go to BAD;
        W3: = name;
        h: = q: = W1: = 0;
        i: = z: = −1;
        y: = 1;
        L: = 4;
   L2:  W4: = A[h];
        if S[W4, 1] ≠ 0 then begin SJ: = 2; go to L4 end;
        W0: = S[W3, 2];
        AL: = true;
   L3:  W2: = S[W4, 3];
        if Nogo (W2, W0) then begin if AL then go to L31 else SJ: = 3 end
           else begin SJ: = 4; z: = z + 1
   L4:  begin comment block 2;
        own Boolean array B[0: z];
        own integer array G, F, K, W[0: z];
           if SJ ≠ 4 then go to L7;
   L5:  B[z]: = AL;
        if AL then h: = h + 1;
        G[z]: = W0;
        F[z]: = W3;
        K[z]: = y;
        W[z]: = h;
        W4: = S[W4, 4];
   L6:  i: = i + 1;
        SJ: = 1;
   L7:  begin comment block 3;
        switch OUT: = BAD, C2, C3, C4, C5;
        Boolean SQ;
        switch WHERE: = L8, L17, L30;
        own integer array R, E, J[0:i];
              go to WHERE [SJ];
        L8:   R[i]: = 0;
              E[i]: = L;
              J[i]: = W4;
```

Fig. 3.1

```
L9:   W1: = S[J[i], 3];
      W2: = S[J[i], 1];
      W3: = S[J[i], 2];
      if W2 ≠ 5 then go to L12;
      if Nogo (S[J[i], 4], G[z]) then go to L23;
      W4: = S[W3, 4];
      L: = 6;
      go to L6;
L12:  if W2 ≠ 4 then go to L14;
      L: = 5;
      go to if Nogo (S[J[i], 4], G[z]) then L17 else L2;
L14:  if W3 ≠ A[h] then go to L17;
      h: = h + 1;
L16:  J[i]: = J[i] + 1;
      go to L9;
L17:  if W1 = 0 then go to L19;
      J[i]: = W1;
      go to L9;
L19:  E[i]: = E[i] −3;
L20:  i: = i −1;
      go to OUT [E[i + 1]];
C2:   W3: = F[z];
      W0: = G[z];
      h: = W[z];
      y: = K[z];
      z: = z − 1;
      go to if B[z + 1] then L31 else L30;
C4:   Write (y, W1);
      go to GOOD;
C5:   z: = z −1;
      R[i]: = R[i] + 1;
      SQ: = false;
L21;  begin comment block 4;
      own integer array N[0: q];
           if SQ then go to L22;
           N[q]: = W1;
           q: = q + 1;
           go to L16;
      L22:  Write (y, N[q]);
            y: = y + 1;
            go to L26
      end block 4;
C3:   W3: = S[J[i], 2];
      W1: = S[J[i], 3];
L23:  if W3 ≠ F[z] then go to L17;
      W1: = −y;
L25:  Write (y, S[J[i], 5]);
      y: = y + 1;
L26:  if R[i] = 0 then go to L29;
      if E[i] = 6 then go to L20;
      i: = i − 1;
      go to L25;
```

```
        L29: R[i]: = R[i] −1;
             q: = q −1;
             SQ: = true;
             go to L21;
        L30: W1: = S[J[i], 3];
             go to L17
        end block 3
    end block 2;
L31: W4: = empty;
    AL: = false;
    go to L3
end block 1
```

Certification

A version of the algorithm given in this paper as Figure 3.1 was compiled and run by E. T. Irons. The machine time on the Control Data Corporation 1604 was provided by the Institute for Defense Analyses at Princeton, New Jersey, incidental to checking out their Algol compiler.

FIG. 3.1

We consider the output format to be a series of cells numbered from top to bottom, starting with one. The first parameter of AP8 is the number of the cell in which the value of the second parameter is to be placed. (We start the numbering of the output cells at one rather than zero to avoid having to distinguish between plus zero and minus zero in the output; this will become clearer below, when the format of the output is discussed.)

The Block Structure of RAVEL

Figure 3.3 shows the block structure of RAVEL, together with the pertinent declarations and statements. Block 1 is the body of the procedure. Inside this block the variables z, i, and q are declared, and are "associated" respectively with blocks 2, 3, and 4. These latter blocks have in common three things:

1) Each block contains a dynamic own array declaration (10, 19),
2) all arrays declared in a given block depend on the variable associated with that block, and
3) the variable associated with a block is incremented (decremented) outside the block and decremented (incremented) inside the block.

Such an arrangement forces the blocks structure to produce a dynamically allocated push-down list, but does not, in general, allow the block structure to be used to delineate the logical

```
begin comment block 1;
integer i, q, z, SJ;
L1: q: = 0; i: = z; = −1;
L2: SJ: = 2;
L3: SJ: = 3; SJ: = 4; z: = z + 1;
L4: begin comment block 2;
.   own Boolean array B[0: z];
.   own integer array G, F, K, W[0: z];
.   .   if SJ ≠ 4 then go to L7
.   L6: i: = i + 1; SJ: = 1;
.   L7: begin comment block 3;
.   .   switch WHERE: = L8, L17, L30;
.   .   own integer array R, E, J[0: i];
.   .   .   go to WHERE [SJ];
.   .   L8:
.   .   L17:
.   .   L20: i: = i −1;
.   .   C2:  z: = z −1;
.   .   C5:  z: = z −1;
.   .   L21: begin comment block 4;
.   .   .   own integer array N[0:q];
.   .   .   .   q: = q + 1;
.   .   .   end block 4;
.   .   L26: i: = i −1;
.   .   L29: q: = q −1;
.   .   L30:
.   .   end block 3;
.   end block 2;
end block 1;
```

FIG. 3.3. The Block Structure of RAVEL.

structure of the program. In RAVEL, the failure of the block structure to follow the logical division of the program necessitates the introduction of the switch declaration "WHERE" in block 3; this declaration is called by the go to statement at the entrance of the block because the block structure cuts across three logical paths in the program in order to achieve the dynamic storage allocation.

Such a scheme is not in general efficient in the object program. The alternative is to declare some arbitrary number for the upper bounds of the arrays and to hope that the declared capacity of the arrays will not be exceeded. This will result in more efficient running programs if enough is known about the strings to be inspected by RAVEL so that reasonable boundary conditions can be determined.

A Segment-by-Segment Discussion of RAVEL

In this section we shall discuss the function of RAVEL segment by segment. By Segment we here mean a sequence of statements prefixed by a label. We shall refer to a segment by the label which precedes it; labels which are prefixed to blocks (L4, L7, and L21) will, however, refer to the first statements of the block (which would be otherwise unlabelled).

We shall discuss the segments in the order in which they appear in RAVEL, rather than attempt to indicate a specific operational order. By the nature of Algol the general flow of control is from first statement to last, and RAVEL was written with this in mind. However, as discussed in the section on the block structure of RAVEL, the logical flow of the program has been distorted to allow the block structure to perform dynamic storage allocation.

Declarations in Block 1: The variables W0, W1, W2, W3, and W4 are temporary storage locations. Variable h is used to keep track of the first unprocessed character in the input string, and is used only as a subscript for the array specified by the fifth actual parameter of 3.2. Variable z is used as an index on the pushdown lists which store the characteristics of the goal currently sought by the processor; this will be discussed more fully with the declarations for block 2. Variable i has a similar relationship to block 3, where it is used as an index on the rule currently recognized by the processor. Again, q is used in block 4 as an index on back-references in the output.

Variable y is used as an index for the output routine, as mentioned in the discussion of 3.2. Variable L is a temporary storage location used to hold the type of iteration that is about to be made by the processor. Variable SJ is used solely to control the logical path across the block structure. Variable AL is used initially as a test for the potential grammaticalness of the input string, and thereafter is used to indicate whether a character from the input string or the null string is being processed.

The Boolean function designator "Nogo (P, Q)" is **true** whenever either the first parameter is equal to zero or T[P, Q] (the element of the transition matrix is **false**; otherwise "Nogo" is **false** (Informally, "Nogo" is **true** if there is no path from the metacomponent in hand to the desired metaresult; see the section on the use of T.)

Segment L1 is used purely for initial setting, and is never

reentered. The first decision made in L1 determines whether the input string is potentially grammatical; there must be a path from the first character of the input string to the head of the language, or from the null string to the head of the language, and in either case, the first character of the input string must be an element of B(L). If the input string is per se ungrammatical, as determined from these conditions, RAVEL exits immediately to BAD.

Segment L2 determines whether the current character of the input string (as determined by h) may occur as a first metacomponent; if not, control is transferred to L4, and if so, to L3. If the current character of the input string may occur as a first metacomponent AL is set to true to indicate that a non-null character is being processed.

Segment L3 determines whether the current character in the input string may indeed lead to the current goal; this is done by invoking "Nogo". If the character may lead, control is (effectively) transferred to L8 for processing to determine whether in this particular case it does lead. If the character cannot lead but is a character from the input string, control is transferred to L31 to try the null string instead. If the null string is being tried (indicated by the falsity of AL) an error is indicated and control is (effectively) transferred to L30.

Note that the transfers of control are not direct, but take place by setting SJ to an appropriate value; the reason for this was discussed in the section on the block structure of RAVEL.

Declarations in Block 2: The five arrays declared in this block are used to store characteristics of the goal currently being sought by the processor. Array B indicates whether the goal is being reached from a character in the input string (**true**) or from the null string (**false**). Array G stores the column number of the column of T associated with the goal. Array F stores the goal itself (actually the number assigned to the goal, in the sense of 2.19).

Array K stores the value that y had at the beginning of the search for the current goal. Array W, similarly, stores the value of h at the beginning of the current search. These two arrays are used if a goal is not found to reset the output and input routines.

Segment L4, consisting of a single **go to** statement, is used to transfer control to segment L7 when a new goal is not required.

y	rule no.	rule
1	541	<factor> <term>
2	611	<primary> <factor>
3	701	<variable> <primary>
4	561	<identifier> <variable>
5	571	<letter> <identifier>
6	30	C <letter>
7	462	<adding operator> <term> <arithmetic expression>
8	−1	
9	400	+ <adding operator>
10	701	<variable> <primary>
11	561	<identifier> <variable>
12	571	<letter> <identifier>
13	40	D <letter>
14	631	<term> <arithmetic expression>
15	541	<factor> <term>
16	532	<factor>↑<primary> <factor>
17	−10	
18	611	<primary> <factor>
19	451	(<arithmetic expression>) <primary>
20	−7	
21	602	<left part list> <arithmetic expression> <assignment statement>
22	−14	
23	581	<left part> <left part list>
24	691	<variable>: = <left part>
25	561	<identifier> <variable>
26	571	<letter> <identifier>
27	10	A <letter>
28	−21	

FIG. 3.5. The output of RAVEL as generated.

This decision was made in segment L2 or L4, when SJ was set unequal to 4.

Segment L5 is entered only when a new goal must be searched for; it stores appropriate parameters in the newly-created locations in the five vectors declared in block 2, and advances the input index if the current character came from the input string.

Segment L6 creates new locations in the arrays declared in block 3 and sets SJ so that the first metacomponent of the rule in hand will serve as the starting point for a search.

Declarations in Block 3. Variable SQ is used to determine whether block 4 should store a new output back-reference for later use (**false**) or should write out a back-reference (**true**).

y	rule no.	rule	
21	602	<left part list><arithmetic expression>	<assignment statement>
14	631	<term>	<arithmetic expression>
15	541	<factor>	<term>
16	532	<factor> <primary>	<factor>
10	701	<variable>	<primary>
11	561	<identifier>	<variable>
12	571	<letter>	<identifier>
13	40	D	<letter>
18	611	<primary>	<factor>
19	451	(<arithmetic expression>)	<primary>
7	462	<adding operator><term>	<arithmetic expression>
1	541	<factor>	<term>
2	611	<primary>	<factor>
3	701	<variable>	<primary>
4	561	<identifier>	<variable>
5	571	<letter>	<identifier>
6	30	C	<letter>
9	400	+	<adding operator>
23	581	<left part>	<left part list>
24	691	<variable>:=	<left part>
25	561	<identifier>	<variable>
26	571	<letter>	<identifier>
27	10	A	<letter>

FIG. 3.6. The output of RAVEL in prefix tree form.

Switch OUT selects a path on the basis of the type of search currently being performed by RAVEL. BAD was defined with 3.2; C2 indicates that a metacomponent other than the first metacomponent of a rule could not be located; C3 indicates that a metaresult which was located turned out, after further investigation, not to be desired; C4 is reached when the input string is grammatical; and C5 is reached when a metacomponent other than the first has been successfully identified. Switch WHERE was discussed with the block structure of RAVEL.

The three arrays R, E, and J are used to store the characteristics of the rule in hand. Array R indicates the stratification number of the rule, and is used to count the number of backreferences which must be written out if this rule is used. Array E indicates the type of exit to be made, via switch OUT, after a rule has been inspected; note that E[i] = 6 behaves like a special exit but does not appear in switch OUT. Array J stores the row

number in S of the metacomponent or metaresult of the rule currently being investigated by the processor.

Segment L7 (as does segment L4) consists of a single go to statement which selects one of the three exits of switch WHERE on the basis of SJ. This was discussed with the block structure of RAVEL.

Segment L8 sets the values of the newly-created cells in vectors R, E, and J to begin searching at the second metacomponent of the selected rule. (The first metacomponent is either an element of B(L), in which case it was identified when it was picked up from the input, or is an element of N(L), in which case it must have been identified in order to have selected this rule to begin with.)

Segment L9 does some minor bookkeeping to avoid subscripted subscripts at later stages of processing. It then transfers control to L12 if the current element of the current rule is not a metaresult. If the current element is a metaresult, but the metaresult cannot lead to the current goal, control is transferred to L23 to decide whether the potential error is an actual one. Otherwise, control is transferred to L6. This last transfer (to L6) indicates that another branch has been tentatively placed in the rule-tree, and a new iteration must be performed to see if this new branch is indeed correct.

Segment L12 is reached from segment L9 when a rule has more than one metacomponent. If the metacomponent being investigated is an element of B(L), control is transferred to L14. If the metacomponent cannot lead to the current goal, control is transferred to L17. Otherwise, control is transferred to L2, where the metacomponent will become the new goal.

Segment L14 is reached when a metacomponent other than the first in a rule is an element of B(L) rather than N(L). The distinction is made because checking for elements of B(L) can be performed without involved iterations; either the next character in the input string is or is not the element of B(L) that is needed. If a match is not found, control is transferred to L17; otherwise, the input index is incremented and control passes to L16.

Segment L16 is reached when the k^{th} element of a rule has been found and it is desired to try to find the $(k+1)^{st}$ element. This effect is achieved by adding a one to the current contents of J[i]; from the discussion of the construction of the matrix S, it will be remembered that all elements of a rule except the first occupy consecutive rows in a packet of rows.

Segment L17 is the general error-checking segment. In segment L9, temporary storage location W1 was set to indicate the alternative path through S (if any) that would be available if the current path failed. If W1 = 0, there is no alternative path, and control is transferred to L19. If there is an alternative path, it is indicated by a (non-zero) row number in W1 which replaces the contents of J[i]. Control is then transferred to L9 to search for the alternative rule.

Segment L19 is reached when there is an error at the current level and there is no way to repair the error (at this level). The values of switch OUT, including the fictitious sixth value, have been arranged so that the error exit is determined by subtracting three from the value of the subscript for the good exit, and the good exit is initially assumed.

Segment L20 reduces the index for the vectors declared in block 3; when block 3 is reentered the values in the locations in the arrays whose subscripts are greater than the value of i will become undefined. Exit is made to switch OUT as conditioned by the exit indication of the previous E[i].

Segment C2, as was mentioned in the description of the block 3 declarations, is reached when a metacomponent other than the first metacomponent of a rule could not be identified. It resets W3, W0, z, h, and y to their respective values at the beginning of the search for the metacomponent that was not identified; if the search which failed started from a character in the input string, control is transferred to L31; otherwise, control is transferred to L30.

Segment C4 writes out the last back-reference and exits to GOOD,

Segment C5 is research when a metacomponent which is an element of N(L) and not the first metacomponent has been identified. Since this metacomponent is the current goal, z is diminished by one; since the metacomponent is an element of N(L), R[i] is increased by one to indicate that an additional back-reference must be written out when output for this rule is generated.

Declarations in Block 4. The only declaration in block 4 is for the array N. This array is also a dynamically allocated push-down list, used to store back-references that have not been written out. Its controlling variable (as mentioned in the description of the declarations in block 1) is q.

Segment L21 first checks to see whether a back-reference

should be stored or written, as determined by the falsity or truth (respectively) of SQ. If a back-reference is to be stored, the reference is moved from W1 to the currently available cell in the array, and the index is increased. Control is then transferred to L16.

Segment L22 writes out a back-reference, increments the output index, and transfers control to L26.

Segment C3 is reached when a metaresult which had been identified turns out not to lead to the desired goal. In this case it is still possible that no error has been made, since the metaresult may itself be the desired goal without being able to lead further. This segment simply sets up the necessary bookkeeping and passes control to L23.

Segment L23 is used to check whether a metaresult which does not lead to the current goal is equal to the current goal. If not, control is transferred to L17, which is the general error segment. If the metaresult which has been identified is equal to the current goal, the current back-reference is stored in W1 and control enters L25. A back-reference consists of the negative of the current cell in the output string, and is generated by placing (−y) into W1.

Segment L25 is the beginning of the output loop. It writes the rule number that was assigned to the rule that has been idenfified (which number was placed in column 5 of row type 5; see chapter 2), increments the output index, and passes to L26.

Segment L26 first checks to see whether back-references must be written; if yes, control is transferred to L29. Next, a check is made to see that output will be generated only for completely identified rules; if a rule is completely identified, the entries in R[i], E[i], and J[i] will all refer to the metaresult of the rule, and in particular, E[i] will be equal to 6. Hence, as long as E[i] is equal to 6 the index i is reduced by one and control is transferred back to L25 to write out another rule number.

Segment L29 clocks the writing of back-references. It decrements R[i] and q, sets SQ to true (which indicates that block 4 should write a back-reference rather than store one), and transfers control to L21, which is the entrance to block 4.

Segment L30 is used purely for bookkeeping when an error is discovered in segment C2; it merely restores the original value of W1 for the benefit of segment L17.

Segment L31 is the segment which sets the processor to consider the null string rather than a character from the input string.

An Example of the Output of RAVEL

Figure 3.5 shows the output from RAVEL which results from supplying as an input string

$$A := (+C) \uparrow D \tag{3.4}$$

and using figures 2.21 and 2.18 as matrices S and T, respectively. The first column of 3.5 is the index y, the second column the rule number, and the third column the rule itself, written out from Figures 2.12.1 and 2.12.2.

The negative numbers in 3.5 are back-references. If the sign is considered as an indicator bit rather than a sign, the back-references indicate the starting point of the sub-string of the output which should be inserted at the point of the back-reference. The termination of the sub-string is at the point where the tail deficiency is zero [Gorn (6, 7)]. It must be remembered that the least significant digit of the rule number was made equal to the stratification number of the rule, as described in the preceding chapter.

The prefix form used in this paper is of the type that convert a+b to +ba rather than +ab. This has the advantage of requring less pushdown storage for the conversion, and allows output to be written as soon as a piece of the input string has been identified. In the event that the output was premature (as determined by further inspection of the input string) the output routine is reset from the value stored in the array K, and the erroneous output is overwritten.

In Figure 3.6 we show the output of RAVEL rearranged to eliminate the need for back-references (by making the substitutions called for by the back-references). It is particularly clear in this configuration that the rules with stratification number 1 are used solely as a renaming convenience and have, themselves, no necessary semantic content. By changing segment L25 of RAVEL to:

```
L25:  W3:=S[J[i], 5];                                        (3.7)
      if W3-10× entier(W3/10) ≠ 1 then
         begin Write (y, W3); y:=y + 1 end;
```

we would avoid writing such rule numbers in the output, and the result would have been that illustrated in Figure 3.8. While this may be desirable if the output of RAVEL is to be used in some further processor (such as one which translates the rule numbers to machine coding), it results in the output of RAVEL not

y	rule no.
1	30
2	462
3	−1
4	400
5	40
6	532
7	−5
8	−2
9	602
10	−6
11	10
12	−9

FIG. 3.8.1.

y	rule no.
9	602
6	532
5	40
2	462
1	30
4	400
11	10

FIG. 3.8.2.

reflecting exactly the rule-tree that was used by the writer of the input string (always assuming that the input string was grammatical).

IV. SOME EXTENSIONS TO RAVEL

Introduction

In the preceding three chapters we have assumed that the language being processed had a Chomsky PSG2 (or, trivially, a PSG3), and that the input consisted of a string of basic symbols which were scanned from left to right. In this chapter we discuss in detail the changes to RAVEL necessary to allow a right-to-left scan and to allow elements of N(L) to appear in the input (e.g. [1]). We shall then consider, in a general way, the introduction of metalinguistic operators which will enable RAVEL to process languages whose grammars are less restricted than PSG2.

Right-to-Left Scan

Ravel as presented in 3.1 is independent of scan direction, so long as one scan direction is maintained. The dependence on scan direction occurs in the construction of the Syntax and Transition matrices and the numbering of the characters in the input string.

TABLES 4.1 and 4.2 are the S and T matrices (respectively) for a right-to-left scan, based on the rules given in 2.12. The technique for the construction of these "reversed" matrices was

TABLE 4.1.1.

row	1	2	3	4	5	row	1	2	3	4	5
1	0	–	21	66	–	41	–	–	–	–	–
2	0	–	21	67	–	42	–	–	–	–	–
3	0	–	21	68	–	43	–	–	–	–	–
4	0	–	21	69	–	44	2	1	0	–	1440
5	0	–	21	70	–	45	1	2	1	106	1450
6	0	–	21	71	–	46	1	4	11	108	1460
7	0	–	21	72	–	47	1	5	10	111	1470
8	0	–	21	73	–	48	1	6	19	112	1480
9	0	–	21	74	–	49	1	7	10	115	1490
10	0	–	21	75	–	50	1	9	5	118	1500
11	0	–	21	76	–	51	1	9	8	122	1510
12	0	–	21	77	–	52	1	11	9	125	1520
13	0	–	21	78	–	53	1	12	2	128	1530
14	0	–	21	79	–	54	2	13	0	–	1540
15	0	–	21	80	–	55	1	15	6	131	1550
16	0	–	21	81	–	56	1	16	4	135	1560
17	0	–	21	82	–	57	1	17	7	140	1570
18	0	–	21	83	–	58	1	18	18	141	1580
19	0	–	21	84	–	59	1	19	7	149	1590
20	0	–	21	85	–	60	2	3	0	–	1600
21	0	–	21	86	–	61	1	10	14	123	1610
22	0	–	21	87	–	62	2	14	0	–	1620
23	0	–	21	88	–	63	–	–	–	–	–
24	0	–	21	89	–	64	0	–	7	150	–
25	0	–	21	90	–	65	0	–	3	153	–
26	0	–	21	91	–	66	5	52	0	9	710
27	0	–	20	92	–	67	5	52	0	9	720
28	0	–	20	93	–	68	5	52	0	9	730
29	0	–	20	94	–	69	5	52	0	9	740
30	0	–	20	95	–	70	5	52	0	9	750
31	0	–	20	96	–	71	5	52	0	9	760
32	0	–	20	97	–	72	5	52	0	9	770
33	0	–	20	98	–	73	5	52	0	9	780
34	0	–	20	99	–	74	5	52	0	9	790
35	0	–	20	100	–	75	5	52	0	9	800
36	0	–	20	101	–						
37	0	–	16	102	–						
38	0	–	16	103	–						
39	0	–	15	104	–						
40	0	–	15	105	–						

mentioned in Chapter II. It should be noted that 4.1 was constructed using the numbering scheme of 2.19 rather than with

TABLE 4.1.2

row	1	2	3	4	5	row	1	2	3	4	5
76	5	52	0	9	810	116	5	57	0	7	1172
77	5	52	0	9	820	117	5	57	0	7	1181
78	5	52	0	9	830	118	4	62	121	5	–
79	5	52	0	9	840	119	4	56	0	5	–
80	5	52	0	9	850	120	5	56	0	4	1193
81	5	52	0	9	860	121	5	56	0	4	1201
82	5	52	0	9	870	122	5	59	0	7	1211
83	5	52	0	9	880	123	3	41	0	–	–
84	5	52	0	9	890	124	5	49	0	14	1221
85	5	52	0	9	900	125	4	51	127	9	–
86	5	52	0	9	910	126	5	51	0	8	1232
87	5	52	0	9	920	127	5	51	0	8	1241
88	5	52	0	9	930	128	4	54	130	2	–
89	5	52	0	9	940	129	5	54	0	0	1252
90	5	52	0	9	950	130	5	54	0	0	1261
91	5	52	0	9	960	131	3	63	134	–	–
92	5	48	0	19	970	132	4	50	0	6	–
93	5	48	0	19	980	133	5	50	0	5	1272
94	5	48	0	19	990	134	5	50	0	5	1281
95	5	48	0	19	1000	135	4	44	139	4	–
96	5	48	0	19	1010	136	4	45	138	4	–
97	5	48	0	19	1020	137	5	56	0	4	1293
98	5	48	0	19	1030	138	5	56	0	4	1302
99	5	48	0	19	1040	139	5	56	0	4	1311
100	5	48	0	19	1050	140	5	55	0	6	1321
101	5	48	0	19	1060	141	3	42	143	–	–
102	5	44	0	0	1070	142	5	46	0	13	1331
103	5	44	0	0	1080	143	3	38	145	–	–
104	5	62	0	0	1090	144	5	61	0	17	1341
105	5	62	0	0	1100	145	3	37	147	–	–
106	4	54	0	1	–	146	5	61	0	17	1351
107	5	60	0	0	1112	147	5	47	148	10	1361
108	4	58	110	12	–	148	5	61	0	14	1371
109	5	47	0	10	1122	149	5	55	0	6	1381
110	5	47	0	10	1131	150	4	45	0	7	–
111	5	57	0	7	1141	151	3	43	0	–	–
112	4	58	114	19	–	152	5	55	0	6	1391
113	5	58	0	18	1152	153	4	59	0	3	–
114	5	58	0	18	1161	154	5	53	0	2	1401
115	4	47	117	10	–						

TABLE 4.2
The T matrix for a right-to-left scan

	0 0 0 0 0 1 2 3 4 5	0 0 0 0 1 6 7 8 9 0	1 1 1 1 1 1 2 3 4 5	1 1 1 1 6 7 8 9
	a b c d e	f g h i j	k l m n o	p q r s
b bm	1 . . t . .	. . . . .	. . . . .	. . . .
1 lm	2	. . . . .	. . t . .	. . . .
:= :=s	3	. . . . .	. t t . .	. . . .
p pa pab	4 . t t . .	. . . . .	. . . . .	. . . .
h hn hnp	5 . t t . .	. . . . .	. . . . .	t . . .
o o↑ o↑h	6 . t t . .	. . t . .	. . . . .	t . . .
s)b)b(q)	7 . t t . .	. . t . .	. . . . t	t . . .
i	8 . t t . .	. . t . .	. . . . t	t . . t
k ki	9 . t t . .	. . t t .	. . . . t	t . . t
g e ge	10 . t t . .	. . t . .	. . . . t	t t . .
d	11 . t . . t	. . t . .	. . . . t	t t . .
dr	12 . t t . t	. . t . .	. . . . t	t t . .
r.	13 . t . t t	. . t . .	. . . . t	t t . .
j j_{10}	14 . t t . .	. t t . .	. . . . t	t t . .
* /	15	. . . . .	. . . t .	. . . .
+ -	16 t	. . . . .	. . . . .	. . . .
r^+ r-	17 . t t . .	. t t . t	. . . . t	t t . .
r	18 . t t t t	. t t . t	. . . . t	t t . .
f fr	19 . t t t t	. t t . t	. . . . t	t t t .
δ	20 . t t t t	t t t . t	. . . . t	t t t .
λ	21 . t t . .	. . t t .	t . . . t	t . . t

the assignment of new numbers. As a result, there are rows (e.g. 14) which are not used at all by the processor, and simply occupy space in the table.

There is no way to provide the ability inside of RAVEL to reverse the direction of its scan without providing, in essence, two complete sets of tables and a parameter (denoting the scan direction) that is used to select between the tables. Since this is the case, we feel that it is appropriate to leave the switching of tables to the routine that calls RAVEL, and allow RAVEL to assume that the tables provided are the correct ones. We shall discuss later the case where an additional formal parameter must be provided for RAVEL so that the desired direction of scan can be indicated to the calling routine each time a value is required for the formal parameters called by name.

```
procedure RAVEL (name, empty, BAD, GOOD, A, S, T, Write);
value name, empty;
integer name, empty;
Boolean array T;
integer array A, S;
label BAD, GOOD;
procedure Write
   begin comment block 1;
   integer W0, W1, W2, W3, W4, W5, h, i, z, y, L, q, SJ;
   Boolean AL;
   Boolean procedure Nogo (P, Q);
    value P, Q; integer P, Q; Nogo: = P = 0 ∨ ¬ T [P, Q];
L1:  W1: = S[empty, 2];
     W2: = S[name, 2];
     W3: = S[A[0], 1];
     W4: = S[A[0], 3];
     AL: = W3 > 2 ∨ Nogo (W4, W2) ∧ Nogo (W1, W2);
     if AL then begin if A[0] = name then go to BAD
        else begin Write (1, S[name, 5]); go to GOOD end end;
     W3: = name;
     h: = q: = W1: = 0;
     i: = z: = −1;
     y: = 1;
     L: = 4;
L2:  W4: = A[h];
     W5: = S[W4, 1];
     if W5 > 2 ∨ W4 = 0 then begin SJ: = 2; go to L4 end;
     W0: = S[W3, 2];
     AL: = true;
L3:  W2: = S[W4, 3];
     if Nogo (W2, W0) then begin if AL then go to L31 else SJ: = 3 end
        else begin SJ: = 4; z: = z + 1 end;
L4:  begin comment block 2;
     own Boolean array B[0: z];
     own integer array G, F, K, W[0: z];
        if SJ ≠ 4 then go to L7;
L5:  B[z]: = AL;
     if AL then h: = h + 1;
     G[z]: = W0;
     F[z]: = W3;
     K[z]: = y;
     W[z]: = h;
     if W5 = 0 then W4: = S[W4, 4];
L6:  i: = i + 1;
     SJ: = 1;
L7:  begin comment block 3;
     switch OUT: = BAD, C2, C3, C4, C5;
     Boolean SQ
     switch WHERE: = L8, L17, L30;
     own integer array R, E, J[0:i];
           go to WHERE [SJ];
```

```
L8:  R[i]:=0;
     E[i]:=L;
     J[i]:=W4;
     if W5≠0 then go to if Nogo(S[J[i],3], G[z]) then L23
        else L9A;
L9:  W1:=S[J[i],3];
     W2:=S[J[i],1];
     W3:=S[J[i],2];
     if W2≠5 then go to L12;
     if Nogo(S[J[i],4], G[z]) then go to L23;
L9A: W4:=S[W3,4];
     comment from this point on this version of RAVEL
        is the same as that given in Figure 3.1;
```

FIG. 4.3.

Elements of N(L) in the Input

The ability to recognize elements of N(L) in the input string allows RAVEL to be used in a more general way. It is obvious that RAVEL as presented in 3.1 is the first pass of a type of compiling system that has begun to receive much attention in the literature [1, 2, 4, 5, 8, 9, 11, 12, 13, 14, 17, 18]. The inclusion of the ability to recognize elements of N(L) as well as elements of B(L) in the input string allows the use of the first pass of such a compiling system to answer the question, "If all of the elements of N(L) in the input string were replaced by strings of elements of B(L) which were correctly named by the elements of N(L) which they replaced, would the input string be a valid input string?". The consequence is that a programmer may debug his program piecemeal, even though the program is not written in machine language; further, the pieces which are assumed to be bug-free (those represented in the input by ele-

TABLE 4.4

	Column in S			
1	2	3	4	5
0	(not used)	row in T	link to packet	(not used)
1	col in T	row in T	link to packet	rule number
2	col in T	0	(not used)	rule number
3	symbol	alternative choice	(not used)	(not used)
4	metacomponent	alternative choice	row in T	(not used)
5	metaresult	alternative choice	row in T	rule number

ments of N(L) need have no relationship to the subroutine structure of the program. This is in contrast to the conventional technique for piecemeal debugging where each subroutine is separately corrected and then inserted into the program.

The addition of this feature to RAVEL requires a number of minor changes to 3.1. Figure 4.3 is a version of RAVEL which will accept elements of P(L) in the input; it requires a minor addition to the Syntax matrix S. This addition consists of assigning a rule number to column 5 of row types 1 and 2; when an element of N(L), say n_k, appears in the input, 4.3 generates a (ficticious) rule of the form $r_{k+m,k}$ and assumes it to have been an element of R(L).

These additional numbers are already in Figures 2.20 and 4.1, where they were inserted without earlier comment. TABLE 4.4 shows the complete format of the row types in the S matrix.

Some Possible Extensions

One interesting direction to extend RAVEL takes into account that RAVEL already has a mechanism (used, in 3.1 and 4.3, only when errors have been encountered) to reset all of the necessary parameters to try for alternatives (if any) at the point of error. This resetting, indeed, is the purpose for the various push-down lists in Block 2.

We first introduce a rule in addition to 1.3, 1.4, and 1.5, and assume explicitly that the direction of scan is from left to right. The additional rule is

$$n_j ::= e_i \lfloor \underline{n_k} \Rightarrow r_{ij} \lfloor \underline{k} \qquad (4.5)$$

We shall read such a rule (in the metalanguage) "n_j names the string e_i, but only when e_i is followed by n_k.". "Followed", in the preceding, implies the direction of scan. Note that n_k is *not* named by n_j; only e_i is. We can interpret this meaning with respect to RAVEL by noting that the effect is that of recognizing n_k as an additional metacomponent of e_i *even* if n_k is recognized in the input string, the parameters of RAVEL are reset from the push-down lists as though an error had occurred in recognizing n_k. In the normal recognition process (assuming no subsequent errors), once a character in the input string has been inspected, it is never reinspected; the "$\lfloor\_$" allows the input string to be inspected more than once, starting from a specified point (the point in the input string where e_j terminates) and proceeding left-to-right.

We note that, with only this addition to RAVEL, there is no way of detecting that a rule has been written of the form

$$n_j ::= e_i \lfloor n_k \lfloor n_p \tag{4.6}$$

and the interpretation would be "n_j names the string e_i, but only when followed by both n_k *and* n_p.". This seems to be an interesting lack of restriction.

By analogy with the above, but assuming a direction of scan from right to left, we can introduce the operator "$\rfloor$" with the same reading.

With the introduction of an additional fromal parameter to RAVEL so that RAVEL may indicate to the calling routine its current direction of scan, a single additional convention allows us to use both the "$\rfloor$" and the "$\lfloor$" in the same rule: (We introduced "$\lfloor$" as prefixed to n_k; we shall use "$\rfloor$" as suffixed.) This additional convention is simply that if a "$\rfloor$" ("$\lfloor$") is encountered while scanning from left to right (right to left) the direction of scan is reversed for the recognition (or failure to recognize) of the n_k associated with the operator; the direction of scan reverts to its original direction when the n_k is (or is not) recognized. Since this must be understood recursively, we shall require an additional own Boolean array to be declared in Block 2, to store the indication of the direction of scan at the various levels of iteration, as well as an additional formal parameter to control table switching.

As we mentioned above, there is nothing in RAVEL to prevent a rule of the form

$$n_j ::= n_a \rfloor n_b \rfloor \lfloor n_c\ e_k \lfloor n_d\, n_e \rfloor \tag{4.7}$$

which can be read "n_j names the string e_k if and only if there is an n_a *and* and n_b to the left of e_k and extending to the left, an n_c beginning at the left of e_k but extending to the right, an n_d beginning at the right of e_k and extending to the right, and finally, an n_e beginning at the right of e_k and extending to the left.".

The only way that seems to make sense to form the Transition matrix, given rules containing "$\rfloor$" and "$\lfloor$", is to ignore both the operators and the n_k's associated with them, and proceed as described in Chapter II. It is still broadly open as to whether there is a more efficient way of avoiding the searching out of fruitless paths which can be extended to accomodate operators in the grammar.

It should be further noted that there are indications that the

addition of these two operators may introduce undecidability in the sense of RAVEL's never terminating. This question has been been raised and not put down; it is mentioned here as a direction in which further study is desirable.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the patience and construc-time criticism of the members of the Mechanical Languages Projects of the University of Pennsylvania and the members of Sub-Committee X3.4.2 of the American Standards Association. In particular, Harry Freedman, of the University, suffered through many revisions of the basic ideas contained herein; Joy Harrell, also of the University, typed both text and tables with great care; and last, my wife Jill nagged me into completing the paper.

REFERENCES

1. Brooker, R. A. and Morris, D; 1962; A General Translation Program for Phrase Structure Languages; JACM Vol 9 Nr 1.
2. Burroughs Algebraic Compiler; 1961; Burroughs Corporation Bulletin 220-21011-P.
3. Chomsky, N; 1959; On Certain Formal Properties of Grammars; Information and Control, Vol 2 Nr 2.
4. Evans, A. Jr., Perlis, A. J. and VanZoren, H; 1961; The Use of Threaded Lists etc.; CACM Vol 4 Nr 1.
5. Glennie, A. E; 1960; On the Syntax Machine and the Construction of a Universal Compiler; Carnegie Institute of Technology.
6. Gorn, S. et al; 1959; Common Programming Language Task, Part I; University of Pennsylvania.
7. Gorn, S. and Parker, E. J; 1960; Common Programming Language Task, Part I; University of Pennsylvania.
8. Holt, A. W. and Turanski, W. J; 1959; Common Programming Language Task, Part II; University of Pennsylvania.
9. Holt, A. W. Turanski, W. J. and Parker, E. J; 1960; Common Programming Language Task Part II; University of Pennsylvania.
10. Ingerman, P. Z; 1961; Dynamic Declarations; CACM Vol 4 Nr 1.
11. Irons, E. T; 1961; A Syntax Directed Compiler for Algol 60; CACM Vol 4 Nr 1.
12. Irons, E. T. and Feurzeig, W; 1961; Comments on the Implementation of Recursive Procedures etc; CACM Vol 4 Nr 1.
13. Leonard, G. F; 1961; The CL-1 Programming System Users Manual; Technical Operations, Inc, Report TO-B-61-3.
14. McCarthy, J. et al; 1960; LISP Programmers' Manual; Computation Center, Massachusetts Institute of Technology.

15. Mechanical Translation Group; 1959; A New Approach to the Mechanical Syntactical Analysis of Russian; National Bureau of Standards Report 6505 and supplement.
16. Naur, P; 1960; Report on the Algorithmic Language Algol 60; CACM Vol 3 Nr 5.
17. Newell, A. et al; 1960; The Elements of IPL Programming; The Rand Corporation report P-1897.
18. Rhodes, I; 1960; Hindsight in Predictive Syntactic Analysis; National Bureau of Standards report 7034.
19. Sattley, K; 1961; Allocation of Storage for Arrays in Algol 60; CACM Vol 4 Nr 1.
20. Warshall, S; 1962; A Theorem on Boolean Matrices; JACM Vol 9 Nr 1.

JACM = Journal of the Association for Computing Machinery.
CACM = Communications of the Association for Computing Machinery.

A GENERAL PROCESSOR FOR CERTAIN FORMAL LANGUAGES

M. Paul

Institute for Applied Mathematics,
Gutenberg University,
Mainz, Germany

0. INTRODUCTION

The development in the preceding years of the various fields of problems to be solved with the aid of digital computers brought up the postulation of convenient methods for programming computers. For this purpose most computers have assembly routines, which allow symbolic addresses to be used, and furthermore most of them can handle symbolic operations too. So far the so-called translation problem has not been serious. It has become serious and slightly difficult, however, since languages (FORTRAN, ALGOL, COBOL a.e.) have been created, which differ to a large extent from the usual machine codes of digital computers.

Therefore the Institutes for Applied Mathematics in Zürich, Munich, Mainz, and Darmstadt started in 1956 a project whose aim was not only to introduce an algorithmic language for communicating and programming processes of Numerical Analysis, but also to define the new languages in such a way that its translation into machine languages could be carried out with a reasonable amount of time and storage. Implicitly formulated, the structure of the language should have such properties, that the translation could be handled by means of the cellar-principle introduced recently by K. Samelson and F. L. Bauer (9), (10). This ZMMD-project has formed a basis for the European proposal at the first ALGOL-conference in 1958.

As we shall see ALGOL may be translated by a cellar processor, and moreover ALGOL may be considered in a quite natural sense as a language, whose syntax is unique with the exception of a few minor points.

Finally we shall define a class of formal languages, which

may be translated by a cellar-processor, similar to the ALGOL-processors.

1. REDUCING FORMAL LANGUAGES

Pursuing the steps outlined above, we shall define at first the central concept of the following analysis by generalizing the description of the ALGOL-syntax in the ALGOL 60 Report (1), (2).

A *reducing formal language* (r.f.l.) γ is a nonempty set $M(\gamma)$ of finitely many elements with the following property: To certain ordered n-tupels ($n = 1, 2, \ldots, N(\gamma)$) of elements of $M(\gamma)$, called *syntactical elements* of γ, there is assigned an uniquely determined element of $M(\gamma)$. We may write these assignments in Backus-notation as:

$$q_{n,i} : := p_{n,i}$$

where $q_{n,i} \in M(\gamma)$ and $p_{n,i} \in M^n(\gamma)$, and call them *productions* of γ.

The uniqueness of these productions may be expressed by: $i \neq j$ implies $p_{n,i} \neq p_{n,j}$ for all n. The characteristic integer $N(\gamma)$ of γ will be called *maximum length of productions* in γ. Furthermore the set of syntactical elements, which are not assigned to any n-tupel, is called the *alphabet* $A(\gamma)$ of γ. We shall give a simple example for a reducing formal language γ_1: Set of syntactical elements in γ_1:

$$M(\gamma_1) = \{V, N, (,), +, -, \times, /, P, Q, F, T, E\}$$

Production in γ_1:

P : : = +	P : : = −	Q : : = ×
Q : : = /	F : : = V	F : : = N
F : : = (E)	T : : = F	T : : = TQF
E : : = T	E : : = EPT	

Maximum length of productions in γ_1:

$$N(\gamma_1) = 3$$

Alphabet of γ_1:

$$A(\gamma_1) = \{V, N, (,), +, -, \times, /\}$$

This example is a part of ALGOL and describes the rational expressions in variables and numbers. We must establish, however, that the complete ALGOL in its present form is not a re-

ducing formal language, because of its productions not being unique.

One has, for instance

<variable identifier> : : = <identifier>
<array identifier> : : = <identifier>

This deviation of ALGOL 60 from a reducing formal language in the sense of the previous definition, however, is not serious. This deficiency may be avoided by simply identifying <variable identifier> with <array identifier> throughout the syntactical description of ALGOL. Clearly the structure of ALGOL is weakened by this identification process for all syntactical elements, which are assigned to the same 1-tupel, but this means only a loss of the possibility of certain interpretations and not at all of syntactic properties of ALGOL. With respect to these remarks, in the sequel, we shall consider ALGOL as a reducing formal language.

2. STRINGS, REDUCTIONS AND WORDS

The main syntactical elements within a reducing formal language are coming immediately from our definition.

We shall call an ordered m-tupel of syntactical elements in a r.f.l. γ a *string with m elements* in γ.

Furthermore the only natural way leading to operations on the strings in a r.f.l. γ is given from the productions in γ. That means, if a string in γ contains a substring, to which a syntactical element X is assigned by the productions of γ, then we may replace that substring by the element X.

Such a transformation from a string into another string in γ will be called a *reduction* of the initial string into the transformed string.

Now, it may happen, that the reduced string again permits a reduction and so on. Therefore we distinguish a special class of strings, which allow a finite sequence of reductions to a single syntactical element Z.

Such strings will be called *words* for Z in γ.

Examples:

E)PQ(is a string with 5 elements in the r.f.l. γ_1, which does not permit any reduction.

VN + VP is a string with 5 elements in the r.f.l. γ_1, which permits reductions, for instance FN + VP and VF + VP are possible reductions. Our string is, however, no word in γ_1.

An example for a word in γ_1 would be V + T. The corresponding reduction sequence is the following:

V + T F + T T + T E + T EPT E

That means, V + T is a word for the syntactical element E in γ_1.

3. SIMPLIFICATION OF REDUCING FORMAL LANGUAGES

In this section we simplify the reducing formal languages with respect to the maximum length of productions. This step has been proved in the work of the ALCOR-group to be important in the treatment of the translation problem. The simplification process runs as follows:

We shall give a recursive construction of a r.f.l. $\mathfrak{R}$ with $N(\mathfrak{R}) = 2$ from a given r.f.l. γ with $N(\gamma) \geq 2$. Besides the simplification with respect to the maximum length of productions, $\mathfrak{R}$ shall have the following properties:

a) The alphabet $A(\mathfrak{R})$ of $\mathfrak{R}$ is the same as the alphabet $A(\gamma)$ of γ and every syntactical element of γ is a syntactical element of $\mathfrak{R}$ too. Therefore every string in γ may be considered also as a string in $\mathfrak{R}$.

b) Every word in γ considered as a string in $\mathfrak{R}$ is a word in $\mathfrak{R}$ too.

Therefore, we define a sequence $R_0, R_1, \ldots, R_{N(\gamma)-2}$ of reducing formal languages with the respective maximum length of productions $N(R_i)$.

$R_0 = \gamma$

For $i = 1, 2, \ldots, N(\gamma) - 2$ we obtain R_i from R_{i-1} by the following algorithm:

At first we take the productions of the maximum length $N(R_{i-1})$ of R_{i-1}. For every one of these productions

$X ::= X_1X_2X_3 \ldots X_{N(R_{i-1})}$

and all pairwise different pairs (X_1, X_2) we then produce an element E which is not a syntactical element of R_{i-1}.

All elements produced in this way must be pairwise different. Finally the syntactical elements of R_i are the syntactical elements of R_{i-1} together with the elements just produced.

The productions of R_i are obtained from the productions of R_{i-1} by cancelling all productions of the length $N(R_{i-1})$ and adding instead of the cancelled production

$$X ::= X_1X_2X_3 \ldots X_{N(R_{i-1})}$$

the corresponding productions

$$E ::= X_1X_2$$

$$X ::= EX_3 \ldots X_{N(R_{i-1})}$$

If the production

$$Y ::= X_1X_2$$

already exists in R_{i-1} then this latter production is omitted too, and will be replaced by the new one

$$Y ::= E$$

One can show by induction quite easily that for all $i = 0, 1, \ldots, N(\gamma) - 2$ the R_i are reducing formal languages with the maximum length of productions $N(R_i) = N(\gamma) - i$. In the same way, one proves all R_i to have the properties a) and b). Therefore, $R_{N(\gamma)-2}$ is the language $\mathfrak{R}$ for which a construction was postulated.

For instance, for our former example γ_1 the simplification process would be the following: It consists only of the step from $R_0 = \gamma_1$ to R_1. We have to produce three elements. Let these elements be G, H and J. Then a simplificated formal language $R^{(1)}$ corresponding to γ_1 would be:

Set of syntactical elements in $R^{(1)}$:

$$M(R^{(1)}) = \{V, N, (,), +, -, \times, /, P, Q, F, T, E, G, H, J\}$$

Productions in $R^{(1)}$:

P : : = +	P : : = −	Q : : = ×
Q : : = /	F : : = V	F : : = N
G : : = (E	F : : = G)	T : : = F
H : : = TQ	T : : = HF	E : : = T
J : : = EP	E : : = JT	

Maximum length of productions in $R^{(1)}$:

$N(R^{(1)}) = 2$

Alphabet of $R^{(1)}$:

$A(R^{(1)}) = \{V, N, (,), +, -, \times, /\}$

Our former example (s.p. 4) V + T as a word for E in γ_1 leads in $R^{(1)}$ to the reduction sequence

V + T F + T T + T E + T EPT JT E

4. TRANSITIVE RELATIONS IN A REDUCING FORMAL LANGUAGE

The analysis of the structure in a r.f.l. γ is based on three transitive relations as follows:

a) A syntactical element X is said to be a head of a syntactical element Y (symbolized by $X \alpha Y$) in a r.f.l. γ, if there exists a word $W = Z_1 Z_2 \ldots Z_m$ for Y in γ with $X = Z_1$.

b) Similar to a) a syntactical element X is said to be a tail of the syntactical element Y (symbolized by $X \omega Y$) in γ, if there exists a word $W = Z_1 Z_2 \ldots Z_m$ for Y in γ with $X = Z_m$.

c) Finally we shall symbolize by $X \mu Y$ the fact that X is a word for Y in γ.

As an important step towards the goal mentioned in the introduction we shall now define certain subsets of $M(\gamma)$ by means of the relations α, ω, and μ. This definition will be strongly formalized in order to shorten the wording of the definition:

Let γ be a reducing formal language and let X and Y be syntactical elements of γ. Then we introduce subsets of $M(\gamma)$ with respect to X and Y as follows:

$P(X, Y) = \{Z \mid (\exists \underline{X}, \overline{Y} \in M(\gamma))\, X \omega \underline{X} \wedge Y \alpha \overline{Y} \wedge \underline{X}\, \overline{Y}$ is a production for Z in $\gamma\}$,

$P_\alpha(X, Y) = \{Z \mid (\exists \overline{Y} \in M(\gamma))\, Y \alpha \overline{Y} \times X \overline{Y}$ is a production for Z in $\gamma\}$,

$P_\mu(X, Y) = \{Z \mid (\exists Y^* \in M(\gamma))\, Y \mu Y^* \wedge XY^*$ is a production for z in $\gamma\}$.

We shall extend our example and give both the relations and the sets $P(X, Y)$, $P_\alpha(X, Y)$, $P_\mu(X, Y)$ for all $X, Y \in M(R^{(1)})$ of the language $R^{(1)}$ in section 3.

In the following relation tables an asterisk in a row marked by X and in a column marked by Y denotes that the relation holds for the ordered pair (X, Y). Empty places mean that the relation does not hold. Similar, a field of intersection of a row marked by X and a column marked by Y in the set-tables contains the elements of the sets $P(X, Y)$ and respectively of the sets $P_\alpha(X, Y)$ and $P_\mu(X, Y)$.

5. THE TRANSLATION PROBLEM

A r.f.l. $\mathfrak{R}$ with $N(\mathfrak{R}) = 2$ has the following important property:

Let W_X be a word for X in $\mathfrak{R}$. Then W_X is X or there exists a reduction sequence

$f_{j_r,k_r} \ldots f_{j_2,k_2} f_{j_1,k_1} W_X = X$ such that, for $\rho = 1, 2, \ldots, r-1$, holds: $k_\rho + 1 \geq j_\rho$.

Here we have introduced the abbreviation $f_{j,k} X_1 X_2 \ldots X_m$ with the following meaning:

The string $X_1 X_2 \ldots X_m$ will be reduced with respect to the indexpair (j, k). As a consequence of $N(\mathfrak{R}) = 2$, in $\mathfrak{R}$ holds $j = k$ or $j = k - 1$. A reduction sequence with the property mentioned above will be called a canonical one.

Remark: As a matter of fact, in every r.f.l. γ with $N(\gamma) \geq 2$ holds that to every word in γ there belongs always a canonical reduction sequence. In the translation problem, however, we are interested only in simplificated reducing formal languages $\mathfrak{R}$ with $N(\mathfrak{R}) = 2$.

Now, we may formulate the translation problem with respect to our introductory remarks. Given a r.f.l. $\mathfrak{R}$ with $N(\mathfrak{R}) = 2$, the problem is to find an algorithm, which decides whether a string in $\mathfrak{R}$ is a word for a syntactical element X in $\mathfrak{R}$ or not. Moreover, in the former case the algorithm should give a canonical reduction sequence from W_X to X, if $W_X \neq X$.

This problem may be solved for a class of reducing formal languages, which we shall define in a strictly formalized manner.

We consider the class of all languages $\mathfrak{R}$ with a maximum length of productions $N(\mathfrak{R}) = 2$ and the following 4 properties with respect to a syntactical element $Z \in M(\mathfrak{R})$:

Relation α in $R^{(1)}$

Relation μ in $R^{(1)}$

Relation ω in $R^{(1)}$

Subsets $P\alpha$ (X, Y) in $R^{(1)}$

Subsets $P\mu$(X, Y) in $R^{(1)}$

Subsets P (X, Y) in $R^{(1)}$

1. $(\forall X \in M(\mathfrak{R}))(\forall r \geq 1)\, f_{j_r,k_r} \ldots f_{j_2,k_2} f_{j_1,k_1} X \neq X$
2. $(\forall Y \in M(\mathfrak{R}))(\forall \chi \in A(\mathfrak{R}))(\underset{0,1}{\exists} \underline{Y} \in M(\mathfrak{R}))(Y \mu \underline{Y} \wedge (\exists \underline{Z} \in P_\alpha(\underline{Y}, \chi))\ \underline{Z} \alpha Z)$ *)
3. $(\forall X, Y \in M(\mathfrak{R}))(\underset{0,1}{\exists} \underline{Z} \in P_\mu(X, Y))\ \underline{Z} \omega Z$
4. $(\forall X, Y \in M(\mathfrak{R}))(\forall \chi \in A(\mathfrak{R}))$
   $(\forall R \in P_\mu(X, Y))\, P(R, \chi) = \phi \wedge (\underset{0,1}{\exists} \underline{Y} \in M(\mathfrak{R}))$
   $(Y \mu \underline{Y} \wedge (\exists S \in P_\alpha(\underline{Y}, \chi))\, P(X, S) \neq \phi) \vee$
   $\vee (\underset{0,1}{\exists} R \in P_\mu(X, Y))\, P(R, \chi) \neq \phi \wedge (\forall \underline{Y} \in M(\mathfrak{R}))$
   $(Y \mu \underline{Y} \rightarrow (\forall S \in P_\alpha(\underline{Y}, \chi))\, P(X, S) = \phi).$

For a r.f.l. of the class just defined the following holds:

For every word W_Z for the syntactical element Z there exists exactly one canonical reduction sequence. Moreover, there exists an algorithm using the cellar-principle which decides whether a string is a word for Z or is not. In the former case the algorithm leads to the uniquely determined canonical reduction sequence from the given string to Z.

By cellar-principle it is meant that the algorithm uses a storage sequence following the last-in-first-out-method, and furthermore the algorithm depends in every step only on the last two elements in this storage sequence and on the next element of the given string read from left to right.

Now, if one has a canonical reduction sequence, one may produce a program in a computer by productions, which correspond to the productions of the reduction sequence. This correspondence depends on the semantic part of the language as well as on the special properties of the computer. It should be analysed, therefore, apart from researches of the syntactical structure of languages. What has been done in the past on this field, may be seen from (3), (4), (6), (8), (9), (10), (11), and from the work of the ALCOR-group in building ALGOL-translators.

Remark: With the exception of a few points, for instance <relation>, ALGOL turns out to lead to a simplificated r.f.l., which belongs to our class. This means that ALGOL in its essential parts (e.g. the example $R^{(1)}$) may be translated by a general processor.

---

* The symbol $\underset{0,1}{\exists}$ denotes the quantificator "there is at most one".

REFERENCES

1. J. W. Backus et al.: Report on the Algorithmic Language ALGOL 60, Num. Math. 2(1960), 106-136.
2. J. W. Backus et al.: Report on the Algorithmic Language ALGOL 60, Comm. Assoc. Comp. Mach. 3(1960), 299-314.
3. A. A. Grau: Recursive Processes and ALGOL Translation, Comm. Assoc. Mach. 4(1961), 10-15.
4. E. T. Irons: A Syntax Directed Compiler for ALGOL 60, Comm. Assoc. Comp. Mach. 4(1961), 51-55.
5. V. Kudielka et al.: Extension of the Algorithmic Language ALGOL, Mailüfterl—Vienna, DA-91-591-EUC-1430. Final Report, (1961).
6. P. Lucas: Die Strukturanalyse von Formelübersetzern, Mailüfterl Wien, (1961).
7. H. Rutishauser: Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen, Mitt. Inst. f. Angew. Math. ETH Zürich, Nr. 3, (1952).
8. K. Samelson: Probleme der Programmierungstechnik, Intern. Kolloquium über Probl. der Rechentechnik, Dresden, 1955.
9. K. Samelson und F. L. Bauer: Sequentielle Formelübersetzung Elektron. Rechenanl. 1 (1959), 176-182.
10. K. Samelson and F. L. Bauer: Sequential Formula Translation, Comm. Assoc. Comp. Mach. 3 (1960), 76-83.
11. J. H. Wegstein: From Formulas to Computer Oriented Language, Comm. Assoc. Comp. Mach. 2 (1959), 6-8.
12. R. Baumann: ALGOL-Manual der ALCOR-Gruppe, Elektron. Rechenanl. 3 (1961), 206-212, 259-265.

FORMAL STRUCTURE OF ALGOL AND SIMPLIFICATION OF ITS DESCRIPTION

Karel Čulík

Československá Akademia Věd
Matematický Ústav
Prague, Czechoslovakia

1. SEMIALGORITHM GENERATING THE LANGUAGE ALGOL.

Let be Γ_t and Γ_n the set of all basic symbols and the set of all metalinguistic variables resp. used in the syntactic recurrent definitions of Reference (1). It may be ommited the variable <code> from Γ_n and joined to Γ_t and also the symbols **comment** and <any sequence not containing;> may be omitted from Γ_t and Γ_n resp.

A syntactic definition of Reference (1) is said to be elementary or composed if it contains the metasymbol | or not (e.g. a syntactic definition $\alpha ::= \beta|\gamma|\delta$ is composed of the following elementary ones $\alpha ::= \beta$, $\alpha ::= \gamma$ and $\alpha ::= \delta$). Now let be $\mathfrak{R}$ the set of all elementary definitions of which are composed the particulary syntactic definitions contained in chapters 2.-5. of Reference (1). Further we join to $\mathfrak{R}$ a trivial syntactic definition <empty>::=<empty> and we put $\Gamma = \Gamma_t \cup \Gamma_n$.

Let be Γ^∞ and Γ_t^∞ the free semigroup the generators of which are the elements belonging to Γ and Γ_t resp. We suppose that <empty> is the unit element of the semigroup Γ^∞ (instead of the introducing of <empty> in Reference (1), section 1.1). The elements of Γ^∞ are denoted by small Greek letters and the subsets of Γ^∞ are said to be languages.

The semialgorithm $S = (\Gamma, \mathfrak{R}; \Gamma_t)$ generating the language ALGOL (as a subset of Γ_t^∞) is a mapping (i.e. $S\Phi \subset \Gamma^\infty$ for each $\Phi \subset \Gamma^\infty$) defined in a certain constructive way as follows:

(1) $\bar{\mathfrak{R}} = \{(\gamma \rightarrow \delta) \mid$ there are ξ and ζ and $(\alpha ::= \beta) \in \mathfrak{R}$, such that $\gamma = \xi\alpha\zeta$ and $\delta = \xi\beta\zeta\}$,

(2) $Q\Phi = \{\xi \mid$ there are ξ_i such that $(\xi_i \rightarrow \xi_{i+1}) \in \overline{\mathfrak{R}}$ for $1 \leq i < n$, where $n \geq 2$ and $\xi_i \in \Phi$ and $\xi_n = \xi\}$,

(3) $S\Phi = Q\Phi \cap \Gamma_t^\infty$.

In References (1) and (4) there are shown only some examples of the using of the syntactic recurrent definitions. We conjecture that an exact and complete meaning of these definitions is stated by our semialgorithm S. Namely $S\Gamma_n$ is exactly the set of all expressions belonging to the language ALGOL described in Reference (1). Especially $S\alpha$ is the set of all "values" of the metalinguistic variable $\alpha \in \Gamma_n$ in the sense of Reference (1), section 1.1.

Obviously the notion of the semialgorithm is very strong connected with the well known notions as the phrase structure grammar of N. Chomsky and the semi-Thue (combinatorial) system of M. Davis or the normal algorithm of A. A. Markoff.

It follows by (2) and (3) that the mappings Q and S are generated from the many-valued mappings defined in Γ^∞, i.e. $Q\Phi = \bigcup_{\varphi \in \Phi} Q\varphi$ and $S\Phi = \bigcup_{\varphi \in \Phi} S\varphi$ for all $\Phi \subset \Gamma^\infty$, and therefore they satisfy many well known properties of abstract closure topologies, e.g. if $\Phi \subset \Psi$ then $Q\Phi \subset Q\Psi$ and $S\Phi \subset S\Psi$, $QQ\Phi = Q\Phi$ and also $SS\Phi = S\Phi$ or $\Phi \subset Q\Phi$, but it may be $\Phi \not\subset S\Phi$ etc.

If Δ_α denotes the set of all β's such that $(\alpha \rightarrow \beta) \in \overline{\mathfrak{R}}$ then by (2) and (3) follows

(4) $Q\alpha = \bigcup_{\beta \in \Delta_\alpha} Q\beta$ and $S\alpha = \bigcup_{\beta \in \Delta_\alpha} S\beta$.

If $\alpha ::= \beta \mid \gamma \mid \delta$ is an arbitrary syntactic definition of Reference (1) then always $\Delta_\alpha = \{\beta, \gamma, \delta\}$ and therefore by (4) follows $S\alpha = S\beta \cup S\gamma \cup S\delta$.

The above mentioned properties of semialgorithms are quite general, but there are also very special properties characterising our semialgorithm S and therefore also the language ALGOL. First of all the substitution rules contained in $\mathfrak{R}$ have the following special form

(5) if $(\alpha ::= \beta) \in \mathfrak{R}$ then either $\alpha =$ <empty> and then also $\beta =$ <empty> or $\alpha \neq$ <empty> and then $\alpha \in \Gamma_n$, i.e. α is a single symbol of Γ_n.

If $\alpha = \alpha_1 \alpha_2 \ldots \alpha_n$, where $\alpha_i \in \Gamma$ for $1 \leq i \leq n$, then in virtue of (5) follows

(6) $S\alpha = S\alpha_1 S\alpha_2 \ldots S\alpha_n$,

where the operation $\Phi\Psi = \{\varphi\psi \mid \varphi \in \Phi \text{ and } \psi \in \Psi\}$ is the usual concatenation of languages Φ and Ψ.

At this occasion we note that the property (5) shows that the language ALGOL is essentially a type-2 (or context-free) language in the sense of Reference (2). On the other side there are some particular languages $S\alpha$, $\alpha \in \Gamma$ being only type-3 languages (or finite state languages or regular events in the sense of S. C. Kleene), e.g. S <identifier>, S <integer>, S <number> etc.

The second special property of our semialgorithm concerns the relations among the particular languages $S\alpha$. First of all we define a binary relation of dependence $\succ$ in Γ as follows

(7) $\alpha \succ \beta$, where $\alpha, \beta \in \Gamma$, if there is $(\alpha :: = \gamma) \in \mathfrak{R}$ such that $\gamma = \gamma'\beta\gamma''$.

We say that a language $S\alpha$ is more complicated than a language $S\beta$ if $\alpha \succ \beta$.

Let be $\Sigma_1, \Sigma_2, \ldots, \Sigma_n$ the all equivalence classes of an equivalence relation $\sim$ defined in Γ in the following way

(8) $\alpha \sim \beta$, where $\alpha, \beta \in \Gamma$, if either $\alpha = \beta$ or there are α_i and β_j such that $\alpha_i \succ \alpha_{i+1}$, $\beta_j \succ \beta_{j+1}$ for all $1 \leq i < p$, $1 \leq j < q$ and $\alpha_1 = \alpha = \beta_q$, $\alpha_p = \beta = \beta_1$.

We note that there are some equivalence classes Σ_i containing more than one element, e.g. the metalinguistic variables <variable>, <arithmetic expression>, <Boolean expression> etc. belong to the same class. This fact is characteristic for the language ALGOL and it causes probably some troubles by understanting and also by constructing of the translator of ALGOL.

In spite of all that if we join to every subset $\Sigma \subset \Gamma$ the following subset

(9) $\tilde{\Sigma} = \{\xi \in \Gamma \mid$ there are ξ_i such that $\xi_i \succ \xi_{i+1}$ for all $1 \leq i < k$, where $k > 1$ and $\xi_1 \in \Sigma$, $\xi_k = \xi\}$,

it is possible to establish (in many different ways) a simple order $\Sigma_1, \Sigma_2, \ldots, \Sigma_n$ of all equivalence classes such that

(10) $\tilde{\Sigma}_m' \subset \bigcup_{i=1}^{m} \Sigma_i$ for all $m = 1, 2, \ldots, n$.

According to this simple order we shall represent a language $S\alpha$ using the representations of the languages which are allways less complicated than $S\alpha$.

2. REPRESENTATION OF PARTICULAR LANGUAGES OF ALGOL.

Let be Γ_k the set of all elements $\alpha \in \Gamma$ such that the representation of the language $S\alpha$ is known, i.e. this representation has been constructed already. In the beginning we may put immediately $\Gamma_k = \Gamma_t$, because it is clear that every language $S\alpha$ contains only a single element α for all $\alpha \in \Gamma_t$. Successively allways we take a class Σ_i, where i is the smallest integer such that $\Sigma_i \not\subset \Gamma_k$, and then we take an arbitrary $\alpha \in \Sigma_i - \Gamma_k$. Under these conditions and by (10) it has to be

(11) $\quad \tilde{\alpha} = \tilde{\Sigma}_i \subset \Gamma_k \cup \Sigma_i$

and in regard to α and to Σ_i $(\alpha \in \Sigma_i)$ we distinguish the following four cases

(i) $\Sigma_i = \{\alpha\}$, i.e. Σ_i contains a single element, and if $\beta \in \Delta_\alpha$ then $\beta \in \Gamma_k^\infty$,

(ii) $\Sigma_i = \{\alpha\}$ and not (i) but if $\beta \in \Delta_\alpha$ then either $\beta = \beta'\alpha$ or $\beta \in \Gamma_k^\infty$ or $\beta = \alpha\beta''$, where $\beta', \beta'' \in \Gamma_k^\infty$,

(iii) $\Sigma_i = \{\alpha\}$ and neither (i) nor (ii),

(iv) Σ_i contains more than one element.

For the reasons of more clearness by writing of the formulae we shall use the following conventions:

(a) a string $\alpha \in \Gamma^\infty$ denotes the language $S\alpha$ (i.e. we may write $S\alpha = \alpha$),

(b) the symbols $\{$ and $\}$ denote our metabrackets,

(c) we identify the set $\{\alpha\}$ containing a single element α with this element α (i.e. we may write $\{\alpha\} = \alpha$, what enables us to use the set union symbol $\cup$ instead of the usual metamathematical coma being a basic symbol of ALGOL),

(d) the usual symbol of equality $=$ is used as a metasymbol in spite of that it is a basic symbol of ALGOL again, but it is possible to avoid the all possible confusions,

(e) at last we use some special metasymbols denoting the necessary operations in the algebra of languages, e.g. $\cup$, $^\infty$, $^\infty_0$, $*$ and "$[\![, , ,]\!]$".

In the case (i) we get in virtue of (4) the following obvious representation of the language $S\alpha$

(12) $\quad S\alpha = \bigcup_{\beta \in \Delta_\alpha} \beta$,

i.e. we may write e.g. <adding operator> = + ∪ − or <multiplying operator> = × ∪/∪ ÷ etc.

In the case (ii) let be $\Delta_\alpha^{(l)}$, $\Delta_\alpha^{(m)}$, $\Delta_\alpha^{(r)}$ the set of all $\beta \in \Delta_\alpha$ such that $\beta = \beta'\alpha$, β does not contain α, $\beta = \alpha\beta''$ resp. Now one easy sees that

$$(13)\quad S\alpha = \left\{ \bigcup_{\beta \in \Delta_\alpha^{(l)}} \beta' \right\}^{\infty_0} \left\{ \bigcup_{\beta \in \Delta_\alpha^{(m)}} \beta \right\} \left\{ \bigcup_{\beta \in \Delta_\alpha^{(r)}} \beta'' \right\}^{\infty_0},$$

where we use the operation of infinite power Φ^∞ introduced in Reference (3) as follows

$$(14)\quad \Phi^\infty = \bigcup_{n=1}^{\infty} \Phi^n, \text{ where } \Phi^n = \Phi^{n-1}\Phi \text{ and } \Phi^1 = \Phi,$$

$$\Phi^{\infty_0} = \bigcup_{n=0}^{\infty} \Phi^n = \Phi^\infty \cup \{\langle\text{empty}\rangle\}, \text{ where } \Phi^0 = \langle\text{empty}\rangle.$$

It is clear that the operation of infinite power is the generation of a free semigroup.

According to (13) and (14) we may write e.g.

$S\langle\text{identifier}\rangle = \langle\text{letter}\rangle \{\langle\text{letter}\rangle \cup \langle\text{digit}\rangle\}^{\infty_0}$

$S\langle\text{unsigned integer}\rangle = \langle\text{digit}\rangle \langle\text{digit}\rangle^{\infty_0} = \langle\text{digit}\rangle^\infty$ and using also (12) we get

$S\langle\text{integer}\rangle = \langle\text{unsigned integer}\rangle \cup + \langle\text{unsigned integer}\rangle \cup - \langle\text{unsigned integer}\rangle = \{\langle\text{empty}\rangle \cup + \cup -\} \langle\text{unsigned integer}\rangle = \{\langle\text{empty}\rangle \cup + \cup -\} \langle\text{digit}\rangle^\infty$

If we proceeded in the representing so far that the all metalinguistic variables the syntactic definitions of which are introduced in Reference (1), sections 2 and 3, belong to Γ_k, then we may write

$S\langle\text{procedure statement}\rangle = \langle\text{identifier}\rangle \cup \langle\text{identifier}\rangle (\{\langle\text{string}\rangle \cup \langle\text{arithmetic expression}\rangle \cup \langle\text{Boolean expression}\rangle \cup \langle\text{designational expression}\rangle \cup \langle\text{identifier}\rangle\} \{, \cup) \langle\text{letter}\rangle^\infty : (\}\{\langle\text{string}\rangle \cup \langle\text{arithmetic expression}\rangle \cup \langle\text{Boolean expression}\rangle \cup \langle\text{designational expression}\rangle \cup \langle\text{identifier}\rangle\}\}^\infty)$

We note on this occasion that the language $S\langle\text{procedure statement}\rangle$ is a type-? language only relatively to Γ_k as was marked above.

In the remaining cases (iii) and (iv) we shall need a general (n+1)-ary operation $[\![\Phi, \Phi_{\alpha 1}, \ldots, \Phi_{\alpha n}]\!]$, where $n \geq 1$ and the indices $\alpha_i \in \Gamma$ are considered as pure helpfull symbols serving to the denoting the free places in which we may substitute some

elements of Γ^{∞}. We introduce two kinds of this operation as follows

(15) $[\![\Phi, \Phi_{\alpha 1}, \ldots, \Phi_{\alpha n}]\!] = \{\varphi_0 \psi_{i1} \varphi_1 \psi_{i2} \ldots \psi_{ik} \varphi_k \mid \varphi_0 \alpha_{i1} \varphi_1 \alpha_{i2} \ldots \alpha_{ik} \varphi_k \in \Phi$ and $\psi_{ij} \in \Phi_{\alpha ij}$ for all $1 \leq j \leq k\}$,

$[\![\ ]\!]^1 = [\![\ ]\!]$, $[\![\ ]\!]^m = [\![\ [\![\ ]\!]^{m-1}, \ldots]\!]$ and $[\![\ ]\!]^{\infty} = \bigcup_{m=1}^{\infty} [\![\ ]\!]^m$

and further

(15*) $[\![\Phi, \Phi_{\alpha 1}, \ldots, \Phi_{\alpha n}]\!]^* = \{\varphi_0 \psi_{i1} \varphi_1 \psi_{i2} \ldots \psi_{ik} \varphi_k \mid \varphi_0 \alpha_{i1} \varphi_1 \alpha_{i2} \ldots \alpha_{ik} \varphi_k \in \Phi$, where φ_i does not contain any α_j for all $1 \leq j \leq n$, $1 \leq i \leq k$ and $\psi_{ij} \in \Phi_{\alpha ij}$ for all $1 \leq j \leq k.\}$

We note that to the language $[\![\ ]\!]^*$ defined in (15*) does not belong any element containing some of symbols $\alpha_1, \ldots, \alpha_n$, if it is the case for the languages $\Phi_{\alpha 1}, \ldots, \Phi_{\alpha n}$. The operation (15) concerns only the quasialorithm Q introduced in (2), but together with the operation (15*) it may be used to the representation of particular languages $S\alpha$.

In the case (iii) let be Δ'_{α} or Δ''_{α} the set of all β's belonging to Δ_{α} such that β contains α or not. In this case it suffices to use only binary operations (15) and (15*), i.e. we put n = 1, and the representation of the language $S\alpha$ looks as follows

(16) $S\alpha = [\![\ [\![\ \bigcup_{\beta \in \Delta'_{\alpha}} \beta\ ,\ \bigcup_{\beta \in \Delta'_{\alpha}} \beta\ ]\!]^{\infty},\ \bigcup_{\beta \in \Delta''_{\alpha}} \beta\ ]\!]^* \cup \bigcup_{\beta \in \Delta''_{\alpha}} \beta$

E.g. we get

$S\langle$open string$\rangle = [\![\ [\![\ `\alpha' \cup \alpha\alpha,\ `\alpha' \cup \alpha\alpha\ ]\!]^{\infty},\ \langle$proper string$\rangle\ ]\!]^*$

where obviously α denotes the free place in which it will be substituted

In the last and most general case (iv) let be $\alpha_1, \ldots, \alpha_n$ the all elements of Σ_i (we suppose $\alpha \in \Sigma_i$ and therefore $\alpha_i = \alpha_j$ for some j). Further let be Δ'_{α} or Δ''_{α} the set of all β's belonging to Δ_{α} such that β contains some α_j or not for $1 \leq j \leq n$ and we may write

(17) $S\alpha = [\![\ [\![\ \Delta'_{\alpha}, \Delta'_{\alpha_1}, \ldots, \Delta'_{\alpha_n}\ ]\!]^{\infty}, \Delta''_{\alpha_1}, \ldots, \Delta''_{\alpha_n}\ ]\!]^*$

We give no example in this case because the number of elements contained in Σ_i is very large in the considered language ALGOL. E.g. to the equivalence class containing the majority of metalinguistic variables of Reference (1), section 3 belong about 30 elements.

3. SIMPLIFICATION OF SYNTACTIC DESCRIPTION OF THE LANGUAGE ALGOL.

By a description of the syntactic structure of the language ALGOL it is possible to eliminate a metalinguistic variable α, $\alpha \in \Gamma_n$ such that

(18) if $\beta \in \Delta_\alpha$ and $\beta = \beta'\gamma\beta''$, where $\gamma \in \Gamma$, then $\gamma \neq \alpha$,

under an assumption that there will be made some necessary modifications concerning the syntactic definitions containing α (naturally the syntactic definition introducing the considered variable α is also eliminated).

Such a simplification of syntactic description allways leeds to a less number of syntactic definitions, but these modified syntactic definitions are not so simple as in Reference (1).

E.g. in Reference (1), section 3.1.1 there are seven syntactic definitions, but using (18) it is possible to eliminate five metalinguistic variables <variable identifier>, <simple variable>, <subscript expression>, <array identifier> and <subcripted variable> and therefore there remain only two modified syntactic definitions

<variable> : : = <identifier> | <identifier>[<subscript list>]
<subscript list> : : = <arithmetic expression> | <subscript list> , <arithmetic expression>

Simultaneously it is necessary to substitute <arithmetic expression> instead of <subscript expression> in 3.5.1, <identifier> instead of <simple variable> in 5.1.1 and again <identifier> instead of <array identifier> in 3.2.1, 4.7.1 and 5.2.1.

In a similar way we may eliminate about a half of the all metalinguistic variables introduced by syntactic definitions in the whole chapters 3 and 4 of Reference (1).

Here we do not discuss the question about an independent semantical significance of some eliminated variables. We note only that these eliminations in the level of our previous representation concern evidentely the case (i), where of course we

construct some relative representations, i.e. the languages used to the representation of a given language may be more complicated than this one. An example of this relative representation was given in the previous section.

At last we may use the relative representations concerning the case (ii) to the further elimination of some metalinguistic variables and their syntactic definitions.

A language $S\alpha$ may be relative represented according the formula (13) if the metalinguistic variable α satisfies the following condition

(19) if $\beta \in \Delta_\alpha$ and $\beta = \beta'\gamma\beta''$, where $\gamma = \alpha$, then either β' does not contain α and $\beta'' = <\text{empty}>$ or $\beta' = <\text{empty}>$ and β'' does not contain α.

E.g. in the above mentioned example we get

$S<\text{variable}> = <\text{identifier}>\{<\text{empty}> \cup [<\text{arithmetic expression}> \{\, , <\text{arithmetic expression}>\}^{\infty}]\}$

REFERENCES

1. Backus J. W. et al., Report on the algorithmic language ALGOL 60, Num. Math. 2, 106-136 (1960)
2. Chomsky N., On Certain Formal Properties of Grammars, Inf. and Control 2, 137-167 (1959)
3. Čulik K., Some notes on finite state languages and events represented by finite automa using labelled graphs, Čas. pro pěst. mat. 83, 43-55 (1961)
4. Woodger M., An Introduction to ALGOL 60, Comp. Jour. 3, 67-75 (1960).

PROGRAMMATION ET THEORIES DES CATEGORIES

J. Riguet

Paris, France

Il y a plus de quinze ans, en 1945, Eilenberg et Mac Lane publiaient un mémoire (1) dans lequel se trouvait introduit pour la première fois le concept de catégorie.

Depuis, le langage des catégories, qui avait déjà donné des preuves de son étonnant pouvoir explicateur en faisant remonter à leurs vraies sources des phénomènes de dualité (2) en algèbre et en topologie, avant lui demeures mysterieux, a non seulement conquis droit de cite dans ces disciplines, mais s'est revele a l'usage assez souple pour pouvoir être utilisé dans presque tous les domaines mathématiques. Et cela avec un tel succès, que je crois que l'on peut dire que dans 50 ans toutes les mathématiques auront été reformulées et refondues en termes de catégories.

Je voudrais montrer dans cet exposé que le langage des catégories peut être également utile pour formuler les problèmes relatifs a la programmation et se faire une idée exacte des rapports entre programme et marche réelle d'une machine.

INTRODUCTION A LA NOTION DE CATEGORIE

Une catégorie est, à très peu de choses près, un graphe orienté sur les flèches duquel on a défini une multiplication associative non partout définie. (A très peu de choses près car la definition d'une catégorie que nous donnons ci-dessous ne diffère de la véritable qu'en appelant "ensemble" des collections qui, pour pouvoir englober celles qui sont considérées dans les applications algébriques ou topologiques de la notion de catégorie, devraient seulement être appelées "classes". Mais pour les applications que nous avons en vue ici il est bien suffisant de considérer des "classes" réduites à des "ensembles".)

Tout le monde sait ce qu'est un graphe orienté. Qu'on nous permette d'en dessiner un

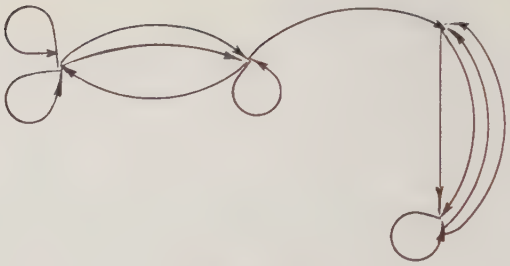


FIG. 1.

Un graphe (orienté) c'est un ensemble de sommets et un ensemble de flèches; à chaque flèche correspondant un sommet appelé origine et un sommet appelé extrémité, ces deux sommets pouvant éventuellement être confondus (flèches singulières).

Une catégorie n'est pas autre chose qu'un tel graphe sur lequel on a défini une multiplication pour les flèches x et y telles que l'extrémité de x coïncide avec l'origine de y, le produit des

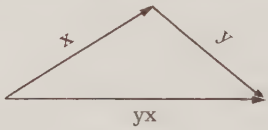


FIG. 2.

flèches x et y étant noté yx et cette multiplication étant associative dans le sens suivant: si on considère trois flèches x, y, z disposées comme le montre la Fig. 3, le produit z(yx) est toujours égal au produit (zy)x:

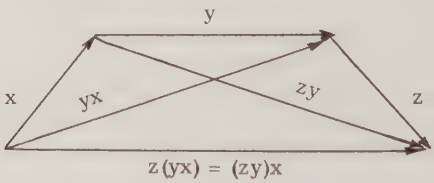


FIG. 3.

De plus, pour chaque flèche x, on supposera qu'il existe toujours une flèche singulière x′ dont l'origine et l'extrémité sont égales à l'origine de x et une flèche singulière x″ dont l'origine et l'extrémité sont identiques à l'extrémité de x, et qui sont des élé-

ments neutres à gauche, respectivement à droite, pour x, autrement dit, qui sont telles que

$x''x = xx' = x$

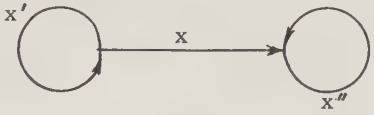


FIG. 4.

La notion de groupoïde de Brandt, introduite vers 1927 par Brandt comme outil algébrique pour ses recherches sur les formes quadratiques et qui, depuis les travaux de Ehresmann, joue un rôle important pour les notions de "structure locale" qui apparaissent en géométrie différentielle, n'est pas autre chose qu'un cas particulier de la notion de catégorie: c'est une catégorie pour laquelle toute flèche x admet un "inverse", autrement dit pour laquelle, pour toute flèche x existe une flèche y dont l'origine et l'extrémité coïncident respectivement avec l'extrémité et l'origine de x et telle que

$yx = x''$

$xy = x'$

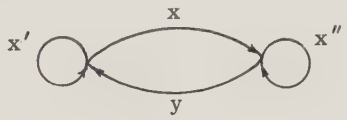


FIG. 5.

(on démontre facilement alors que y est unique et on le note x^{-1})

CATEGORIES DE PROBLEMES[1]

Il existe de très nombreux exemples de catégories. En fait toute structure mathématique donne naissance à un grand nombre de catégories qui fournissent un langage adéquat pour parler de telle ou telle de ses propriétés. Mais je voudrais me contenter de donner deux exemples.

Le premier est constitué par *la catégorie des applications des parties d'un ensemble X dans les parties de ce même ensemble X.* Elle se définit ainsi: Si X_1 et X_2 sont deux parties de X et si f est une application de X_1 dans X_2, on peut considérer f comme une flèche joignant le sommet X_1 au sommet X_2.

[1] Voir a ce sujet J. Riguet (9).

Considérant alors trois parties X_1, X_2, X_3 de X et deux applications f et g appliquant respectivement X_1 dans X_2 et X_2 dans X_3, on peut définir la flèche gf joignant X_1 à X_3 comme l'application composée des applications f et g, autrement dit comme l'application gf de X_1 dans X_3 telle que $\forall x \epsilon X_1$ $(gf)(x)_{def} = g(f(x))$.

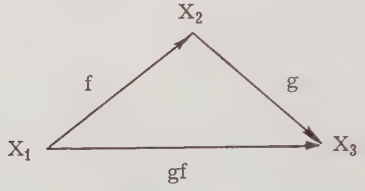


FIG. 6.

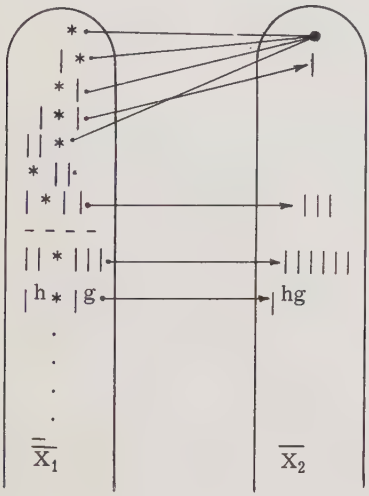
Il est alors facile de montrer que l'ensemble de telles applications (ou de telles flèches, comme on voudra) constitue bien une catégorie.

Le second exemple, qui, lui, est lié directement aux problèmes qui sont étudiés dans ce congrès et qui est apparenté au premier exemple est fourni par la catégorie des problèmes sur un alphabet S. Cette catégorie n'est pas autre chose que la catégorie des applications des parties de X dans les parties de X, X désignant cette fois l'ensemble des mots (de longueur finie) qu'on peut construire à partir de l'alphabet S.

Elle ne diffère donc guère du premier exemple choisi que par un choix particulier de l'ensemble X. Cependant elle mérite le nom particulier que nous lui avons attribué pour la raison suivante: Résoudre un problème (général) à l'aide d'une machine ou plutôt trouver une méthode de resolution d'une classe de problémes, c'est bien la même chose, au fond, que de trouver une application d'un sous-ensemble X_1 de X dans un sous-ensemble X_2 de X. X_1 constitue l'ensemble des diverses données possibles du problème, X_2 constitue l'ensemble des diverses solutions possibles du problème et une méthode de résolution du problème général posé consiste bien à associer au problème particulier spécifié par une donnée, c'est à dire par un élément particulier de X_1, la solution particulière c'est à dire l'élément particulier de X_2 qui lui correspond.

Pour illustrer ceci par un exemple sur lequel nous reviendrons, considérons l'alphabet S constitué par les quatre symboles: |, *, a, b et l'ensemble X des mots construits à partir de ces symboles. Le problème (général) de la multiplication des bâtons

consiste bien à trouver une application f de X_1 dans X_2, X_1 étant le sous-ensemble de X constitué par les mots de la forme $|^p * |^q$ (c'est à dire des mots constitués par p bâtons suivis de *, suivis de q bâtons) et X_2 étant le sous-ensemble de X constitué par les mots composés uniquement de bâtons, f étant l'application de X_1 dans X_2 qui à $|^p * |^q$ fait correspondre $f(|^p * |^q) = |^{pq}$.



Interprétation de la Résolution d'un Problème en Termes de Catégorie

Résoudre un problème à l'aide d'une machine peut se formuler en termes de catégorie de la manière suivante: Avoir à sa disposition une machine travaillant avec l'alphabet S c'est avoir à sa disposition une collection de problèmes autrement dit une collection de flèches $f_1, \ldots, f_m$ de la catégorie des problèmes. La résolution d'un problème f à l'aide de la machine revient donc à *trouver un chemin* (c'est à dire une suite de flèches consécutives dans la catégorie) uniquement constitué à partir de $f_1, \ldots, f_m$ et tel que le produit dans la catégorie de toutes les flèches du chemin est identique à f. La résolution d'un problème à l'aide d'une machine sur l'alphabet S revient donc à factoriser une flèche de la catégorie des problèmes sur l'alphabet S en "facteurs" qui appartiennent tous à un sous-en-

semble donné de la catégorie (à savoir le sous-ensemble des problèmes "élémentaires" résolus par la machine.)

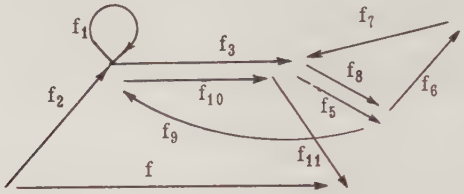


FIG. 8

Par exemple si la machine permet de résoudre 11 problèmes élémentaires $f_1, \ldots, f_{11}$ on a résolu le problème f à l'aide de la machine si on a trouvé le chemin $f_2, f_1, f_1, f_3, f_8, f_6, f_7, f_5, f_6, f_7, f_8, f_9, f_{10}, f_{11}$ tel que

$$f = f_{11}f_{10}f_9f_8f_7f_6f_5f_7f_6f_8f_3f_1f_1f_2$$

Comment, pratiquement, opère-t-on pour trouver une telle factorisation? On utilise pour cela des organigrammes.

ORGANIGRAMMES

Je pense qu'il est inutile que je détaille ici ce qu'on entend par organigramme: c'est une suite de boîtes

soit rectangulaires  (éléments opératoires)

soit en forme de losange  (éléments de décision)

reliant entre eux des sommets.

Chacune des boîtes rectangulaires contient un mot d'une certaine langue L; la boîte reliant les sommets i et j contenant le mot l_{ij}.

Chaque boîte en forme de losange contient un prédicat de la langue L suivi d'un point d'interrogation et une réponse à cette interrogation matérialisée par deux flèches partant de la boîte et portant chacune les réponses "oui" et "non".

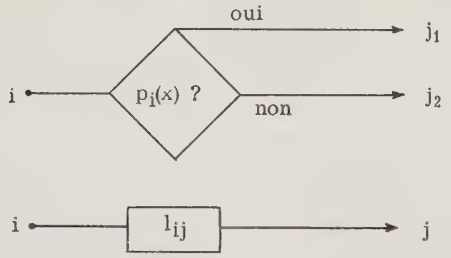


FIG. 9.

Il est facile de transposer en termes de graphes ce concept d'organigramme. Il suffit pour cela d'effacer les boîtes rectangulaires en conservant seulement la "valuation" l_{ij} de l'arête ij et de remplacer les boîtes en losange telles que celles representées à la Fig. 9 par deux flèches joignant les sommets i et j_1 et les sommets i et j_2 et "valuées" respectivement par $p_i(x)$ et $\overline{p_i(x)}$ ainsi qu'il est montré sur la Fig. 10.

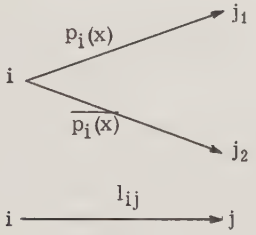


FIG. 10.

En généralisant très légèrement en considérant des aiguillages à n voies, le cas des organigrammes ordinaires correspondant à n = 2, on parvient à la définition suivante:

Etant donné un ensemble I (ensemble de sommets) dans lequel on a distingué deux éléments l_0 (entrée), l_1 (sortie) et une certaine langue L on appelle *organigramme* de support R valué par L un couple (R, l) constitué par une relation binaire $R \subset I \times I$ et une application l de R dans la langue L telle que

$i_1 \in \bar{R}(i_0)$, $\bar{R}^{-1}(i_0) = \phi$, $R(i_1) = \phi$

$\bar{R}$ étant la fermeture transitive de la relation R

$R(i) = \{j\} \rightarrow l_{ij}$ est une sentence de la langue L

$R(i) = \{j_1, \ldots, j_s\} \rightarrow l_{ij_1}, \ldots, l_{ij_s}$ sont des prédicats de la langue L mutuellement exclusifs et tels que l_{ij_1}, ou ..., ou l_{ij_s} est vrai.

D'une manière analogue, étant donné un ensemble de sommets I dans lequel on a distingué une entrée i_0 et une sortie i_0 et un ensemble X, on appelle *organigramme fonctionnel* de support R valué par Φ un couple (R, Φ) constitué par une relation binaire $R \subset I \times I$ et une application Φ de R dans l'ensemble des applications d'une partie de X dans une partie de X telle que

$i_1 \in \bar{R}(i_0)$, $\bar{R}^{-1}(i_0) = \phi$, $R(i_1) = \phi$

$R(i) = \{j\} \rightarrow \Phi_{ij}$ est une application d'une partie de X dans une partie de X

$R(i) = \{j_1, \ldots, j_s\} \rightarrow \exists X_{ij_1}, \ldots, X_{ij_s}$ partition de X telle que $\Phi_{ij_1} = \Delta_{X_{ij_1}}, \ldots, \Phi_{ij_s} = \Delta_{X_{ij_s}}$

ou Δ_X est l'application identique de X dans X.

Dans le cadre où nous nous plaçons on pourrait définir une *sémantique* comme le passage d'un organigramme écrit dans une langue L à un organigramme fonctionnel.*

Nous allons montrer comment un organigramme fonctionnel sur un ensemble X permet de construire une application d'un sous-ensemble de X dans un sous-ensemble de X, autrement dit de résoudre un certain problème lorsque X est l'ensemble des mots d'un alphabet S.

Considérons l'exemple suivant où $I = \{1, 2, 3, 4, 5\}$ 1 étant le sommet d'entrée et 5 le sommet de sortie, $X = \{a, b, c, d\}$

$R = \{(1,2), (2,3), (3,2), (2,4), (3,4), (3,5), (4,5)\}$

les "valuations" de R étant portées sur la Fig. 11.*
L'application d'un sous-ensemble de X dans un sous-ensemble de X définie par cet organigramme est alors $\begin{pmatrix} abd \\ dcd \end{pmatrix}$.

---

*L'ecriture $\begin{pmatrix} abc \\ cdb \end{pmatrix}$ désignant l'application qui à a fait correspondre c, à b fait correspondre d, à c fait correspondre b.

*La notion de foncteur d'une categorie dans une autre intervientrait ici. Eilenberg Mac Lane (1).

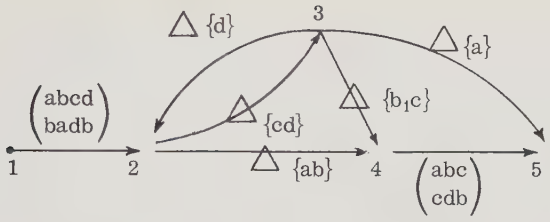


FIG. 11.

En effet a a pour image d puisqu'on a la séquence abbd qui correspond au chemin 1245

b a pour image c puisqu'on a la séquence baac qui correspond au chemin 1245

c n'a pas d'image puisque, arrivé en 2, on tourne indéfiniment sur le cycle 23

d a pour image d puisqu'on a la séquence dbbd qui correspond au chemin 1245

On voit donc par cet exemple que l'application définie par un organigramme fonctionnel n'est pas autre chose que le produit des flèches de l'unique chemin joignant l'entrée à la sortie si on considère l'organigramme comme plongé dans la catégorie des applications des sous-ensembles de X dans les sous-ensembles de X.

On dira qu'un organigramme fonctionnel est *admissible* lorsque l'application qu'il définit est non vide (c'est à dire est définie sur un sous-ensemble non vide).

Pratiquement les organigrammes que l'on utilise en calcul automatique ne sont pas quelconques: l'ensemble Φ des valuations de l'organigramme est fabriqué par une machine et la machine sait résoudre un problème dans la mesure où un certain organigramme sait le résoudre aussi. Il nous faut donc étudier quels sont les rapports entre organigrammes et machine. Ceci nous amène donc dans les paragraphes qui suivent à formuler de façon algébrique précise ce que nous entendons par machine. Mais auparavant nous allons préciser le rapport entre organigramme fonctionnel et la notion de diagramme.

LA NOTION DE DIAGRAMME

La notion d'organigramme fonctionnel que nous venons de définir n'est qu'un cas particulier de la notion de *diagramme* introduite par Grothendieck (3) dans le cadre de la théorie des catégories. Rappelons la définition d'un diagramme: Etant

donne une catégorie $\mathfrak{C}$ et un graphe (orienté) G on appelle diagramme (sur G, à valeurs dans $\mathfrak{C}$) une application D de l'ensemble des flèches de G dans l'ensemble des flèches de $\mathfrak{C}$, ainsi qu'un application (qu'on désigne aussi par D par abus de langage) des sommets de G dans l'ensemble des sommets de $\mathfrak{C}$ telle que si g_1 et g_2 sont deux flèches de G telles que l'extrémité j de g_1 coincide avec l'origine j de g_2, les flèches $D(g_1)$ et $D(g_2)$ de $\mathfrak{C}$ qui leur correspondent ont respectivement pour origines D(i) et D(j) et pour extrémité D(j) et D(k).

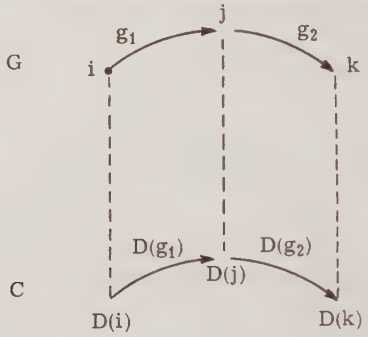


FIG. 12.

En résumé, on peut dire qu'un diagramme est un graphe G dont on a "valué" les flèches par les flèches de la catégorie $\mathfrak{C}$ et dont on a "valué" les sommets par les sommets de la catégorie $\mathfrak{C}$. Les relations binaires sur un ensemble I constituent un cas particulier de graphe: I sera l'ensemble des sommets et les couples $(i,j) \in R$ l'ensemble des arêtes.

La définition de diagramme prend alors l'aspect suivant: Etant donné une catégorie $\mathfrak{C}$ et une relation binaire $R \subset I \times I$ on appelle diagramme (sur R à valeurs dans $\mathfrak{C}$) une application D de l'ensemble des couples appartenant à R dans l'ensemble des flèches de $\mathfrak{C}$ ainsi qu'une application (qu'on désigne aussi par D par abus de langage) des éléments de I dans l'ensemble des sommets de $\mathfrak{C}$ telle que, quels que soient $(i,j) \in R$, la flèche D(i,j) a pour origine i et pour extrémité j. Si alors on considère un organigramme fonctionnel Φ, d'ensemble de sommets I, de support R, on voit immédiatement que Φ est un diagramme sur R à valeurs dans la catégorie des applications des parties de X dans les parties de X (on prend évidemment Φ_i = sous-ensemble sur lequel est défini l'application Φij).

DEFINITION ALGEBRIQUE DE LA NOTION DE MACHINE

La notion de machine, sans doute définie pour la première fois en termes algébriques par Ashby (3) (Voir aussi Riguet (7) pour la notion de couplage) et reformulée depuis par de nombreux auteurs (Voir en particulier Mealy(6)), peut être introduite de diverses façons plus ou moins équivalentes. Il est donc nécessaire que nous définissions de manière précise le concept de machine qui nous est nécessaire pour la suite.

Etant donné deux ensembles C et X nous appellerons *machine* une application Σ d'un sous-ensemble E de $C \times X$ dans E ayant la propriété suivante:

Si $(c, x) \in E$, $\Sigma(c, x)$ est de la forme $(c, \Sigma_c(x))$

où Σ_c désigne une application d'un sous-ensemble de X dans X.

On appellera E l'ensemble des *états* de la machine; un élément $c \in C$ sera appelé une valeur de la *variable de contrôle* ou encore *un paramètre* de la machine; un élément $x \in X$ sera appelé un *état de mémoire* de la machine. Dans ce qui suit, lorsque nous parlerons de machine, nous supposerons toujours que C contient un élément distingué c_0 tel que, quel que soit $x \in X$, $\Sigma(c_0,x) = (c_0,x)$ D'un point de vue intuitif, la condition sur Σ signifie que l'ensemble des états est partitionné en "plans" par les diverses valeurs possibles du paramètre, la machine effectuant la transition d'un état à un autre immédiatement suivant, en restant toujours dans ces "plans". Ainsi, par exemple, si l'on considère les ensembles $C = \{0,1,2\}$ où 0 est l'élément distingué et $X = \{a,b,c\}$, l'application Σ définie par

$\Sigma(1,c) = (1,b)$, $\Sigma(1,b) = (1,a)$, $\Sigma(1,a) = (1,a)$,
$\Sigma(2,c) = (2,b)$, $\Sigma(2,b) = (2,c)$ $\Sigma(0,a) = (0,a)$
$\Sigma(0,b) = (0,b)$, $\Sigma(0,c) = (0,c)$

est une machine. Les applications Σ_1, Σ_2 et Σ_0 sont définies par

$\Sigma_1(c) = b$, $\Sigma_1(b) = a$, $\Sigma_1(a) = a$ $\Sigma_2(b) = c$, $\Sigma_2(c) = b$,

Σ_0 = application identique.

Une représentation graphique d'une telle machine est donnée par la Fig. 13 (chaque flèche representant le passage d'un état à l'état immédiatement suivant).

Etant donné deux ensembles C et X nous appellerons *transfert de contrôle* une application σ d'un sous-ensemble E de $C \times X$

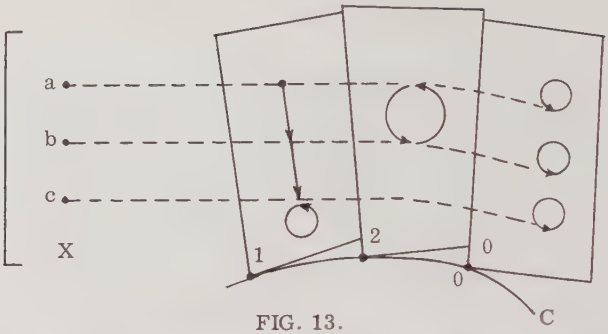


FIG. 13.

dans E ayant la propriété suivante: Si $(c,x)\epsilon E$, $\sigma(c,x)$ est de la forme $(\sigma_X(c), x)$ où σ_X désigne une application d'un sous-ensemble de C dans C. On dira que E est *l'ensemble des états controlés* par le transfert de contrôle σ ou que σ est un transfert de contrôle sur E. On remarquera que le concept de transfert de contrôle est en quelque sorte dual de celui de *machine*, qu'on aurait pu baptiser aussi bien "transfert d'états". Etant donné deux ensembles C et X, un couple (Σ,σ) constitué par une machine Σ avec ensemble d'états E et un transfert de controle σ sur ce même ensemble sera appelé une *machine avec transfert de contrôle*. On remarquera que cette définition coïncide à quelques détails près (importants pour le but que nous poursuivons) avec la notion de machine décrite par Mealy. Les machines (Σ,σ) que l'on rencontre dans la pratique présentent toujours pour le transfert d'instruction σ la particularité suivante: Etant donné une instruction $c\epsilon C$ il existe une partition de X en sous-ensembles disjoints $X_c^1, \ldots, X_c^S$ et s instructions $c_1, \ldots, c_S \epsilon C$ telles que, quel que soit $x\epsilon X$

$$\sigma_X(c) = \left| \begin{array}{l} c_1 \quad si \quad x\epsilon X_c^1 \\ - - - - - - - - - \\ - - - - - - - - - \\ c_S \quad si \quad x\epsilon X_c^S \end{array} \right.$$

Soit $\mathcal{O} = (R,\Phi)$ un organigramme fonctionnel d'ensemble de sommets I

On dira que $\mathcal{O}$ *est admissible pour* (Σ,σ) si l'ensemble des Φij, où R(i) est réduit à un élément, est contenu dans l'ensemble des transitions Σ_c de la machine et si l'ensemble des Φij où R(i) contient plus d'un élément est contenu dans l'ensemble des transferts d'instruction de la machine. En d'autres termes, on dira que $\mathcal{O}$ est admissible pour (Σ,σ) s'il existe une application

(unique) γ de I dans C telle que $\Phi ij = \Sigma_{\gamma(i)}$ lorsque R(i) est réduit à un élément et telle que $\Phi_{i,j_t} = \Delta_{X^t_{j(i)}}$ lorsque $R(i) = \{j_1, \ldots, j_s\}$ a plus d'un élément.

Si l'on considère seulement une machine Σ on dira que ω *est admissible pour* Σ s'il existe une application (unique) γ de I dans C telle que $\Phi ij = \Sigma_{\gamma(i)}$ lorsque R(i) est réduit à un élément.

L'application γ que nous venons de définir rend compte du rapport étroit qui existe entre un programme et la marche réelle de la machine: γ fait du graphe représentant la marche réelle de la machine un revêtement de l'organigramme exprimant le programme, le mot revêtement étant employé dans le sens "Uberlagerungsräume" des topologues [Voir (10) et (11)].

Pour illustrer les considérations précédentes nous allons considérer l'exemple que nous fournit l'algorithme de Markov α_s [Markov (12) p. 90] ou plutôt la machine qui lui est associee [Riguet (8)] et qui se définit de la manière suivante. L'ensemble des variables de mémoire X de la machine est constitué de tous les mots qu'on peut former à partir de l'alphabet

$S = \{|, *, a, b\}$

L'ensemble C des variables de commande de la machine est égal à $C = \{0, 1, 2, 3, 4, 5, 6, 7\}$. On désigne par X_m l'ensemble des éléments de X contenant comme sous-mot le mot $m \in X$. La machine Σ est alors décrite par les applications de parties de X dans X définies de la manière suivante

Σ_1 est l'application de $X_{b|}$ dans X consistuant à substituer le mot |b à la première occurence du mot b| dans chacun des éléments de $X_{b|}$. On dira que Σ_1 est une application de la règle $b| \rightarrow |b$

Σ_2 est l'application de X_{a1} dans X consistuant de même l'application de la règle $a| \rightarrow |ba$

Σ_3 est l'application de X_a dans X consistant en l'application de la règle $a \rightarrow$ (c'est à dire consistant a supprimer la première apparence de a dans les mots de X_a)

Σ_4 est l'application de $X_{|*}$ dans X consistant en l'application de la règle $|* \rightarrow *a$

Σ_5 est l'application de $X_{*|}$ dans X consistant en l'application de la règle $*| \rightarrow *$

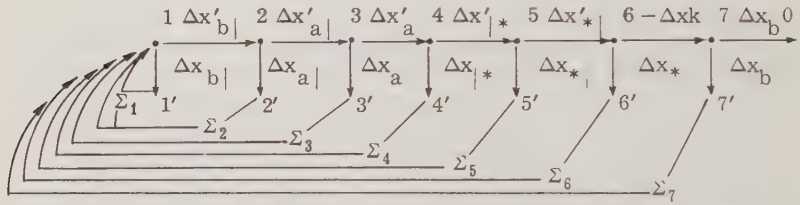
Σ_6 est l'application de X_* dans X consistant en l'application de la règle $* \rightarrow$
(c'est à dire consistant a supprimer la première apparence de * dans les mots de X_*)

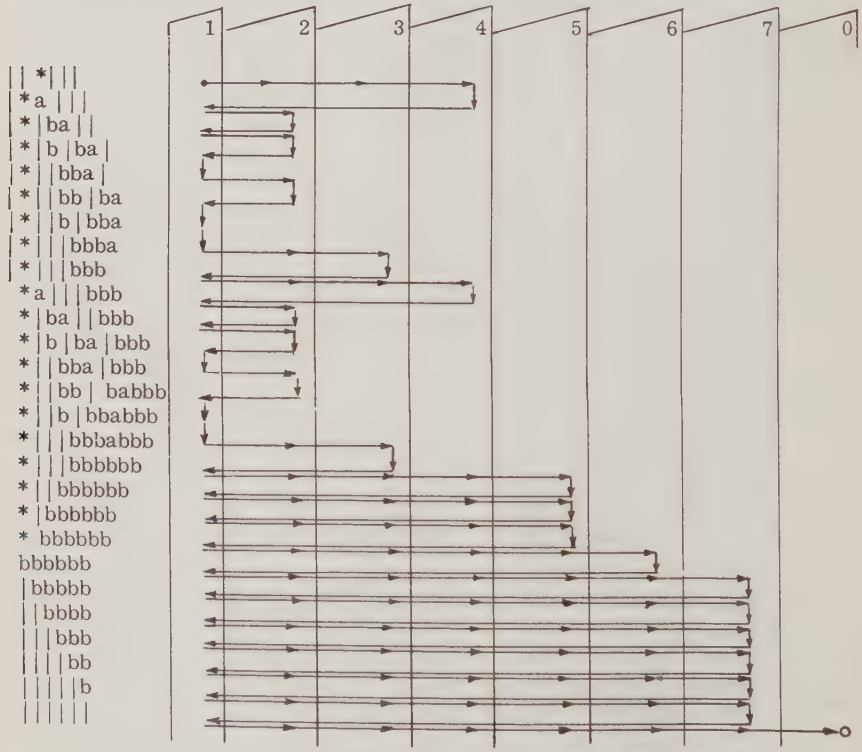
Σ_7 est l'application de X_b dans X consistant en l'application de la règle $b \rightarrow |$

Σ_0 est l'application identique de X dans X

L'organigramme ayant pour ensemble de sommets

$I = \{0,1,2,3,4,5,6,7,1',2',3',4',5',6',7'\}$





dessiné sur la partie supérieure de la figure 14 donne la programmation de la machine. L'application γ est définie par $\gamma(0) = 0 \quad \gamma(1) = \gamma(1') = 1 \ldots \gamma(7) = \gamma(7') = 7$. On voit clairement sur la figure que la marche réelle de la machine est un revêtement de l'organigramme. On constate que la machine a resolu le problème de la multiplication de || par |||. Nous laissons au lecteur le soin de vérifier que le problème général de la multiplication de $|^p$ par $|^q$ est bien résolu par l'organigramme proposé.

CONCLUSION

Il y a beaucoup d'autres aspects de la théorie des machines et de la programmation qui relèvent de la théorie des catégories. Grâce à la notion de couplage on peut constater que les machines forment une catégorie. Une notion de couplage des organigrammes, bien facile à imaginer puisqu'elle consiste simplement à les "mettre en série", permettrait de définir la catégorie des organigrammes. Ces deux catégories entretiennent des rapports simples qui s'exprimeraient aisément en termes de foncteurs.

Signalons enfin que l'on peut baser sur la théorie des catégories et sur les foncteurs constitués par les operateurs combinatoires de Schönfinkel-Curry [voir par ex. (13)] une théorie de la sémantique au sens défini plus haut c'est a dire une théorie de l'interprétation d'un organigramme écrit dans une langue L à un organigramme fonctionnel.

REFERENCES

1. Eilenberg Mac Lane, General theory of natural equivalences, Transactions of the american mathematical society, 1945.
2. Mac Lane, Categories and duality, Bulletin of the american mathematical society, 1952.
3. A. Grothendieck, Elements d'algèbre homologique et catégories abeliennes, Tohoku math. journ., 1947.
4. Ross W. Ashby, Design for a brain, Chapmann & Hall, London 1952.
5. Ross W. Ashby, Introduction to cybernetics, Chapmann & Hall Urley, London, New York 1956.
6. G. H. Mealy, A method for synthesizing sequential circuits, Bell system technical journal, **34** (1955) 1045-1079.
7. J. Riguet, Sur les rapports entre les concepts de machine de multipole et de structure algébrique, Comptes-rendus, Académie sciences, Paris, 237 - 425 - 427, 1953.

8. J. Riguet, Algorithmes de Markov et théorie des machines, Comptes-rendus, Académie sciences, Paris, 242 p 435 - 437, 1956.
9. J. Riguet, Le calcul des relations en tant qu'outil Méthodologique dans "La méthode dans les sciences modernes", Editions Travail et Industrie (Exposés réunis par F. Le Lionnais), Paris, 1957.
10. Reidemeister, Einfuhrung zur kombinatorischen Topologie, Chelsea pub. Company.
11. Seifert Threllfall, Lehrbuch der Topologie, Chelsea pub. Company.
12. A. Markov, Teoria algoritmii, Publications de l'institut Steklov, Moscou, 1952.
13. Curry Feys, Combinatory logic I, North Holland pub. Comp., 1961.

PANEL DISCUSSION

Metasyntactic and Metasemantic Languages

Moderator	:	M. WOODGER	(U. K.)
Members	:	C. BÖHM	(Italy)
		S. GORN	(U. S.)
		P. LANDIN	(U. K.)
		P. LUCAS	(Austria)
		M. PAUL	(Germany)
		C. STRACHEY	(U. K.)

WOODGER (U. K.) - We appreciate that the majority of the participants in this Congress neither speak English nor follow English well. Therefore we have to improve as far as we can the understanding by the audience of what is said by a panelist or a speaker from the floor. The objective is to achieve free discussions, and free discussions are generally difficult to follow. In order to assist in the language problems, I have some questions which have been put by some panelists who did not wish to put them themselves and perhaps I should start by putting these questions in the order in which I received them, and then getting discussion on each one.

The question I have from Mr. Paul (Germany) is the following: Is the time ripe for formalizing a metalanguage in which the syntactic and the semantic part of a language can be defined? His attitude to this question is as follows: that he who thinks there is a need for directly defining syntax and semantics of a formal language, does not see the necessity for formalizing already today a metalanguage to be used for this purpose. He regards the work which has been done recently in the field of the automatic treatment of such a metalanguage as nothing more than a general approach. And this is then related to the question of Mr. Lucas who indicated that one should distinguish the purposes of a metasemantic language. That one should distinguish at least one purpose: the description of a translation process. I would like to deal first with these points with panelists before going on to other matters.

GORN (U. S.) - I disagree with the first point made that it is wrong trying to begin to formalize metasyntactic languages, and I hope to give closer reasons later.

PAUL (Germany) - For the languages that have been dealt with up to now, the question of how the object language is defined has not played an important role with respect to the way translators have been constructed. What we can say today is that the class of languages to which our experiences up to now can be applied is defined by a set of productions, each production being a correspondence of an ordered set of syntactical variables to a syntactical variable. And for this class of languages, it is immaterial what notation is used for expressing this syntax and the semantic content, but it is essential to give a metasyntactical description of an object language in such a way that to each production a clear semantical meaning can be given.

WOODGER (U. K.) - I'd like to ask if there are any comments from the floor.

INGERMAN (U. S.) - I think that it seems that an appropriate reason for having a standardized specification language is that one cannot forget entirely the problem of designing languages. And when you are about designing a process in which the compilation can take place, the effect of this is, rather than standardizing languages, standardizing a process which is sufficiently general so that any language inside this awfully interesting class of languages can be then compiled with an "epsilon" additional effort. And it is for this reason that I think I would entirely disagree with what I understand your statement to be, namely that the time is not ripe.

STRACHEY (U. K.) - I should like to raise just one more point about this; and this is: that there is a very distinct difference between formalizing a language and standardizing it. And I think it is never too early to start talking of formalizing a language, by which I mean making absolutely precise what you mean by doing a language. I think it is almost never too soon to stop standardizing a language, which is saying that this is the only language people must use.

GARWICK (Norway) - May I stress before you the great importance of having this generalized processor because it permits us to change languages. I know situations where people say "for God's sake don't change angles so people have a chance fo finish their work." Of course it takes an enormous time to make changes by present days takings. Only when we have a generalized processor a change of angles can be effectuated in a week's time and we can have a complete modern language up to date any time. This of course is not only true as regards angles but much more in languages where there seems to be a new edition every year.

UNIDENTIFIED (U. K.) - I merely wanted to pose the question as to the extent to which the distinction between language and meta-language is a real one or an artificial one, and the extent to which a meta-language can be put between the frame of a language itself in the sense that according to some points of view (I think also Gorn's point of view), there is a continuous transition between language and meta-language. To what extent then is it necessary to make a rigid distinction between lan-

guages and meta-languages? I am not completely clear myself on this problem and possibly the panel have some views on this matter.

DIJKSTRA (Netherlands) - It is not a question but a remark. I am not completely convinced by the argument given by Garwick that we should have formalized meta-languages in order to be able to change our normal languages very soon. And he gives us the argument that the reason that people don't want to have angles changed is that it would be difficult to change their translators. But this argument does not convince me. The reason is that I am not attracted at all to make programs which are obsolete next year. In my opinion, the fact that people want languages to be more or less constant shows that the pressure on this tendency is not so much on the side of translator makers as on how to use them.

NAUR (Denmark) - I have one remark and one question.

I disagree with the statement made - I think by Mr. Paul - that metalanguages have had no importance for compiler construction or otherwise. I am quite convinced that if you have clearly defined languages, this would save you a tremendous lot of trouble. If metalanguages help you to define your languages, they do help you.

The second is a question. Who has made one of these famous systems which allows the compiling of a new language with an additional effort of "epsilon"?

WOODGER (U. K.) - I think I should make at this point a contribution of Mr. Böhm. We should separate the discussion in order to discuss the formal language, completely well defined as regards the context. He is using a system, the most important part of which is a reduction rule, and he says "our problem is to find some methods of mechanizing the reduction rule in a practical manner." We have no concern for the meaning as this is not *a priori* determined by the context, and moreover it may differ in every context.

STRACHEY (U. K.) - At least one point of view is that you should not try to separate the semantics from the syntax; you should try to integrate them more completely so that it is in fact impossible to make a statement which is syntactically correct but semantically meaningless. This is obviously a very desirable ideal.

GORN (U. S.) - I was very happy to hear the first part of the statement Mr. Strachey made. I also agree as to that one should not try to separate the syntax from the semantics, but I was disappointed to hear the second part because if the language is powerful enough, it must be able to make nonsensical statements in it, or you would not be able to get the complete power of all the sensible statements you want.

VAN WIJNGAARDEN (Netherlands) - Indeed, listening to discussions, I just object to what you are talking about. Of course a language should have no syntax whatsoever: it should only have semantics, and I do not understand all that you mean by nonsense. If a hypothetical machine acts on a certain text, then this apparently has been the meaning of this text, so I do not know what can be the idea of a nonsense.

STRACHEY (U. K.) - For nonsense program I mean one that makes the machine work indefinitely for example, and by nonsensical definition I mean one whose object does not exist. And my statement was, if you want a language powerful enough to define all the objects you want to exist, or to specify all the programs that you want to run, then we must allow the possibility of a language being misused. You cannot have the power without the responsibility of using a propellant.

DIJKSTRA (Netherlands) - I do not feel inclined to integrate syntax and semantics. I should like to abolish syntax completely. I mean, if you regard a language definition as the machine which carries out the process described when fed with a string of symbols in the language, and you get this kind of pragmatic semantic language definition, you don't need language syntax at all. A language definition should be a definition of what happens when you write down a string of symbols and present them as a program, and the reaction upon any string of symbols should be defined. If you look at a language definition in this way, you can regard syntax as completely superfluous, let us say, a corollary of minimum requirements which must be met by any text which does not present a presumably uninteresting reaction. Syntax is a minimum requirement because it is only a status for those properties of the text which can be derived lexicographically. But syntax, I do not think that it should have a defined function at all. This is my remark.

LANDIN (U. K.) - I wanted to say something which seemed obvious to me, but now I wonder whether it is after just hear-

ing Dijkstra's talk. One of the things that came out from the attempts to formalize the syntax of our languages, has been that these formal descriptions of a class of admissible strings are useful not only in order to vary the criterion of what is a correct program but also because they lead to a semantically useful partitioning and structuring of any particular text. Now, even if it should turn out that a particular syntactical description makes in fact any type of strings sensible, it might nevertheless be the case that some description of the universal set of strings will lead to a semantically useful structuring of any particular text and some won't. And the ones that will might conveniently be called syntactic descriptions of the language.

INGERMAN (U. S.) - I would like to suggest that one of the interesting features of taking advantage of the syntax of a language - I must restrict this now to languages whose syntaxes are reasonably well behaved, and I include ALGOL in such a class - is that there appears to be a "meaning" which is attachable to each syntactic rule, the meaning of which may be vacuous. By "meaning," I mean here specifically a sequence of instructions in some machines such that if one unravels the tree of rules that were used to write this string, and the tree is established by purely respecting the syntax, one ends up with a list of instructions which one can proceed to execute. Now, this may not be particularly deep, but I think it is certainly a justification for maintaining the existence of syntax rather than abolishing it entirely.

SAMELSON (Germany) - I'd like to turn Dijkstra's argument around and I feel that I will end up by taking Böhm's point of view. In describing a language we must give a set of rules telling us how from the initial characters we go up to words and sentences in the language, whereas it seems to me that the machine interpreting strings is somewhat like the British Museum procedure to verify admissible strings. We must have rules telling us how to develop admissible strings, and we could have a machine testing the admissibility of a string without taking any notice of the meaning. Semantics or interpretation would be simply another machine connected to the testing machine, but it could be left to anybody how to do interpretation and to construct any number of different semantic machines giving any interpretation desired. For example, I can identify the plus sign with addition, but I could also interpret it as meaning multiplication, and I don't see any reason why I should not be allowed to do it. This may not be very sensible but I think

that it shows that decomposition of strings according to syntactic rules and interpretation of the elements are quite distinct operations. Syntax gives the rules for composing correct strings and interpretation is something quite different.

DIJKSTRA (Netherlands) - I should restrict myself to comment on the last remark, that one where syntax was defended as giving you a rule deriving a grouped set of instructions from the structure of your text, but if I have adopted a program describing a process, I am completely disinterested in what instructions the translator makes out of it. I am completely disinterested in this two-stage execution. The only point is whether the process described is carried out correctly. This is my remark to Prof. Ingerman.

As to Prof. Samelson's remark I failed to appreciate it. I'm sorry.

WOODGER (U. K.) - The Moderator got some of Prof. Samelson's remarks. What I understood him to say at the end was that in the going through of the general execution of a program, which is checking the syntax of a piece of program, each time it gets to a terminal point in the checking for - shall we say - a variable, or it recognizes any other syntactic category, one may add the proper piece of instructions to do certain things, and one can put them quite arbitrarily, depending only on the meaning attached to that category determined by syntax recognition.

DIJKSTRA (Netherlands) - In response to that I should say that I do not see the virtue of two-stage language definitions, so that if I give you a syntax which tells you what is allowable to write down, and secondly I give you a description of what something means when it is written down, I think that the last thing is sufficient. As soon as my language definition, merely pragmatical, can tell me how to execute a process once its text is given, I don't see need for syntax any more. It may have an explanatory value, but not a defined function. This is another reason why I am not attracted by a so-called syntax oriented translator. In my opinion it is starting at the wrong end. If we write down a process description in a certain language, the thing that we want to happen is that the process which is described is carried out. The only thing we need for that purpose is the complete definition of how to react upon a description purely semantically.

WILKES (U. K.) - I think that Dijkstra may have said more or less what I wanted to say. I will then take up an earlier point.

It seems to me a misuse of words to ask "should we have

syntax or not''? A language has syntax, whether we like it or not. The question is how are we going to formalize the syntax and how shall we make use of that formalization. One, to determine whether or not what we have written is syntactically admissible, and two, to determine whether what we have written has any meaning. I believe there is much to be said to the view that only the second point is necessary, and that it carries with it as much of the syntactic analysis as it is necessary. And, if that is what is meant by saying that we should not have syntax, then I agree.

STRACHEY (U. K.) - What a language is, is what one expects from the language, is what it does, what the machine does. Now I think it is very important to distinguish between the language which is a description and the actual action of the machine which is an operation. And however far we go in describing a program of syntax, semantics, you can't get away too much from the fact that you have a machine and you give it commands, which is not the same thing as the language in which you are describing it. I think it is very important logically to distinguish between actually doing something or talking about it. And a great deal of people are bringing confusion between talking about things and doing them. And this is what we are doing now. We are talking about things and we are not doing them.

CARACCIOLO (Italy) - The point I should like to stress is that the distinction between semantics and syntax lies in my opinion in the problem of what one is doing with regard to equivalent syntactical descriptions of the same process. From the semantical point of view, syntax sometimes describes a way in which one gets a certain result; semantics is related with the result that we obtain, independently from the way we obtain it. Therefore, what I think is that syntax has to describe something, and of course it must be connected with a semantical description, a semantical interpretation of the description given by syntax, but the distinction is important in order to discuss the problem of having different syntactical descriptions of the same process - I mean the same from the semantical point of view.

INGERMAN (U. S.) - I may be now on the other side of the fence from where I was a moment ago but I'd like to say that there seems to be one area that has not been touched in this discussion, and I ask Dijkstra if he wants to comment upon it, if I might.

The first is the syntax of a language as it is defined in some metalanguages, for example the Backus notation. Second, there

is the semantics which can perhaps be functionally defined as saying "what there is that needs to be known in order to perform the process described by a statement." But I'd like to add that people are not perfect and errors are made. A person who does not know anything about the problem whose solution is being attempted can nevertheless take up a copy of the statement of the problem in some suitable language, and in fact correct it without damaging the logical structure of the problem. Now, I observe that this is, I think, what I understand of Dijkstra's meaning, a purely syntactic manipulation. And I would like to ask him if he would comment on that.

DIJKSTRA (Netherlands) - Did I understand right that you are referring to the case for instance where the operator inserts an additional closing bracket that the programmer has left out? An additional semicolon after the statement? You say that he can do this on purely syntactical ground?

This is - as far as I can see - not in contradiction with my remark that syntax should not have a defining function. The question is that I should like a language to be defined completely pragmatically as a way of defining the desired result. These machines which define a language have certain properties. You can derive its properties, but some of these properties can be conveniently expressed by syntax, and this may be a great help in checking whether a problem is at least obviously wrong. You can check this all right, and it may be a helpful means which acts statistically, but that is certainly not wrong. And all the time I regard the syntax as having no defining function at all, but only as a corollary written for some interesting purpose after a language has been defined.

Our main aim is of course to write down syntactically correct programs and then do things with them. I thought that the purpose of an algorithmic language is to describe algorithms, and of a machine to perform these algorithms.

JENSEN (Denmark) - I would like to ask one question. If you have a: a: in front of a statement, is it a syntactic error or a semantic error? In case it is a syntactic error, the full indication of this means that you have to consult semantics before you do what we now call the syntactic checking. Which means again that syntax can only be a help and not a basic concept.

VAN WIJNGAARDEN (Netherlands) - I will take up this point again. When I said that a language would have no syntax, I did not mean of course that it is forbidden to write a chapter on syntax. Because in one of the vaguest states of describing your

language, syntax may help first of all to give first elementary impressions about what may be the actions of the machine upon the given text. And, moreover, it is very useful to give in this chapter some examples in the way of syntax. You may say, for instance "if you give a sentence which is syntactically structured in this way, the effect will be this and that" as an example to the reader. And of course this is not forbidden in the least. And, here again, all these discussions and questions about the wrong program, they do not make any sense to me. If I write an extra semicolon, then obviously it was my intention to write an extra semicolon, and some action will take place. If the action is that the machine goes on for ever, then this was my intention. At some lectures I gave in America some time ago, I presented a full "semantic description" of a language. Very properly one of the listeners observed that if the programmer added at the end of its program one extra closing bracket, this would have the effect that the whole language description was destroyed in the computer. So what! This is not an error. Obviously, it was the intention of the programmer to destroy the whole semantical description of the language, otherwise he would not have written this extra closing parenthesis.

STRACHEY (U. K.) - I think that a very important point about syntactical checking is that it is unable to make sure that if there are redundancies in the language, which there are in all usable languages, including ALGOL, then the program I have, in fact, uses these redundancies correctly. A meaningless statement, in that sense, is one which does not obey the syntactical rules.

WOODGER (U. K.) - My list contains three questions that Gorn wishes to put to us.

GORN (U. S.) - Of the three points I wanted to raise, the first two of them are extremely specific and describe possible expansions to normal form to get a more powerful metasyntactic language. The third point is concerned with properties that I expect such metasyntactic languages will have to have in the future before we have control of as much as we would like to control.

For point number one, I think it has been remarked by several speakers in the afternoon, for instance Mr. Culik, that Backus normal form languages correspond to Chomsky phrase structure languages of type 2. He also remarked that if you restrict the use of concatenation once more, but have the same uses of the union of sub-languages called syntactic types, and

putting syntactic types together like this of course corresponds to concatentation...I think he remarked that if you restricted this use much more you would get to what is the equivalent to Kleene's finite state machine, corresponding exactly to Chomsky languages of type 3. I want to point out that there exists also an addition to the Backus normal form, which yields the more extensive phrase structure language of type 1.

The second point concerns what I called this morning "language operators." This permits us to get an infinite number of productions in a single symbol.

Now you see, the advantage in having this kind of notations is that if we expect to be able to get languages, metasyntactic languages, in which we can automatically construct translators, these might be the roots to follow, and certainly we could not do it with type 2 languages.

These are some of the specific suggestions.

Now for the very general things we will expect languages to have, even if we do not know how to do it now. First of all, it seems to me that we should go to multidimensional languages, for instance, the grass and trees, we talked about. Those are two dimensional languages; we can specify syntax with their help and we like to use them. For example, later on this year, I will be talking of how to take a Backus normal form specification for a language and write a complete graph that shows the whole language. I will show how to take such a complete graph and break it over into an infinite periodic tree. This is a very hard concept to get with strings representing these objects. It is easy when we visualize them multidimensionally, having infinite periodic trees with the specification of a language, we keep just using it in order to use the same prefix processes and we can start printing out the generation of all successive words coming out of this specification of a language, and then we have the generation of the words and how they derive from the syntactic rules as well.

The second point is that we want languages in which we can express some actions. A natural language is of this kind when it uses symbolism with words like "but." The words like "but" call for you not only to say "and" and look at what you had before the "but" and what you have after it, but also calls for some opinions with which to compare the two items.

Third point, we want the languages not to be strictly command languages and not to be strictly descriptive languages, with declared words like equals. We want them to be mixed, and mixed so that we can control them. And lastly, we want lan-

guages to have an unstratified control, and this means that if we want to be able to speak about the syntax, descriptive or command but syntax languages, we don't have to go on to another set of languages to do that, we don't have to have the meta-meta-meta-meta syntax operating descriptive and command of a language. There should be some means to get the semantic intent of syntax into the language itself, and have three levels which can be freely moved from one another, and properly all machines now have with their storage, operational units and control.

WOODGER (U. K.) - Thank you very much. This ends the session.

THEORY OF LANGUAGES– LIST PROCESSING AND PROCESSORS FOR SYMBOL MANIPULATION LANGUAGES

COMIT: A LANGUAGE FOR SYMBOL MANIPULATION

Carol Bosche

Research Laboratory of Electronics
Massachusetts Institute of Technology, Cambridge, USA

COMIT is a general-purpose symbol manipulation programming language which provides facility for pattern searches and transformations on symbolic elements held in lists. It has been programmed as a compiler-interpreter system by the mechanical translation group and the computation center of the Massachusetts Institute of Technology. The program has been running for a year on the IBM 709-7090 and has been distributed through SHARE.*

List structures provide flexible means of connecting symbols with each other. The position and word length restrictions given by the original structure of computer memory as, effectively, a table or matrix of bits are inconvenient in manipulating symbols. To evade these restrictions we might rely on complex data descriptions to indicate how information in a given relative position is to be understood; business-data processors, such as COBOL, take this approach. But if symbolic material to be transformed is of indeterminate character, particularly with respect to format or word and record length, it may be difficult to formulate such data descriptions. List processors, by restructuring the original table of bits, make it possible to manipulate information in more general form. Associated with each symbol is a pointer giving the location of a "next" symbol; rearrangements of this order are then accomplished by rearranging pointers rather than symbols, and searches for information, similarly, proceed down a line of pointers.

*Those who have worked on this project include J. Bennett, S, Best, C. Bosche, W. Cooper, M. Greber, K. Hansen, F. Helwig, M. Kannel, K. Knowlton, G. Matthews, A. Siegel, M. Weinstein, and V. Yngve.

This work was supported in part by the National Science Foundation, and in part by the U. S. Army Signal Corps, the Air Force Office of Scientific Research, and the Office of Naval Research.

The COMIT *workspace* is a string-form list structure, with subsidiary addressable lists, the *shelves*. In it, symbolic data are held. Perhaps because COMIT's notation was first designed to accommodate the needs of structural linguists in handling natural language, COMIT focuses attention on the manipulation of variable-length symbols in concatenation. The string operations of substitution, insertion, deletion, copying, and rearrangement may be performed on a series of *constituents*, thought of as elements separated by plus signs, in the workspace. Similarly, strings of constituents or single constituents may be transferred between the workspace and the numbered auxiliary shelves.

A list processor, however, may be constructed to emphasize other aspects of the situation of symbol manipulation. For example, LISP focuses attention on the specification of computable operations, recursions as well as iterations and linear programs. The LISP programmer may then give relatively less attention to the question of whether a given list structure or a copy of it is transformed in order to evaluate a given function. IPLV, when it does not require that the programmer's consciousness be directed to its mechanics of operation, keeps him aware of the possible complex directions of connection to be set up by ordered links between elements of information. Tree-structures, for example, are natural to IPLV as modes of organization of information.

It is possible to write recursions with several COMIT rules; so also it is possible to represent complex structures of information in COMIT.* The workspace constituent, in the first place, need not be an atomic unit, but may very well be a cluster of information, the parts of which naturally belong together. To the *symbol* may be attached optional subscripts; one *numerical subscript* and one or more *logical subscripts*. The logical subscripts may in turn have *values*, representing a lower level of classification. (Both these sets—values of a logical subscript and logical subscripts on a constituent—are in general taken to be unordered, unlike the constituents in the workspace, which are represented in a left-to-right ordering.) The subscripts on a constituent, then, can be used as pointers to provide different directions of connection from that of the main workspace string. They may also be used as cross-referencing or classificatory elements, or as convenient storage for subordinate information

*COMIT is equivalent to a general Turing machine.

associated with the constituent symbol. Furthermore, constituents themselves may be used similarly, in groups, to hold together still more complex units of structure. Finally, the shelves may be used, not only for storing material out of the way of current workspace operations, but also for further classificatory power. Since they are referred to by numbers, the shelves are addressable lists and can be used to set up matrices.

A COMIT *rule* in general contains a pattern search (*left-hand side*) as a means of locating those elements of the list on which operations are to be performed (in the *right-hand side* and *routing*). The left-hand side may specify, for instance, that the first occurrence in the workspace of a certain sequence of constituents is to be considered the operand of this rule. The most useful pattern search, however, is less definite in specification than the items which will be found as instances of the pattern. COMIT provides several degrees of definiteness in specification, through, for example, class-inclusion logic on subscripts, such that a workspace constituent will be said to match the left-hand side specification if it has *at least* the subscripts mentioned and *at least* those values specified in the rule. Constituents themselves may be left indeterminate, identified by their number or position; thus, the left-hand side may ask for the first three constituents before the occurrence of a specified symbol, in turn followed by a copy of the constituent found by that symbol. This pattern would be written*

```
$ 3 + SYMBOL + 2
```

By the latter device, specification through constituent number, in the order in which constituents occur in the rule, the search may refer indirectly to whatever was found by a previous constituent specification. Finally, a search may specify an indeterminate number of constituents by their left and right boundaries; these boundaries may extend to the limits of the workspace. Thus

```
, + $+ ,
```

searches for the first two commas and whatever is between them, and

```
$
```

*The $ sign was chosen to represent "something" or "anything" as the otherwise least useful character on the keyboard. All stand-

finds the whole workspace. Such a pattern search makes explicit the dependence of certain elements on context.

The left-hand side search may find workspace constituents which match its specification, or it may fail, in which case control is transferred to the next rule. This process of matching or failing may be compared to branching on a conditional statement or the fulfillment of a functional matrix.

The operations which may be performed on the elements found by such a search (as well as the search program itself) are provided by the COMIT system, which runs as an interpreter on the translated version of the program of rules produced by the two-pass COMIT compiler. This system design assumes that the operations provided by the interpreter are of sufficient generality and completeness that most would be required by a typical COMIT program; the programmer is thus saved the effort of defining them himself, as functions or macros. A COMIT program may, however, be divided into main routine and subroutines to define further higher-level general operations. The right-hand side and routing allow, among other things, the following operations: string manipulations; simple arithmetic on numerical subscripts; complement and approximate "and" logic with subscript values; operations on shelves which treat them as push-down lists or queue lines, so that constituents may be stored at either end of a shelf but removed from one end; rapid dictionary lookup in a list which is ordered at compile time; multiple branching, as, for example, on an end-of-file in a read operation; partially indirect description of the operation to be performed, for instance, storing a constituent on the shelf whose number is given by a numerical subscript; and determination of the flow of control.

The right-hand side is written after the left-hand side and an equals sign, which has the approximate meaning "is transformed into." For example,

```
$1 + SYMBOL*1 + $1/SUBSCRIPT VALUE =
  3 + 1/SUB*2 + M
```

would read, "Find the first constituent immediately before SYMBOL1 and a constituent which has the subscript SUBSCRIPT

ard characters, including those which have syntactic significance in a COMIT rule, may, however, be used in rules to match workspace data. The notation in a rule for a constituent +SYMBOL= would be *+SYMBOL*=.

with the value VALUE. Rearrange this string so that the third constituent mentioned ($1/SUBSCRIPT VALUE) is first, the first one ($1) is next and has added to it the subscript SUB with values as they are on the second constituent (SYMBOL1), and then the constituent M is inserted at the end." Since there is a right-hand side, and the second constituent found in the left-hand side (SYMBOL 1) has no place in it, the constituent is deleted.

The routing, which controls shelf operations, list look-up, input-output, and flow of control, then follows, indicated by slash bars. For example,

```
//*A 12 3, .G2000 RULE
```

added to the preceding rule would read, "Put all the contents of shelf 12 in place of the third constituent (M), continue if free storage is greater than 2000 registers, otherwise proceeding immediately to the following rule, and go to the rule named RULE." The *name* and *go-to* sections at the beginning and end of a rule are necessary only if control is to be transferred to a named rule through a go-to; if not, an asterisk may be substituted for either, and the flow of control follows the order of rules as they are presented.

Arrangements for the transfer of control in COMIT allow considerable communication between the rules and the constituents in the workspace on which they operate; such interrelation is particularly desirable for programs which simulate learning processes, since these latter require the activation of new rules whose formulation depends on input data. COMIT rules are not themselves lists and cannot be manipulated as data; the compiler puts them in table form. But since rule and subrule names, which govern the flow of control in COMIT, are in the same class as subscript names and their values, the subscripts on workspace constituents may be used to determine flow of control, either immediately, or by means of *dispatcher* settings, in a way which will determine which of several possibilities may be taken at a future branch point. Some of the indirect addressing possible in COMIT is useful for this purpose. For example, the dispatcher setting LANGUAGE FRENCH, written in the routing, will set a rule or rules named LANGUAGE so that the subrule FRENCH of each will be executed when control passes to the rule. If, instead, the second constituent should have the subscript LANGUAGE with value FRENCH, the routing instruction *D2 will add these to pre-

vious dispatcher settings and control the branching at the rule in the same way.

Since a large part of the program is fixed at compilation time, however, the introduction of certain novelties of rule structure would require re-compilation and therefore communication between several COMIT programs or a COMIT program and subordinate machine language programs. This method of organization contrasts with, on the one hand, the freedom of instruction modification available to the machine-language programmer and, on the other, the same freedom which might be introduced into a higher-level interpretive language which would recompile on signal, adding to the already-compiled program new symbols of the type to which its compiler is relevant. The obvious difficulties with these latter possibilities are, for machine-language programming, inconvenience; and for an interpreter which would allow recompilation, cumbersome system operation. Such difficulties may perhaps be avoided where program functions are held in list structure and relatively independent of each other, and where, as in LISP, a general defining function is available to the programmer.

In general, the choice of a level at which compilation stops and interpretation begins requires a delicate balance between, on the one hand, the convenience and speed of a compiled symbolic program, the parts of which are mutually interdependent, and, on the other hand, flexibility of operation. In COMIT, since the operations provided by the interpreter are themselves quite powerful, a method can be found of representing a rule-modification problem in another way.

Since there are several different "logical parts" to COMIT storage, information may be coded in several ways and programs of quite different types written to solve the same problem. Information may be held in constituents, order of constituents, subscripts, the dispatcher, the implicit flow of control, shelf order, and so on; for example, logical subscripts and their values are useful in representing trees, because of the logical operations which may be performed on sets of subscript values. Search specifications, also, may take account of the left-to-right ordering of the workspace and of the shelves. Thus COMIT programs can differ widely in effectiveness.

COMIT was written as a programming language with the intention that non-programmers might learn it easily. As a result has come a certain naturalness of expression, following the nat-

ural mode of combining symbols in a concatenated string, which makes it generally easy to organize, code, and check out a COMIT program.

AN INTRODUCTION TO THE KLS PROCESSING SYSTEM

by J. Weizenbaum

General Electric Company
Computer Laboratory
Sunnyvale, California, USA

This paper describes a list processing language called the KNOTTED LIST STRUCTURE system (KLS). In this day of the prolification of computer languages it is in order to say a few words relating to the motivation underlying the design and construction of yet another language. Most importantly, the design of KLS was undertaken not so much in an attempt to create a list language superior to existing languages, but rather as an effort to gain insight into the hardware software relationships list languages in general might bring to light. The fundamental question which ought ultimately to be answered by efforts of this nature relates therefore to efforts to design computer organizations which may be very effective vehicles for list processing. At the moment, of course, all list processing systems exist, as does the KLS system, merely as compiled macro systems or as interpretive systems, in other words, as programming systems imposed otherwise "orthodox" computers. The fact that there exist a class of problems which are currently being solved only by list processing systems of one form or another, even though these systems impose serious penalties in computer running time, certainly suggests that one day a computer may be designed which will be particularly effective for that class of problems or for subproblems of "ordinary" problems which fall into that class. The basic motivation mentioned seems therefore to be realistic.

What distinguishes list processing from "ordinary" processing? The main distinction lies in a redefinition of the successor relationship which is fundamental to the random access memory machines known today. In such machines a datum found in a particular memory cell, say cell *alpha*, has its successor stored in the "next" cell, namely *alpha + one*. Many of the operational characteristics of ordinary computers are derived from this single

fact. Furthermore, a vast inventory of programming techniques is built on it. In particular a premium is placed on the orderly placement of data within the core memory. A consequence of this is that data which in its natural environment has a complex structure (as opposed to, say, matrices which have a very orderly natural structure) must nevertheless be fitted into memory in a very "regular" way. The natural complexity of the data structure must therefore be reflected in the programs which manipulate these data. This gives rise to programs which are very complicated not so much because of what they must "do to" data, but because of the "decoding" they must do on the structural properties of the data themselves. List processing introduces a different successor relationship. In list processing each symbol is accompanied by an ancillary datum which serves as a *pointer* to its successor. This pointer is called a LINK. In addition, each symbol carries with it an identifying tag (called the SUFFIX) which, among other things, makes it possible for the symbol itself to be identified as a pointer to some other datum. List processing systems have been proposed in which each symbol carries with it a number of links, thus making possible elaborate cross referencing. The KLS system provides for only one like with each symbol. It is readily apparent that a link is simply an address of a cell in memory.

An objection frequently voiced by people when they are first exposed to a list processing system is that links waste memory space. The issue is really whether or not links are economic carriers of useful information. When data to be stored is naturally highly sequential (e.g. ordinary matrices with few null elements), links can contribute very little. But, where the important point about data is that they are interrelated in a complex way and, often even more important, that processing causes them to be reorganized dynamically, particularly where reorganization itself constitutes part of the solution of the problem, there links contain information which would otherwise have to be contained in program. Experience with list processing programs reveals this point to be of vital importance.

A LIST is a sequence of list cells linked to one another by link addresses. Every list has one member which has the property that the link address of no other cell in the system is pointing to it. This list cell is called the TOP of the list. Every list also has a terminal member, i.e. a list cell whose link address is not pointing to any other member of the list. This cell is called the

LIST TRAILER and has a distinct format reflecting its special function. The list cell whose link address is pointing to the trailer is called the BOTTOM of the list. The address of the trailer is the NAME of the list.

Names of lists are single symbols and are handled as such. This means that data structures (of arbitrary lengths and complexities) are capable of being treated as single symbols when appropriate.

When a list is transferred from core to bulk storage (e.g. drum) the trailer stays in core. It is marked such that its list can be retrieved from bulk storage whenever required. Although the body of the list is returned to an entirely new set of disjoint registers, the name of the list does not change.

A SIMPLE LIST consists of a sequence of list cells none of the symbols of which are names of lists and one of which is a trailer (Figure 2). An empty list is a simple list consisting of only a trailer (Figure 1). Any list may be a sublist of another list, i.e. its name may appear (with suitable suffix) on a list. Sublists may in turn have sublists and so on. A complex consisting of a list together with all its sublists, the sublists of its sublists, etc., is called a LIST STRUCTURE (Figure 3).

In the KLS system a list cell consists of three parts: the ID suffix, the function of which is to identify the function of the cell; the link portion L, which serves to determine the successor relationship of the cell to another; and the symbol portion S which stores symbols. In addition, there are three other types of cells with distinct formats. These are:

1. Two types of trailers

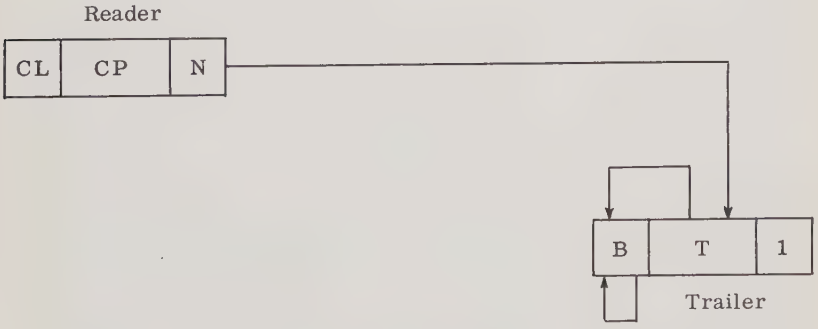


Fig. 1 An empty list.

a) Trailers for lists which have description lists
b) Trailers for lists which do not have description lists

2. Readers

The following table identifies each type of cell which may occur in the system:

TABLE I

Suffix Table

Suffix	Mnemonic	Type
0	R	Reader
1	DLIST	Description Trailer
2	LIST	Non-description trailer
3	AN	Alphanumeric
4	—	Machine detectable alphanumeric delimeter
5	DI	Decimal number
6	BI	Binary fixed point number
7	BFL	Floating point binary number
8	RO	Occurrence of a list name
9	RM	Mention of a list name
10	A	Address of list cell
11	—	Command
12	—	Address of reader

A trailer consists of four parts: the suffix P, the address of the top of its list (i.e. the list of which it is the trailer), the address of the bottom of its list, and a reference counter. The function of the last will be defined below. The address fields of the trailer of an empty list both contain the address of the trailer itself. The suffix of a trailer may have one of two values. One of these indicates that the list named by the trailer is non-empty and that the S portion of the top cell of the list is the name of a special list called the DESCRIPTION SUBLIST of the subject list. The other value of P signifies that the top cell of the list is an ordinary list cell, possibly the trailer itself.

There exists with the system a special kind of list called a DESCRIPTION LIST. What distinguishes this list from other

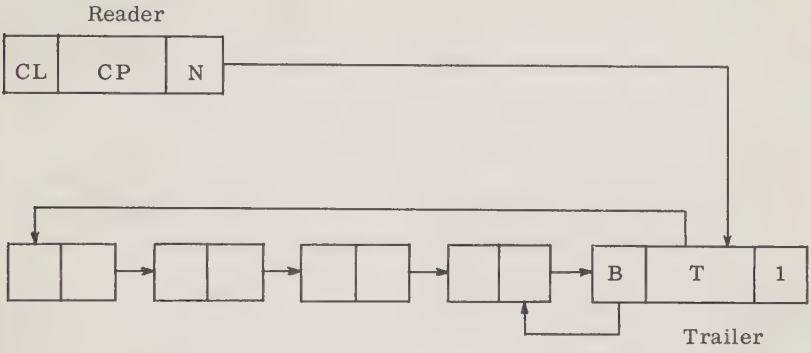


Fig. 2. A simple list.

lists is that it is thought of as having a special format. In fact, any list may be *treated* as a description list, i.e. may be subjected to operations which assume that list to have the description list format. The TOP cell of a non-empty description list is called an ATTRIBUTE, the next cell the VALUE of that attribute, the next cell another attribute, and so on. There are operations which, for example, will search such a list for an attribute and produce the value of that attribute as an output. That value may, of course, be the name of a list structure, a number, or any other kind of datum.

Certain lists are said to be DESCRIBED, i.e. to have a description list associated with them. A described list is one the TOP cell of which contains the name of a list which can be retrieved on the basis of commands which address themselves to certain description list processes (e.g. IDL: Input the name of the description list of the list referenced by the address portion of this command). Any reference to the TOP cell of such a list in non-description list context is a reference to the cell linked to that containing the name of the description list. The programmer need exercise no special precautions in this regard because the trailers of described lists are so marked that the system automatically handles any special problems which may arise in this connection.

Lists and list structure exist as entities in memory. Their content becomes available to other parts of the machine by a mechanism call list reading. There exists a piece of machinery

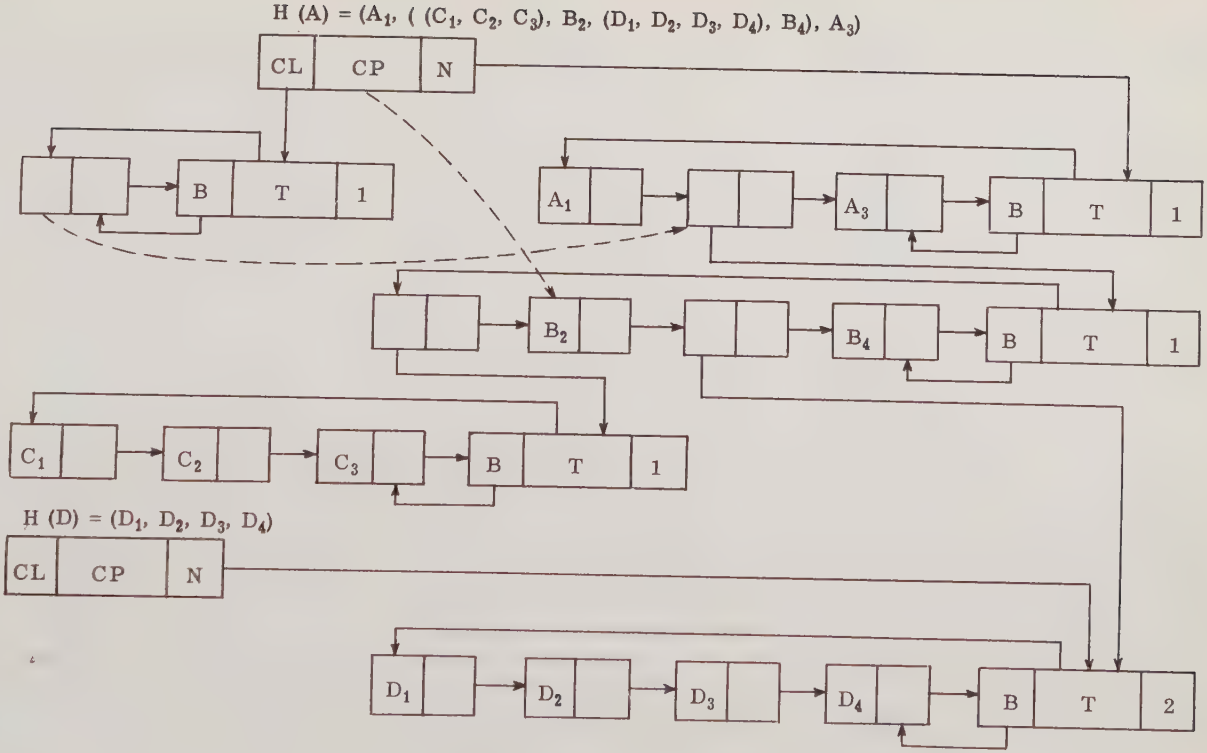


Fig. 3. A list structure.

called a READER. A reader is a list cell which contains the following pieces of information:

1. It suffix identifying it as a reader
2. An address called the CURRENT POINTER (CP)
3. An address field containing the address of the trailer for which this cell is currently serving as a reader (N)
4. An address field which, when not empty, contains the name of a list called the control list (CL) of that reader.

A reader is said to be empty or cleared when its CP, N and CL fields are zero. It is in its initial state when its CP field is the same as its N field and neither are empty. There is no restriction on the number of readers in which the name of a given list may appear.

There exist operations whose functions are to advance the CP within any given reader and to cause a certain datum to be deposited in working registers accessible to the programmer. These READ operations make possible the traversal of list structures. The general philosophy underlying reading is that the CP is to be advanced such that it points to one after the other of the list cells of a list structure. When the name of a sublist is encountered as an occurrence, that sublist should either be turned down on, or the main list should be pursued at the programmer's option. In any event, after one of these operations has been partially executed, it remains to be determined what datum is to be deposited in a working register. In general, this datum will consist either of the information pointed to by CP after the advance, or some information pointed to by that information, however indirectly. The content of a CP advance command is therefore:

1. An operation code signifying the advance CP operation *per se*, i.e. distinguishing it from other operations.
2. A "mode" indicator which can take either the value "linear" or "structure". The first of these indicates that any sublist encountered *is not* to be turned down on as a consequence of this operation, the second that any sublist so encountered *is* to be turned down on.
3. A "target" indicator which determines on what kind of symbol the CP is to stop. The CP continues to advance until either a symbol of the indicated type is found (within the constraints of the indicated mode) or the end of the list (in

the linear mode) or the end of the list structure (in the structure mode) is encountered. The types of targets which can be specified are:

a) "Word", the next symbol encountered is the target, whatever its type (other than trailer).

b) "Element", the next symbol which is not an occurrence of the name of a sublist (nor of a trailer) is the target.

c) "Name", the next symbol which is an occurrence of the name of a sublist is the target.

d) "Mention", the next symbol which is an occurrence of the mention of a list is the target.

e) "Occurrence", the next symbol which is an occurrence of the occurrence of a sublist name is the target.

f) "Element or Mention", the next symbol which is the occurrence of either a mention of a list name or an element is the target.

The distinctions among "NAME", "OCCURRENCE", and "MENTION" are the following: The NAME of a list, as has already been pointed out, is the address of its trailer. NAME is therefore a general term which serves as a token for the whole list (or list structure) named. If a list L-0 is to be a sublist of another list L-1, then the name of the list L-0 will appear on the list L-1 in the form of a symbol encoded (i.e. with an identifying suffix) as an OCCURRENCE. If, on the other hand, the list L-0 is simply a single datum (no matter how lengthy or complex) on, but not part of the structure of, the list L-1, then its name will appear on the list L-1 in the form of a symbol encoded as a MENTION. The operational significance of this will become clear when the READ instructions are described.

With the machinery so far described, it is possible to traverse a list up to the first encounter of a sublist, traverse that sublist until a sublist of it is encountered, turn down it, and so on until finally a trailer is reached. The mechanism for going back up the list structure is the control list. The control list (CL) whose name appears in the reader is a push down list. Whenever a sublist is turned down on, the address of the cell of the higher level list which led to the sublist is pushed down on CL. When the trailer of a sublist is reached, CL is popped up and the contents of the popped up cell placed in CP. The required advance is then

carried on from that point. Hence traversing list structures is accomplished with a minimum of attention from the programmer. The control list is, of course, outside the realm of accessibility of a programmer.

There must be essentially three ways of erasing a list structure; one is to erase or clear a reader of a list structure, another to delete the name of a sublist from a list, and finally to overwrite the name of a list structure. In any case, the intention is to erase the entire list structure to which the reference points, i.e. the list named as well as all sublist structure to which that list leads. (The special case where only the list starting at CP is intended for erasure is not discussed separately here.) The available space list has a TOP the address of which appears in a register called, say MT. A list is effectively erased, i.e. all its cells restored to the available space list, when the present contents of the MT register are made the link portion of the trailer and the address of the top cell of the erased list is placed in MT. Two difficulties are that (1) the name of the list to be erased may appear within the system in other contexts, e.g. in another reader, and (2) the erased list may have sublist structures itself, i.e. some of its symbols may be names of sublists which may need erasure. The second of these difficulties is taken care of when list cells are fetched from available space. It is at that time that an examination of the previous symbol status is made. If the list cell about to be used last contained the occurrence of a name of a sublist, then that sublist is erased.

The difficulty exemplified by the circumstance that a list may have several readers is remedied by the reference counter in the list trailer. This counter counts the number of references currently being made to that list, i.e. how many readers it has plus how many times its name appears anywhere within the system. When a list is to be erased, the reference counter is decreased by one. Only if the resulting count is zero is the list restored to available space as described.[1] Clearly, the time required to erase a list is independent of the length of the list.

In addition to the list, there is another device for storing information within the system. This is the STORAGE CELL, a term borrowed from IPL-V.[3] The term STACK is used equivalantly. Any cell in memory may be a storage cell. The format of such

a cell is exactly that of a list cell. The essential difference between a storage cell and a list is that a list has a trailer the address of which serves as the name of the list, whereas the storage cell is a list without a trailer, a list the name of which is the address of its top cell. This distinction has certain important consequences:

1. No reader may be appointed for a storage cell. Access to information on a storage cell is gained only via its top cell. This means that if the datum next to that contained in the top cell is to be retrieved, the top cell must first be vacated and the desired information placed in it. The operation which causes this sequence of events to take place is called "RSS", or RESTORE STACK.

2. The only way a datum may be placed on a storage cell is for that datum to overwrite whatever information is already contained in the top cell of the stack. Since, in many cases, it is desirable to preserve the datum in the top cell even though some new datum needs to be on the stack, an operation called "PRS", or PRESERVE STACK, is provided which causes the symbol at the top of the storage cell to be copied into a cell which is placed just below the top (i.e. to which the top cell is then linked).

One important application of the storage cell concept is the accumulator of the KLS system. The command format and philosophy of the system is essentially that of a single address computer. This implies that, in general, every command contains an operation code which as usual, specifies what the command is to do to some data, and an address portion which designates the operand on which the specified operation is to be performed. Many operations are, however, binary in the sense that they require two operands (but usually yield only a single result). One of the two operands may be pointed to (however indirectly) by the address portion of the command. The other operand is generally understood to be in the accumulator. In certain cases the accumulator contains an address or the name of a list which, by a chain of indirection the length of which is not explicitly given, leads to the desired operand. In the KLS system this accumulator is called "WO" and is a storage cell. The operations "PRS" and "RSS" therefore apply to the accumulator. One implication of this fact is that, under suitable circumstances, intermediate results need not be stored in especially selected storage cells but

may be merely preserved in WO for later recall by way of the RSS operation.

Commands themselves are symbols in list cells. A program is therefore a list or list structure. The functions usually assigned to the *command counter* are carried out by a reader which is designated as the CURRENT INSTRUCTION READER. Fetching the next command from storage is thus simply a READ operation. A consequence of this approach is that the control list (CL) which is part of the reader serves as a powerful subroutine linkage device. There exists a command "VST" (VISIT) which is an unconditional transfer of control (i.e. a branch in the program) to the address specified. The effect of this operation is, however, not merely to transfer control but to store the address of the VST command itself on the top of the control list of the current instruction reader. When the command "TERM" (TERMINATE) is encountered as part of a program, the control list on the current instruction reader is "popped up", i.e. its top element deleted, and the link address of the command pointed to by the popped up datum is placed in the current pointer portion of the current instruction reader. The effect of this is, of course, to resume the computation at the point after the VST command upon subroutine termination. Because of the fact that the control list is a list, visits and termination may be nested in any arbitrary way. This helps make the programming of the computation of recursive functions quite easy.

Within the KLS system indirect addressing of operands is the rule rather than the exception. (This is also the case in the IPL-V system.) In most cases the programmer need hardly be aware of the fact that he is arriving at his operands indirectly. Commands which require a particular kind of operand, e.g. the READ operations require a reader, will begin with the address given in the command, examine the contents of the location specified, and, if the information stored there is not of the type required, will follow the chain of indirection (if possible) until the proper datum is located. If the chain of indirection so initiated does not terminate successfully, then an error is reported to the programmer by means of an appropriate remark sent to an output device of the computer.

Controlled indirection can also be achieved at the option of the programmer. Associated with certain commands is a designation

code called the "Q CODE". For those commands for which Q is meaningful, the following table defines Q:

Q CODE TABLE

Q	Meaning
1	The content of the address given in the command is to be considered as if it had been the address portion of the command and the Q CODE had been absent. (One level address substitution).
2	This Q CODE applies only to commands the address portion of which leads (by any chain of indirection) to a datum which is either a number or an alphanumeric symbol. The Q = 2 code then signifies that the address of that datum is the desired operand.
3	The same conditions on commands as applicable to Q = 2 apply in this instance. The desired operand is the numeric or alphanumeric datum at the end of the chain of indirection.

SUMMARY:

The following section gives an abbreviated description of certain important commands which form a basis of the KLS system. This short outline should give the reader some general feel for the system. The section after that gives a sketchy outline of a small problem which was actually run within the system. The interested reader will find it full of opportunities for exercises. But even after having labored through all the text here presented, the reader is entitled to ask what the essential contribution of this system is thought to be.

The fundamental motivation of this work arose out of questions of machine organizations. The experience gained in designing and using this system has indeed revealed a number of points of great interest in that context. Unfortunately, discussion of hardware issues must be beyond the scope of this presentation. But, apart from gains in that direction, it may be claimed that the KLS system has contributions to make. One of these is that KLS is easy and "natural" to use. Of course, the use of the word "natural" here is undoubtedly related to the fact that the single address concept has become a habit. Nonetheless, the ease with which a small sample of programmers have become enthusiastic

converts to this system is remarkable. Another contribution which will lead to further development and is therefore perhaps more important, relates to the so-called "responsibility issue" The question arises as to what program has the responsibility for finally erasing a list which may exist in different program contexts. When the last appearance of a list name within the KLS system is overwritten or otherwise destroyed, the list associated with that name is erased. Because of the way lists are finally restored to available space, it turns out that nothing is ever erased if the subject program does not run through available space at least once. But the main thing is that a large part of the responsibility (for list erasure) burden has been taken over by the system itself. All other advantages which normally accrue to list processing have been kept. Threaded lists[4] are a subcase of knotted lists, KLS being more general at least in the sense that a given sublist may be a sublist of many lists. The way to see a relationship between KLS and T-Lists is to think of the KLS *READER* as the T-List *ALIAS*.

INSTRUCTIONS

This paper is not an appropriate place to give a complete programming manual. The following presents a list of almost all instructions of the KLS system and a detailed description of most instructions involving readers. It is believed that many of the unexplained instructions will be obvious at least to readers already familiar with other list processing systems.

INSTRUCTION LIST

-Arithmetic-

ADD	ADD
DVD	DIVIDE
MPY	MULTIPLY
NTL	NEGATIVE TALLY
SUB	SUBTRACT
TAL	TALLY

-Branch-

BCF	BRANCH ON CONTROL FF
BDF	BRANCH ON DECISION FF
BSF	BRANCH ON SEARCH FF

BTO-9	BRANCH ON CONSOLE TOGGLE 0-9
BTF	BRANCH ON TEST FF
BU	BRANCH UNCONDITIONALLY
BWA	BRANCH IF WO CONTAINS AN ADDRESS
BWAN	BRANCH IF WO CONTAINS AN ALPHANUMERIC
BWB	BRANCH IF WO CONTAINS A BINARY INTEGER
BWC	BRANCH IF WO CONTAINS A COMMAND
BWD	BRANCH IF WO CONTAINS A DECIMAL INTEGER
BWE	BRANCH IF WO CONTAINS AN ELEMENT
BWM	BRANCH IF WO CONTAINS A MENTION
BWN	BRANCH IF WO CONTAINS A NAME
BWO	BRANCH IF WO CONTAINS AN OCCURRENCE
BXL	BRANCH ON X FLIP FLOP LINEAR
BXS	BRANCH ON X FLIP FLOP STRUCTURE
TERM	TERMINATE
VST	VISIT

-Control (Machine Manipulation)-

ASC	ASCEND
ASG	ASSIGN
CLER	CLEAR
CXF	CHANGE X FLIP FLOP
PAUSE	PAUSE
RSRL	RESET READER LINEAR
RSRS	RESET READER STRUCTURE
RVO	REVERSE ONE LEVEL
RVT	REVERSE TO TOP
SDF	SET DECISION FLIP FLOP
SL	SHIFT LEFT
SLX	SHIFT LEFT INTO X
SR	SHIFT RIGHT
STOP	STOP PROGRAM
SXL	SET X LINEAR
SXS	SET X STRUCTURE
XEQ	EXECUTE

-Create-

CES	CREATE EMPTY STORAGE CELL
CRN	CREATE NAME

-Description List-

AVA	APPEND VALUE AND ATTRIBUTE
EDN	ERASE DESCRIPTION LIST NAME
EVA	ERASE VALUE AND ATTRIBUTE
FVA	FIND VALUE OF ATTRIBUTE
IDL	INPUT DESCRIPTION LIST
PDL	PUSH DOWN DESCRIPTION LIST
RVA	REPLACE VALUE OF ATTRIBUTE

-Erase-

ERL	ERASE LIST
ERN	ERASE NAME
ERR	ERASE READER
ERS	ERASE STACK

-Data Into Machine-

IC1	INPUT CARDS MODE 1
IC2	INPUT CARDS MODE 2
IF1	INPUT FROM CONSOLE TYPEWRITER MODE 1
IF2	INPUT FLEXOWRITER MODE 2

-List Manipulation-

NUL	NULIFY
PRL	PRESERVE LIST
RLD	RESTORE LIST TO DELIMITER
RSL	RESTORE LIST
XCL	EXCHANGE ON LIST

-Command Requires Name-

COL	COPY LINEAR
INN	INPUT NAME
LOC	LOCATE
MNS	MAKE NAME SAFE

-Data Out of Machine-

OCF	OUTPUT CELL TO CONSOLE TYPEWRITER
OCP	OUTPUT CELL TO PRINTER
OLF	OUTPUT LIST TO CONSOLE TYPEWRITER
OLP	OUTPUT LIST TO PRINTER
OSF	OUTPUT STRUCTURE TO FLEXOWRITER
OSP	OUTPUT STRUCTURE TO PRINTER

-Command Requires a Reader-

APR	APPOINT READER
COP	COPY READER
ICP	INPUT CURRENT POINTER
INR	INPUT READER
LCL	LOCATE INEARLY
LCS	LOCATE STRUCTURALLY
SCP	STORE CURRENT POINTER

-Stack Manipulation-

PRS	PRESERVE STACK
RSD	RESTORE STACK TO DELIMITER
RSS	RESTORE STACK
XCS	EXCHANGE ON STACK

-Transfer Stack to WO-

INP	INPUT
IWR	INPUT WO AND RESTORE

-Transfer to WO-

IND	INPUT DELIMITER
INS	INPUT SYMBOL

-Transfer List to WO-

DLE	DELETE LINEAR ELEMENT
DLN	DELETE LINEAR NAME
DLW	DELETE LINEAR WORD
NLR	NULL READ
RLW	READ LINEAR WORD
RLE	READ LINEAR ELEMENT
RLN	READ LINEAR NAME
RLM	READ LINEAR MENTION
RLO	READ LINEAR OCCURRENCE
RLEM	READ LINEAR ELEMENT OR MENTION
RSW	READ STRUCTURE WORD
RSE	READ STRUCTURE ELEMENT
RSN	READ STRUCTURE NAME
RSM	READ STRUCTURE MENTION
RSO	READ STRUCTURE OCCURRENCE
RSEM	READ STRUCTURE ELEMENT OR MENTION
RXW	READ X WORD

RXE READ X ELEMENT
RXN READ X NAME
RXM READ X MENTION
RXO READ X OCCURRENCE
RXEM READ X ELEMENT OR MENTION
RLF READ LINEAR FIND
RSF READ STRUCTURE FIND

-Test-

TCE TEST CONTROL LIST EMPTY
TEQ TEST EQUAL
TGE TEST GREATER THAN OR EQUAL
TGR TEST GREATER THAN
TID TEST IDENTICAL
TLE TEST LESS THAN OR EQUAL
TLEW TEST LIST LENGTH EQUAL
TLGW TEST LIST LENGTH GREATER THAN
TLLI TEST LIST LENGTH EQUAL ONE
TLLO TEST LIST LENGTH EMPTY
TLS TEST LESS THAN
TMI TEST FOR MINUS SIGN
TNE TEST NOT EQUAL
TNS TEST NATURE OF SYMBOL
TUN TEST FOR UNITY
TZR TEST FOR ZERO

-Transfer WO to List (Name)-

PL0 PUSH ON TOP, LIST, RESPONSIBLE OCCURRENCE
PL2 PUSH ON TOP, LIST, RESPONSIBLE MENTION
PL4 PUSH ON TOP, LIST, LEAVE SYMBOL CODED AS IS
PL5 PUSH ON BOTTOM, LIST, RESPONSIBLE OCCURRENCE
PL7 PUSH ON BOTTOM, LIST, RESPONSIBLE MENTION
PL9 PUSH ON BOTTOM, LIST, LEAVE SYMBOL CODED AS IS

-Transfer WO to List (Reader)-

IN0 INSERT BEFORE, RESPONSIBLE OCCURRENCE
IN2 INSERT BEFORE, RESPONSIBLE MENTION
IN4 INSERT BEFORE LEAVE SYMBOL CODED AS IS
IN5 INSERT AFTER, RESPONSIBLE OCCURRENCE
IN7 INSERT AFTER, RESPONSIBLE MENTION
IN9 INSERT AFTER, LEAVE SYMBOL CODED AS IS

-Transfer WO to Stack-

PS0	PUSH DOWN ON TOP OF STACK
STL	STORE ON LIST
STS	STORE ON STACK
AND	LOGICAL AND
OR	LOGICAL OR
SBIT	SET BIT (TO ONE)
RBIT	RESET BIT (TO ZERO)
CBIT	CHANGE BIT
TBIT	TEST BIT

-Monitor Control-

ETM	ENTER TRACING MODE
LTM	LEAVE TRACING MODE

Read Instructions

R (L/S/X) (W/E/N/M/O/EM)
R = Read
(L/S/X) = Linear, Structure or X Mode
W = target of read is Word
E = target of read is Element
N = target of read is Name
M = target of read is Mention
O = target of read is Occurrence
EM = target of read is Element or Mention

Linear Mode e.g. RLN A

"A" must lead to a reader. The reader's current pointer is advanced linearly along the list into which it is currently pointing. After the CP is advanced once, the symbol in the cell to which the CP is pointing is tested for the target criterion given by the instruction, in the case of a RLN instruction, "Name". If the symbol is the target of the instruction, it plus its ID is copied into WO. If the symbol is not the target, the CP is advanced again, etc.

If, after the CP is advanced, the cell tested is found to be a trailer, the previous

CP is left in the reader, the control flip flop is set, and WO is left unchanged.

Description list names cannot be the target for any of the read instructions.

Structure Mode **e.g. RSN A**

"A" must lead to a reader. Before the current pointer is advanced, the symbol currently pointed to is tested for the occurrence (O) of a Name. If the cell is an O, the CP is pushed down on the reader's control list (CL) and the O replaces the CP in the reader. (The CP is now pointing one level deeper into the list.)

The read mode now is identical to the linear mode with two exceptions:

1) If another O is encountered in the list before the target is reached, the CP is again pushed down on the CL and the reader's CP replaced by the O.
2) If a trailer is encountered before the target is reached, the CL is popped up and the top CP used to replace the reader's CP. (This results in an exit from a sublist into the next higher list.) If the CL is found empty when a trailer is encountered, the CP is left pointing to the cell just ahead of the trailer, the control flip flop is set, and WO is left unchanged.

X Mode **e.g. RXN A**

If when a Read X instruction is executed, the X flip flop is in the linear state, the read instruction is executed in the linear mode. If X is in the structure state, the read instruction will be executed in the structure mode.

DLE Delete Linear Element **e.g. DLE A**

DLN Delete Linear Name **e.g. DLN A**

DLW Delete Linear Word e.g. DLW A

These three commands are identical to their corresponding Read Linear counterparts with the following exception. If the current pointer in the reader for the list being read is pointing to a symbol on the list and not the trailer, the symbol is removed from the list either by deleting or nullifying the cell containing the symbol. The cell is nullified only if it is the bottom cell on the list.

NLR Null Read e.g. NLR A

"A" must lead to a reader. The symbol pointed to by the current pointer is copied into WO unless the CP is pointing to a trailer. If the CP is pointing to a trailer, the control flip flop is set, otherwise it is reset.

NUL Nullify e.g. NUL A

"A" must lead to a reader. The cell pointed to by the current pointer is made null unless the cell contains a trailer.

RLF Read Linear Find e.g. RLF A

"A" must lead to a reader. The current pointer is advanced linearly along the list, into which it is currently pointing, and each symbol encountered is compared with the symbol in WO_0 using WO_1 (the second cell on the WO stack) as a mask. For every "1" bit in WO_1, the corresponding bit positions in the symbols from the list are compared with the corresponding bit positions of the symbol in WO.

If a symbol is found on the list which results in an equal comparison, the symbol in WO_0 and the mask from WO_1 are popped off the WO stack and a copy of the equal symbol pushed down on WO. The CP is left pointing to the symbol on the list and the search flip flop is set.

If no symbol is found on the list which results in an equal comparison, the symbol from WO_0 is popped up, leaving the mask in WO. The CP is left pointing to the bottom symbol on the list and the search flip flop is reset.

RSF — Read Structure Find — e.g. RSF A

The Read Structure Find differs from the Read Linear Find only in the sense that the CP is advanced structurally instead of linearly. The final results are the same for both instructions.

APR — Appoint Reader — e.g. APR A

WO must lead the name of a list. Q(A) becomes a reset reader with an empty control list for the list named.

COR — Copy Reader — e.g. COR A

"A" must lead to a reader. A copy of this reader is made in a cell from available space. The new cell address is placed in WO with an address ID.

ERR — Erase Reader — e.g. ERR A

"A" must lead to a reader. The reader is returned to available space and the reference counter in the list pointed to by the reader is decremented. If the reference counter goes to zero, the entire list, including trailer, is returned to available space.

TCE — Test Control List Empty — e.g. TCE A

"A" must lead to a reader. The control list from this reader is tested for empty. If the CL is empty, the test flip flop is set, otherwise, it is reset.

LCL — Locate Linearly — e.g. LCL A

"A" must lead to a reader. Each symbol, including its ID, on the reader's main list is compared with the symbol and ID in WO. If the symbol, including its ID, is

identical to the symbol and ID in WO, the current pointer in the reader is set to the point to the symbol and the search flip flop is set. If no such symbol is found, the reader's current pointer is not changed and the search flip flop is reset.

LCS — Locate Structurally — e.g. LCS A

Locate Structurally is identical to Locate Linearly with the exception that the entire list structure pointed to by the reader, found through "A", is searched for a symbol and ID identical to those in WO.

RSRL — Reset Reader Linear — e.g. RSRL A

The current pointer in the reader described by "A" is set to point to the trailer of the list into which it is currently pointing.

RSRS — Reset Reader Structure — e.g. RSRS A

The control list of the reader described by "A" is popped up until empty and the reader's current pointer is set to point to the trailer of the main list.

RVO — Reverse One Level — e.g. RVO A

The control list of the reader described by "A" is popped up and the current pointer from the cell popped off replaces the current pointer in the reader. The control flip flop is reset.

If the control list can not be popped up, the control list being already empty, the control flip flop is set.

RVT — Reverse to Top — e.g. RVT A

The control list of the reader described by "A" is popped up until empty. The last current pointer popped off the control list replaces the current pointer in the reader.

If the control list can not be popped up,

the control list being already empty, the reader's current pointer is left unchanged.

PDL — Push Down Description List — e.g. PDL A

"A" must lead to the name of a non-describable list. WO must contain a name. The name in WO with an identity that of a responsible occurrence is pushed down on the list. The list's trailer is made describable.

If "A" leads to a describable list or WO is not a name, the appropriate statement is printed out and the program continued.

IDL — Input Description List — e.g. IDL A

"A" must lead to a describable list. The name of the description list with an identity that of a responsible occurrence is copied into WO.

If the list is not describable or the top cell is not a name, the appropriate statement is printed out and the program continued.

AVA — Append Value and Attribute — e.g. AVA A

"A" must lead to a description list. The "attribute" in WO_0 and the "Value" in WO_1 are appended to the description list of the describable list.

WO_0 and WO_1 are unchanged.

FVA — Find Value of Attribute — e.g. FVA A

"A" must lead to a description list which is searched for an attribute identical to the attribute in WO.

If the attribute in WO is found on the description list, the value is copied into WO and the search flip flop set.

If the attribute in WO does not appear on the description list, WO is unchanged and the search flip flop reset.

RVA — Replace Value of Attribute — e.g. RVA A

"A" must lead to a description list which is searched for an attribute identical to the attribute in WO.

If the attribute in WO_0 is found on the description list, the value in WO_1 replaces the value on the description list and the search flip flop is set. WO_0 and WO_1 remain unchanged.

If the attribute in WO_0 does not appear on the description list, the search flip flop is reset. WO_0 and WO_1 remain unchanged.

EVA — Erase Value and Attribute — e.g. EVA A

"A" must lead to a description list. This list is searched for an attribute identical to the attribute in WO.

If the attribute in WO is found to be on the description list, both the attribute and its value are removed from the list and the search flip flop set. WO remains unchanged.

If the attribute in WO does not appear on the description list, the search flip flop is reset. WO remains unchanged.

EDN — Erase Description List Name — e.g. EDN A

"A" must lead to a describable list. The name of the description list is returned to available space and the list is made non-describable.

If the list is either non-describable or the top element on the list is not a name, the appropriate statement is printed out and the program continued.

As an illustration of the use of the system, the code for an example problem is given. This code is a realization of a proof procedure for the propositional calculus developed by Hao Wang[6]. For a detailed description and justification of the procedure, the reader is referred to the paper cited. In brief, however, the procedure is as follows:

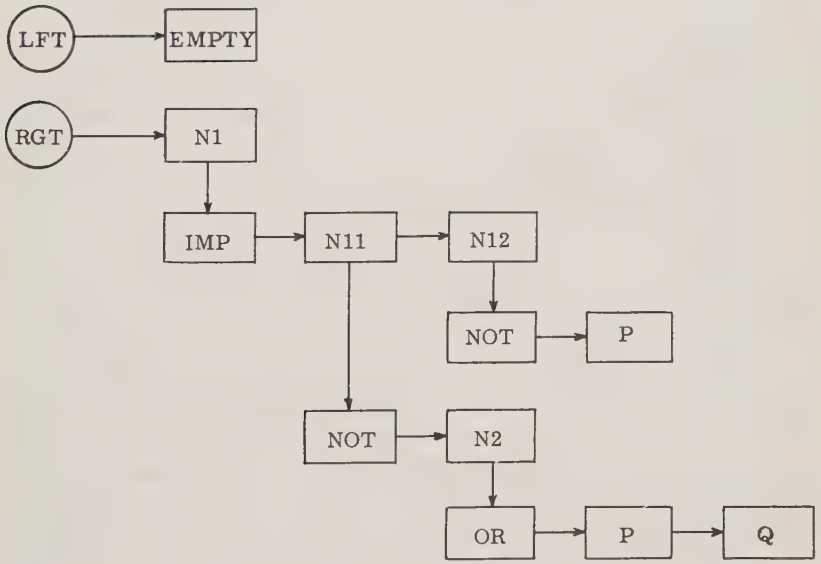
Given any theorem to be proved, that theorem is first written in Polish prefix notation. This notation is convenient in that it

brings the main connective to the front of the expression. An arrow is prefixed to the total expression which is then thought of as having a LEFT and a RIGHT side with respect to the arrow.

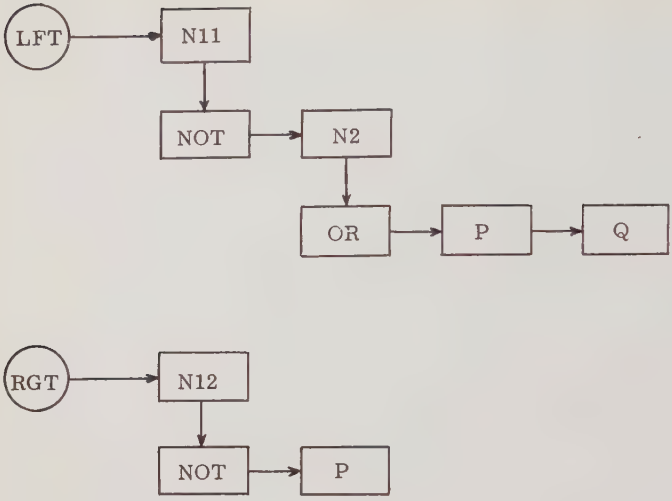
Initially, of course, the LEFT side is empty. Associated with each connective and the side on which it appears, there is a rule which eliminates that connective and which dictates which of the connected terms are to be either left in place or moved to the other side of the arrow. Certain rules also dictate that a subproblem is to be generated, that is that the connected terms are to be reassigned to LEFT and RIGHT sides respectively in two different ways. One of the expressions so created is then set aside as a problem to be treated subsequently. When finally all connectives have been removed by the application of the rules, the LEFT side is inspected to see if it contains terms in common with the RIGHT side. If it does, then that portion of the proof indicates validity. If all subproblems end validly, the theorem is true, otherwise not.

The way the program handles these issues can be seen from

((NOT (P OR Q)) IMP (NOT P))



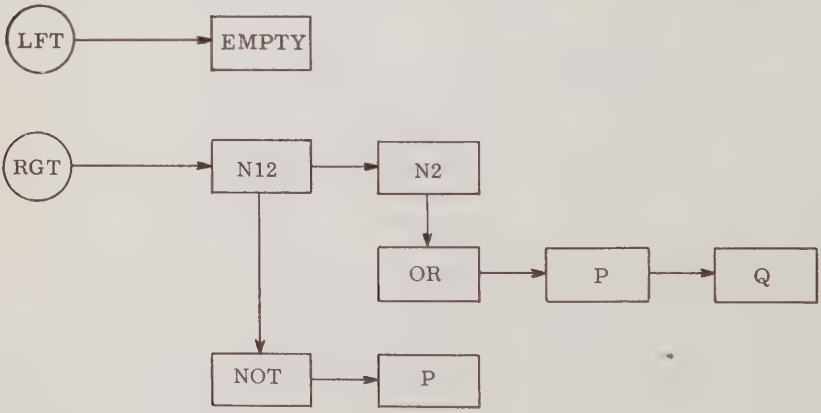
I



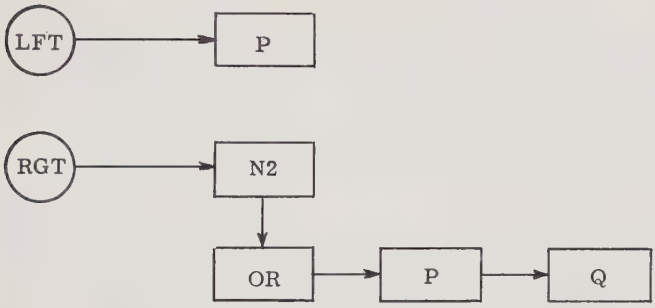
II

the accompanying illustration of the proof of the simple theorem (Not (P Or Q)) Imp (Not P)).

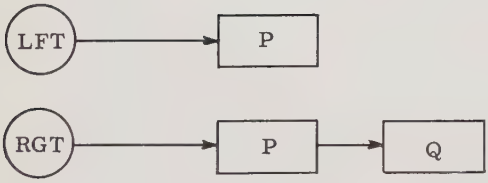
Initially the LEFT side is empty, while the RIGHT side contains the whole theorem in a list structure form. Notice that every connective is the first (i.e. top) element of a separate list



III



IV



V

which is, in fact, a sublist of some list structure. The program now looks for the first name of a list on the LEFT side. Finding none there, it proceeds to look for a name on the RIGHT side. This name is that of a list which has as its top element the connective "IMP". The appropriate rule for the "IMP" connective found on the RIGHT side causes the first term of the implication to be moved to the LEFT side and the second term to remain where it is, the connective itself being eliminated. Thus the second picture is generated. This time, the search for a *name* on the LEFT side is successful. The "NOT" connective is found and the appropriate rule applied. Going on in this fashion finally results in the fifth picture. There the literal "P" appears on both the LEFT and the RIGHT sides and there are no more names of lists on either side. Since, in this simple example, there are no subproblems, the theorem is proved.

The accompanying program appears more complicated than the discussion would lead one to believe mainly for the reason that it includes editing functions both for input and output. The theorem

to be proved, for example, is put into the system in ordinary parenthethis notation which is then translated to list structure format by the initialization procedure. Similarly, the output is presented in a form a little more readable to a nonspecialist than a straight printout of the internal machine format would be. Both these editing programs, by the way, provide good exercises to the interested reader.

An important consideration relating to the efficiency of the system is that when a list structure is moved, only the name of that structure is actually transferred to a new location in memory. The structure to which that name "points" is carried along as a natural consequence. This illustrates the important property of list manipulation systems that entities of arbitrary complexity may be manipulated as if they were in fact single symbols. Another important observation is that the program is not written with any detailed knowledge of the depth or complexity of the list structures which it must manipulate. The program knows, so to speak, only the general characteristics of these structures. The complexity is contained in the way in which the data is stored, as opposed to the more usual situation in which data is stored very "regularly" but programs are very complex. Furthermore, the complexity in data organization arises partially as a result of processing, i.e. is created dynamically. In particular it is not planned in great detail—only in general outline—by the programmer. Thus a great planning burden is lifted from the shoulders of the programmer.

	PROGRAM	WANG3	YYY	PROGRAM	WANG3	YYY
	LIST			ED26	BU	ED24
EDIT	CRN	X3	2	LPP	TID	LP
	CRN	LED			BTF	J1
	INN	LED			TERM	
	STS	OUTP		J1	VST	ED23
	BTO	AOO			TERM	
	IC2			J5	STOP	
	BCF	J5		ED2	RLW	ER1
	OSP2	WO			TID	NOT
	APR	ER1			BTF	XK1
	RLW	ER1			VST	LPP
	VST	ED2			RL4	OUTP
	ERL	L			RLW	ER1
	ERL	R		XK1	PL4	OUTP
	INN	OUTP			RLW	ER1
	PLO	R			VST	LPP
	VST	AOO	3		PL9	OUTP
	BU	EDIT			RLW	ER1

PROGRAM	WANG3	YYY
	TERM	
ED23	CRN	WO
	PSO	OUTP
	VST	ED2
	INN	OUTP
	RSS	OUTP
	TERM	
DONE	AN	END
LP	AN	(
RP	AN	)
AOO	INN	EG
AO	CLER	R1
	CLER	LFT
	CLER	RGT
	APR	R1
AO11	RLW	R1
	APR	LFT
	RLW	R1
	APR	RGT
	BU	AO1
A1	CLER	CRT
	RSRL	RGT
	RSRL	LFT
	RLN	LFT
	BCF	A2
	APR	CRT
	NUL	LFT
	RLW	CRT
	BCF	A3
A3	FVA	LCL
	BSF	A31
A31	XEQ	
A2	RLN	RGT
	BCF	A4
	APR	CRT
	NUL	RGT
	RLW	CRT
	BCF	A5
A5	FVA	RCL
	BSF	A51
A51	XEQ	
RNOT	RLW	CRT
	PL9	LFT
	BU	AO1
ROR	RLW	CRT
	PL9	RGT
ROR1	RLW	CRT
	PL9	RGT
	BU	AO1
RIMP	RLW	CRT
	PL9	LFT
	RLW	CRT

PROGRAM	WANG3	YYY
	PL9	RGT
	BU	AO1
RAND	VST	A20
	RLW	CRT
	PL9	W2
	BU	ROR1
RIFF	VST	A20
	RLW	CRT
	PL9	RGT
	PL9	W1
RIFI	RLW	CRT
	PL9	LFT
	PL9	W2
	BU	AO1
LAND	RLW	CRT
	PL9	LFT
	BU	RNOT
LOR	VST	A20
	RLW	CRT
	PL9	W1
	BU	RNOT
LIMP	VST	A20
LIMI	RLW	CRT
	PL9	W2
	BU	RNOT
LIFF	VST	A20
	RLW	CRT
	PL9	W2
	PL9	LFT
	BU	RIF1
AO1	INN	LFT
	VST	OSPA
	INN	RGT
	VST	OSPA
	BU	A1
A4	RSRL	LFT
A41	RLW	LFT
	BCF	A50
	LOC	RGT
	BSF	VALD
	BU	A41
A50	OCP	D1
	BU	A6
D1	AN	FALS
QED	AN	QED
VALD	OCP	QED
	BU	A6
A6	CLER	R1
	CLER	RGT
	CLER	LFT
	CLER	CRT
	INN	X3

```
PROGRAM  WANG3   YYY
         APR     R1
         RLW     R1
         BCF     A7
         RSRL    R1
         CRN     X4
         RLW     R1
         NUL     R1
         PL9     X4
         RLW     R1
         NUL     R1
         PL9     X4
         INN     X4
         APR     R1
         BU      AO11
A7       OCP     DONE
         TERM
A20      COL     LFT
         PL5     X3
         STS     W1
         COL     RGT
         PL5     X3
         STS     W2
         TERM
         LMODE
1        +
G-OSPA   CRN     OUTP
AO       APR     RDR     2
         VST     A2
         OLP2    OUTP
         CLER    OUTP
         CLER    RDR
TERM     TERM
A2       BWN     A3
A5       PL9     OUTP
         BU      A11
A3       TLLO    WO
         BTF     A4
         INP     LP
         PL9     OUTP
         RSW     RDR
         VST     A2
         INP     RP
         PL9     OUTP
         RVO     RDR
         BU      A11
A4       INP     MT
```

```
PROGRAM  WANG3   YYY
         BU      A5
LP       AN      (111
RP       AN      )111
MT       AN      ()11
A11      RLW     RDR
         BCF     TERM
         PRS     WO
         INP     COMA
         PL9     OUTP
         RSS     WO
         BU      A2
COMA     AN      .111
         +
         GMODE
         LEND
LCL      LIST
NOT      AN      NOT
         BU      ROR1
OR       AN      OR
         BU      LOR
AND      AN      AND
         BU      LAND
IMP      AN      IMP
         BU      LIMP
IFF      AN      IFF
         BU      LIFF
         LEND
RCL      LIST
         AN      NOT
         BU      RNOT
         AN      OR
         BU      ROR
         AN      AND
         BU      RAND
         AN      IMP
         BU      RIMP
         AN      IFF
         BU      RIFF
         LEND

EG       LIST
         RO      L
         RO      R
         LEND
```

ACKNOWLEDGMENT

The actual coding of the KLS simulation system, including the assembler and input-output systems associated with it, was done by Mr. R. C. Shepardson, Mr. D. M. Masters, and Mrs. B. R. Hellerstein, with Mr. Shepardson acting as project leader. The attempt to actually put this system "on the air" resulted in a number of new insights and some modifications of the original scheme. Were it not for the loyalty and devotion contributed by the above mentioned team, this work would still be in the general philosophising stage.

The above citation of references is meant to indicate not merely that the works mentioned were consulted during the course of this effort, but the very real intellectual debt which the author owes to those contributors. The greatest indebtedness is obviously to the pioneering work of the creators of IPL-V.

REFERENCES

1. Collins, G.E., A Method for Overlapping and Erasure of Lists, Comm. ACM, 3 (1960), 655 - 657.
2. McCarthy, J., Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I., Comm. ACM, 3 (1960), 184 - 195.
3. Newell, A., et al., Information Processing Language V Manual, Sect. I, II, Rand Corp., P.-1918, March, 1960.
4. Perlis, A.J. and Thornton, C., Symbol Manipulation by Threaded Lists, Comm. ACM 3 (1960), 195 - 204.
5. Shaw, J.C., et al., A Command Structure for Complex Information Processing, Proceedings of the 1958 WJCC, May 1958.
6. Wang, Hao, "Toward Mechanical Mathematics", IBM Journal of Research and Development, V.4, No. 1, 1/60, p.2.

A GROWING TREE FOR DESCRIPTOR LANGUAGE TRANSLATION*

W. I. Landauer and *N. S. Prywes*

University of Pennsylvania,
The Moore School of Electrical Engineering,
Philadelphia, Pennsylvania, USA

1. INTRODUCTION

The impetus for using the Growing Tree structure as a stratagem for an efficient descriptor language translation stems from recent attempts to program computers for problems that require intelligence for their solution by humans. In these programs the entire structure of data evolves in the course of processing. Thus, it is necessary to have the program construct its own data structure dynamically.

A major objective of the work presented in this paper has been to facilitate communication, with a computer that is applied to information storage and retrieval problems and generally to simulation of cognitive processes. For an efficient operation of such a computer, it seems that one should be able to "drop" the information-carrying items into an automated file and then later be able to recall them by name. Such a process is desirable for a variety of applications where the items dropped may be the routines that compose a program library or that constitute the data belonging to a documentation file. An application of this type will be described in Section 5 by illustrating the use of the Growing Tree as a basic constituent in the Multi-List System.[1]

2. DESCRIPTION OF THE TREE

A *tree* is a framework of *nodes* and *branches* arranged in levels as illustrated in Fig. 1 and Table 1. A number of branches, fixed

*The research reported in this paper was supported partially by Rome Air Development Center Contract No. AF30(602)-2382 and Office of Naval Research Information Systems Branch Contract No. NOnr551(40).

TABLE I.

Tree Items (Nodes)	Address of Associative Catenae	Contents of Associative Catenae		
		Symbol	Key	Link Address
I_1	A_0	S	K_0	A_6
	A_1	S	K_1	A_3
	A_2	S	K_2	A_9
I_2	A_6	S	K_3	A_{30}
	A_7	S	K_4	A_{33}
	A_8	S	K_0	A_{36}
I_3	A_3	S	K_5	A_{21}
	A_4	S	K_6	A_{24}
	A_5	S	K_1	A_{27}
I_4	A_9	S	K_7	A_{12}
	A_{10}	S	K_8	A_{15}
	A_{11}	S	K_9	A_{18}

in advance and in conformity with the conditions of minimum retrieval time, emerges from each node. For the sake of simplicity only a three-way branching of the nodes will be considered in the following examples. The optimum number of branches may be individually determined for each case.[2] Fig. 1 (and also all subsequent figures) portray the tree in an inverted form, i.e., the root point is at the top and the branches are oriented downward.

The *nodes* of the tree correspond to *tree items* I_1, I_2, etc., in Fig. 1. An item contains *associative catenae* which correspond to the number of branches emerging from each node. The catenae of an item are stored in consecutive addresses (exceptions to this rule are discussed later). The address of an item is then that of

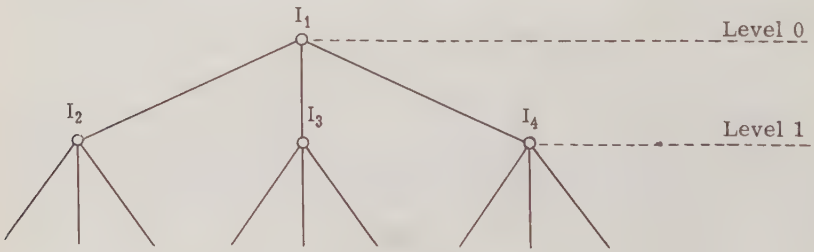


Fig. 1. The tree in schematic form.

its first associative catena. As shown in the table at the top of Fig. 1, an *associative catena* contains a *symbol*, a *key*, and an *address*. The *key* is a numerical value representing one or more descriptors. The key values in the associative catenae of an item are stored in a monotonically increasing order; e.g., in item I_1, $k_0 < k_1 < k_2$ and in I_2, $k_3 < k_4 < k_0$. If a tree is portrayed, as in Fig. 1, the items that are in a horizontal row make up a level of the tree; e.g., items I_2, I_3 and I_4 are in the second level from the top of the tree. Regardless of their storage location, the values of the keys that are in a single level are arranged in monotonically increasing order of their values. Consequently, for the second level in Fig. 1, $k_3 < k_4 < k_0 < k_5$, etc. The *addresses* provide the links between *parent* and *filial* items; e.g., in Fig. 1, I_1 is a parent item with respect to its filial items I_2, I_3, and I_4. Also, k_2 is a *parent* key with respect to the *filial keys* k_7, k_8, and k_9.

The *symbol* consists of two bits. One bit distinguishes between an associative catena, (0), whose contents are described above, a and a *data catena* (1), that contains information which has not been classified in the form of descriptors. The second bit indicates the last catena in an item, i.e., *End of Item* (1). Tree items are composed of associative catenae only, whereas *data items* contain associative as well as data catenae.

The branches emanating from the nodes of the lowest tree level enter the *multi-association* area as *lists*. A list is a string of items which have at least one key in common. Data items are organized in lists in the *multi-association* area of the memory. The data which are on a list have a single characteristic described by the common key. These data may be located by using a *reference key* in tracing through the tree and obtaining the address of the head of the list (see Fig. 2). The reference key serves then as an input to which all other keys are compared in a searching process. In our discussion a *list key* has a value which is equal or larger than, but nearest to, the value of the

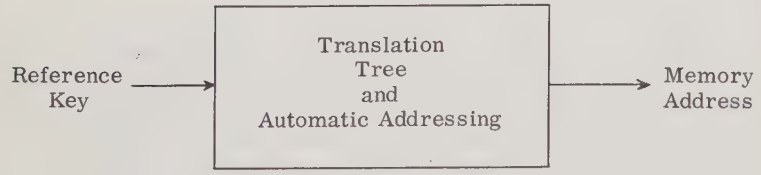


Fig. 2. Descriptor translation in schematic form.

reference key. Tree branches provide for a path through the tree to the list that is identified by the reference key.

A *Balanced Tree* is accomplished by filling the levels of the tree to completion, i.e., no new levels are to be initiated as long as there are vacant catenae available in the preceding level. Consequently, at a certain stage of growth, the only incompletely filled tree level is the lowest one. In traversing the balanced tree from the root to the multi-association area via different paths, the number of nodes (and, hence, the number of tree items) encountered can at most differ by ONE. In other words, a search which requires the traversal of the balanced tree is quasi equal in time for all the lists. A tree which encompasses all the branches and ramifications of branches emanating from any tree item is a *Subtree*. The order of the subtree depends on the level where it starts to branch. If the root of the subtree is located on the lowest level, the order of the subtree is zero; if the root coincides with the root of the entire tree, the order is n; if the root is located between the lowest and highest level, the order of the subtree is n - v, where v indicates the level count that begins from the root of the entire tree.

3. THE CONSTRUCTION OF A BALANCED TREE

The balanced tree is built up by adding progressively more keys as more data items are entered into the Multi-List memory. The keys of the first data item to be filed in the memory constitute the tree item corresponding to the root of the tree. Keys of subsequent data items either may have been entered into the tree previously or may be encountered for the first time. Accordingly, various cases of incorporating these keys in the tree structure arise. These cases are discussed and illustrated in this section. The addresses associated with the keys are omitted in the following examples, and instead they are indicated by directed lines connecting the catenae of a parent item to the corresponding filial items.

In order to gauge the growth of the tree, a count of the number of levels is maintained and updated with the start of a new level. Since the construction of a new level, n + 1, will be initiated only if level n is completely filled, new keys will always be entered in the lowest level of the tree. A new tree item will be added in the n^{th} level if a vacancy for an item exists. The new tree item contains only two keys in the beginning, namely, the parent key re-

curring in the last catena of the new tree item and the key whose insertion provoked the formation of the new tree item. The first, second, etc., up to the penultimate catena of the new tree item remain vacant until the insertion of an additional key, due to the addition of a new list, requires their usage. Hence, vacant catenae will be found at the left of an incompletely filled tree item; consequently, in search for an empty space the inspection of the first catena yields the information if any vacant catenae (one or more) are available in the particular item.

An alternate approach may allocate for the new tree item as many catenae as are actually needed at a time. Whenever in an incompletely filled tree item the need for an additional catena arises the incompletely filled tree item is deleted and the memory space for a new tree item with the required number of catenae is allocated from the unused memory space. A readjustment of the link addresses to preceding and subsequent items is necessary. However it would be more cumbrous to describe this scheme, therefore, the previous one is used in the rest of this section.

When a new item is to be added to the information stored the first step is to determine if the keys characterizing the new item exist in the memory (i.e., if they are part of the existing tree).

If items that have the same keys as the currently added item were filed previously then the lists in the multi-association area are located and the new item is added to the corresponding lists according to a desirable order; the tree remains unchanged. Figs. 3a and 3b illustrate the state of the multi-association area before and after the filing of an item characterized by the keys 27 and 78.

When only some of the required lists exist, new lists have to be

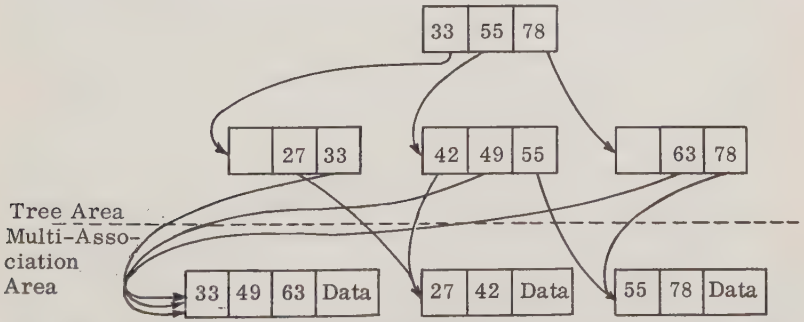


Fig. 3a. State of the tree before filing the item whose keys are 27, 78.

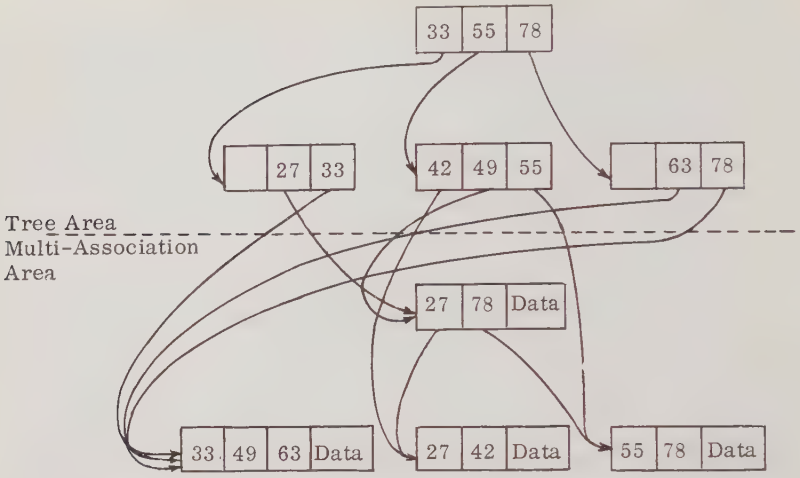


Fig. 3b. State of the tree after filing the item whose keys are 27, 78.

added. In order to maintain the monotonic increasing order of the key values the keys corresponding to the new lists have to be entered at the appropriate location in the lowest level of the tree. If the tree item corresponding to this location contains a vacant catena the addition of a new list is relatively simple. The vacant catena is located, then the key corresponding to the new list and the address which refers to the item to be filed are inserted in such a manner that the monotonic increasing order of the keys is maintained. If, for example, the values of the keys to be entered in the tree of Fig. 4a, were $k < 17$ or $71 < k < 89$, vacant catenae would be available. However, to accommodate an item whose new keys are 33 and 91 existing keys have to be shifted so that the new keys can be inserted in the proper order. This is shown in Fig. 4b where keys 17, 29 and 89 are shifted each one position to the left so that keys 33 and 91 can be entered. Note that, to accommodate key 33 it was necessary to modify also the parent item. This will be discussed further.

When the tree item which is to contain the list key at the lowest level is completely occupied a search is conducted of the corresponding subtrees for the nearest tree item containing an empty catena. The order of the subtrees increases with the advancing search as shown in Fig. 5a. When a vacant catena is located all the keys of the subtrees, within the limits of which the search is made, are shifted either to the right or left if the vacant catena

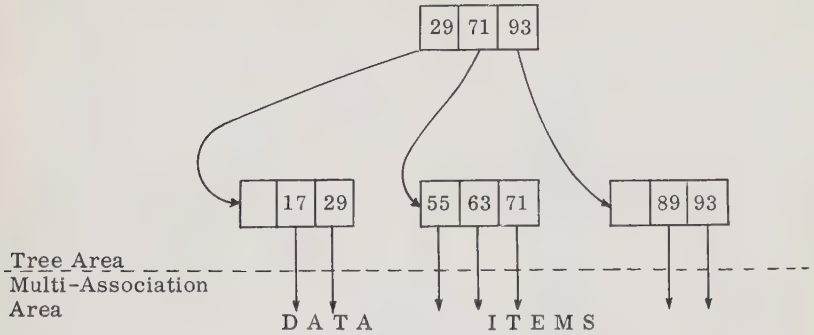


Fig. 4a. State of tree before filing the item whose keys are 33, 91.

is found on the right or on the left of the list key, respectively. When after recursive shifting the catena to the left of the list key is vacated the new key, (which is the reference key) can be inserted.

Fig. 5a portrays the search pattern for the nearest vacant catena. Suppose that a new key, the value of which is 57, has to be added in the tree shown. The search covers the subtrees of order ONE and one subtree of order TWO. The search would have been extended to the second subtree of order TWO had not a vacant location been found in the 0^{th} subtree of order TWO.

After the nearest vacant catena is located, all the keys between 19 and 54 inclusive are shifted to the left such that 57 can be inserted in the proper order. This is shown in Fig. 5b.

4. AUTOMATIC MEMORY ASSIGNMENT

A memory synchronizer constitutes an intermediate link between the random access memory and the processor. Any mem-

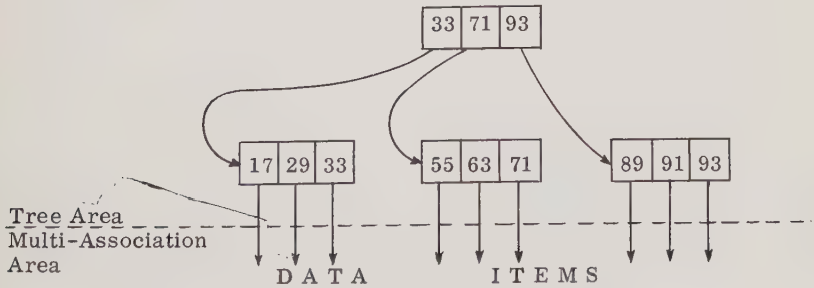


Fig. 4b. State of tree after filing the keys 33, 91.

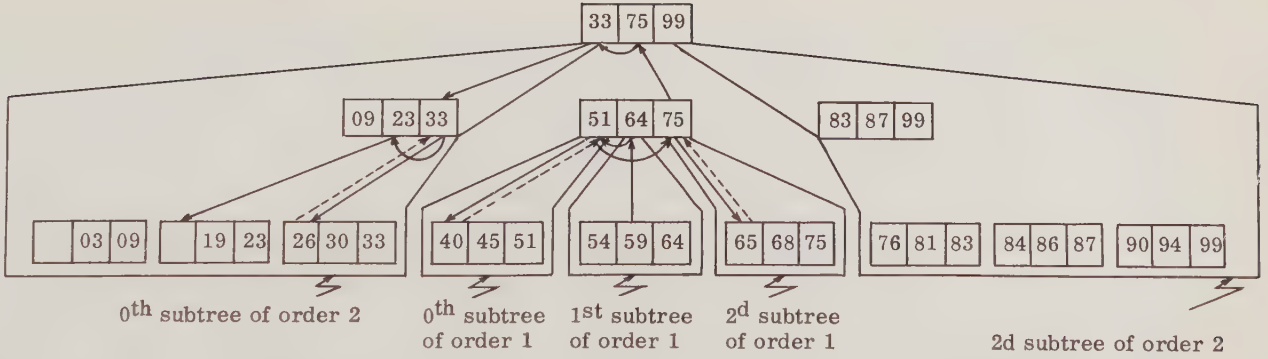


Fig. 5a. Pattern of search for a vacant catena to accommodate the key value 57.

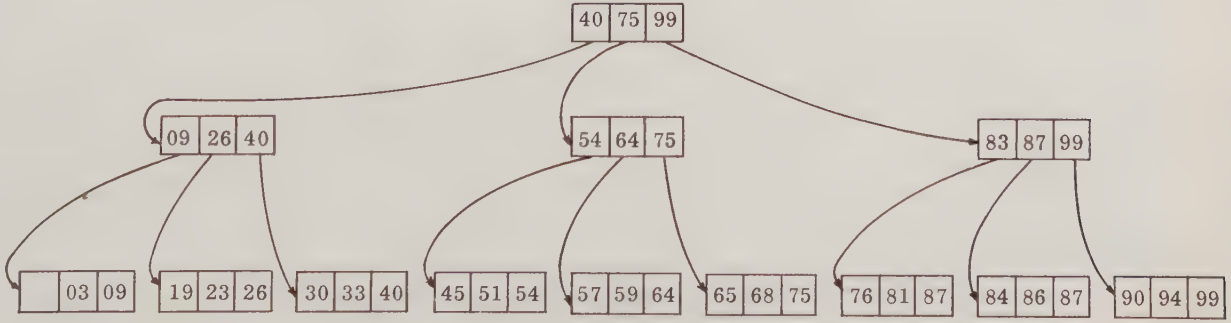


Fig. 5b. State of the tree after the insertion of key value 57.

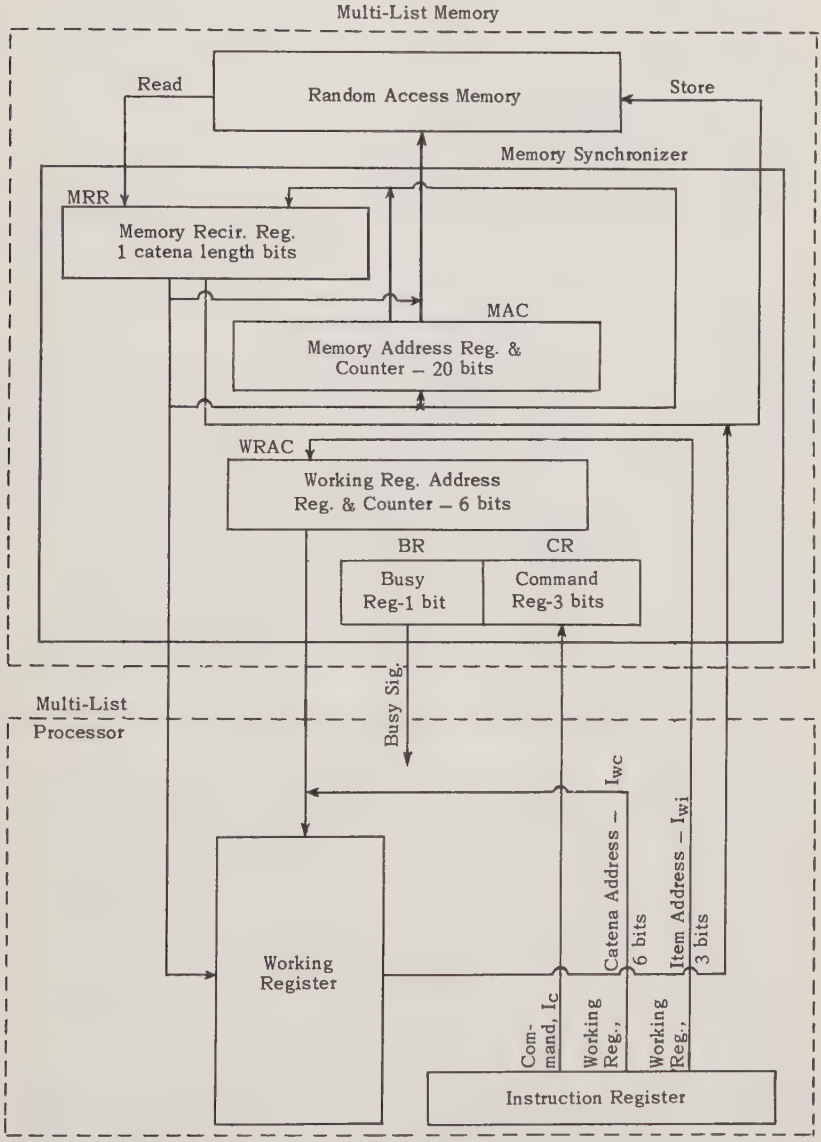


Fig. 6. Block diagram of the memory synchronizer.

ory access must be affectuated via the memory synchronizer, hence, the memory synchronizer may be considered as an integral part of the memory. Among other functions[1] the memory synchronizer stores and deletes information in the memory. These two functions are described in the following:

Fig. 6 presents the functional units of the memory synchronizer and their interconnections in detail.

The *Memory Recirculation Register* (MRR) constitutes an intermediate station in the sequential flow of information from and to the random access memory. The retrieval from and the storage in the memory takes place only in entire units—the so-called catena units; however, from the MRR the contents of the catenae can be dispatched piecemeal to other registers, i.e., symbols, keys (tags), and addresses can be separated and assembled in the MRR. In the flow charts this procedure is indicated by attaching the subscripts s, k, and a, for symbol, key, and address, to the word representing the contents of the register; e.g., MRR_a stands for the address portion of MRR. Absence of any subscript implies an operation on the entire catena.

The *Memory Address Register and Counter* (MAC) contains the address of the catena to be read into MRR in the read-operation, or it contains the address of the location where a catena is to be stored in the store-operation. These operations appear in the flow charts as follows:

(MAC) $\rightarrow$ MRR or MRR $\rightarrow$ (MAC), respectively.

The *Working Register Address Register and Counter* (WRAC) controls the flow of information between MRR and the working register, i.e., the setting of this register determines where in the working register the information is directed to or taken from. The three most significant bits of WRAC are used to divide the working register into 8 segments called items. Each of these items in turn contains 8 catenae. The three least significant bits of WRAC divide each item segment into 8 catenae. These 3 least significant storage elements of WRAC are constructed in the form of an octal counter. WRAC may be set by a six bit instruction code, as I_{wc}, or by a three bit instruction code, as $I_{wi} \cdot I_{wi}$ sets the three most significant bits of WRAC and implies zero-setting for the three least significant ones, which in the Store operations then act as a counter. WRAC essentially constitutes a mod 8 counter counting from a multiple of 8 given by I_{wi} until overflow occurs, i.e., up to $I_{wi} + 8$. Thus the maximum length

of the automatic "Read Item" executed by the memory synchronizer is eight catenae.

In order that the memory synchronizer may continue the execution of instructions which, once initiated, are independent of others executed concurrently in the working register, the so-called *Command Register* (CR) is incorporated in the structure of the memory synchronizer. This register contains the command to be executed in the memory synchronizer which, together with the location of the operands, is given by the instruction. The *busy signal* is set at the beginning of a command execution in the memory synchronizer and is reset at the end of it.

The memory space for storing the catenae of an item is obtained from the *List of Available Space* (LAS). Normally the available space is contiguous, i.e., the catenae of an item are stored in consecutive locations. If the number of consecutive locations is not sufficient to accommodate all the catenae of an item to be stored, *jump catenae* are inserted to provide the links between areas of the LAS which only together have a sufficient number of consecutive locations. The memory synchronizer executes this type of non-contiguous storage by a) obtaining the available space from the LAS, b) storing the item at these locations, c) inserting jump catenae in the appropriate positions, and d) inserting the address of the new first available space into the head of the LAS, so that the LAS is readily accessible for future storage requirements.

Flow Charts for the Memory Synchronizer Operations

BS indicates "Busy Signal"
I_S indicates the symbol in the instruction catena
EOI indicates "End of Item" symbol
I_C, I_{Wi}, I_{WC}, as well as MRR, MAC, WRAC and their subscripted versions are explained in Section 4.

(x) indicates the content of a register whose address is x
((x)) indicates the content of a register whose address is stored at address x.

The branches of a branch point represent the different outcomes of a decision; the straight continuation of the decision in question represents the affirmative outcome, while a negative outcome leads to the branch headed by a "NO" in case of a single decision or a "NONE" when several decisions are involved.

"Store Item" Flow Chart

a) Instruction Format

I_c (6 bits): "Store Item"

I_{wc} (6 bits): Working register catena location into which the synchronizer places the memory address of the item to be stored.

I_{wi} (3 bits): Working register item address containing the item to be stored (there are eight such items which overflow into the memory).

b) Mode of operation

In the "Store Item" operation, the new memory space, to be taken from the LAS, has to be allocated for the item to be stored. The description of the process is given in Figs. 7a and 7b.

The item to be stored is found in the working register item location designated by I_{wi}. The address of the memory space to be allotted for the item to be stored is taken from the head of LAS and transferred into the working register location designated by I_{wc}.

The LAS is categorized into two classes, which have different priorities with respect to their employment as available space. Those catena which were not used during the current memory organization are said to belong to the class of unused LAS catenae. Those catenae which were used before, but were returned to the LAS by invoking "Delete Item" operations, are said to belong to the class of used LAS catenae. Blocks of used LAS catenae are likely to be interspersed among items belonging to other lists. These "deleted" blocks are interconnected by jump catenae.

The memory space for an item to be stored will be assigned from the class of unused LAS catenae as long as members of this class are available. If this class is exhausted, i.e., the memory location 11. . .1 has been encountered, flip-flop T_1 will have been set. This indicates that further storage space has to be provided by the class of used LAS catenae. If the class of used LAS catenae also becomes exhausted during a "Store Item" operation of the "End of LAS" Contingency Routine is invoked.

If a "Jump" is encountered among the catenae coming from

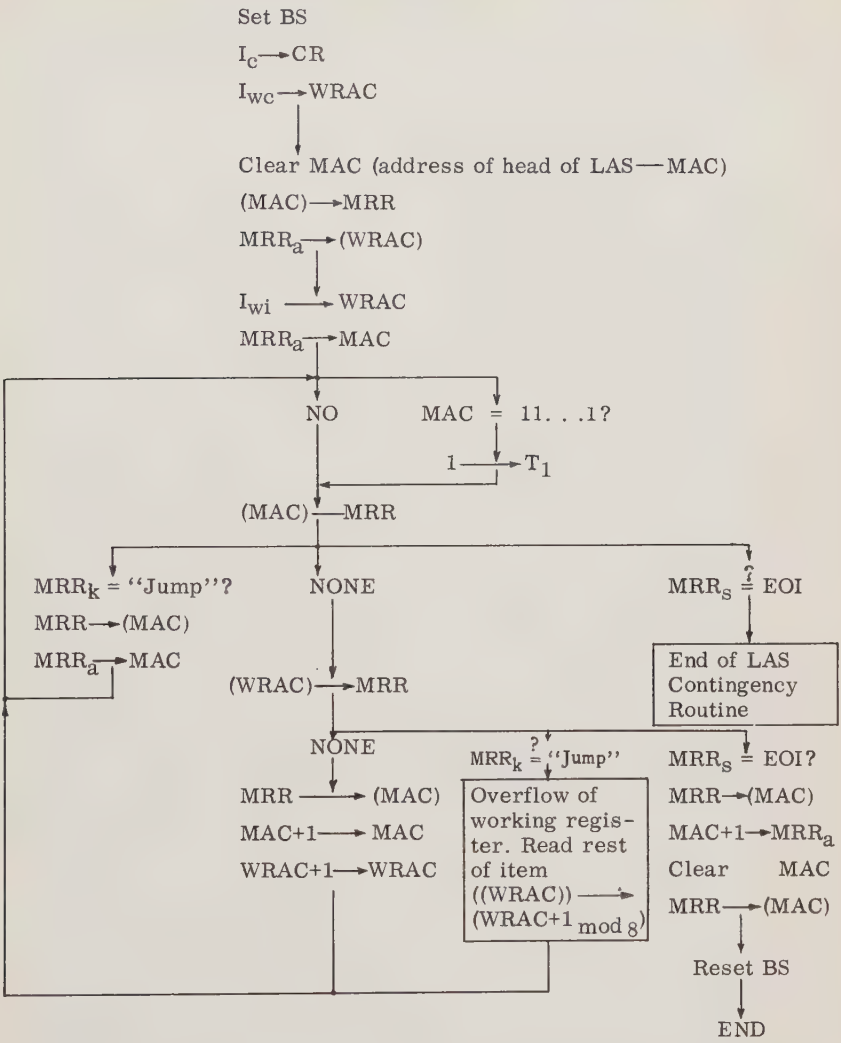


Fig. 7a. "Store item" flow chart.

LAS, the contents of the jump-carrying catena is returned to the memory without alteration. Transfer and storage of portions of the item are repeated until the entire item is stored. This feature of the "Store Item" mechanism lends itself readily for a transfer of items among different memories in a hierarchy of

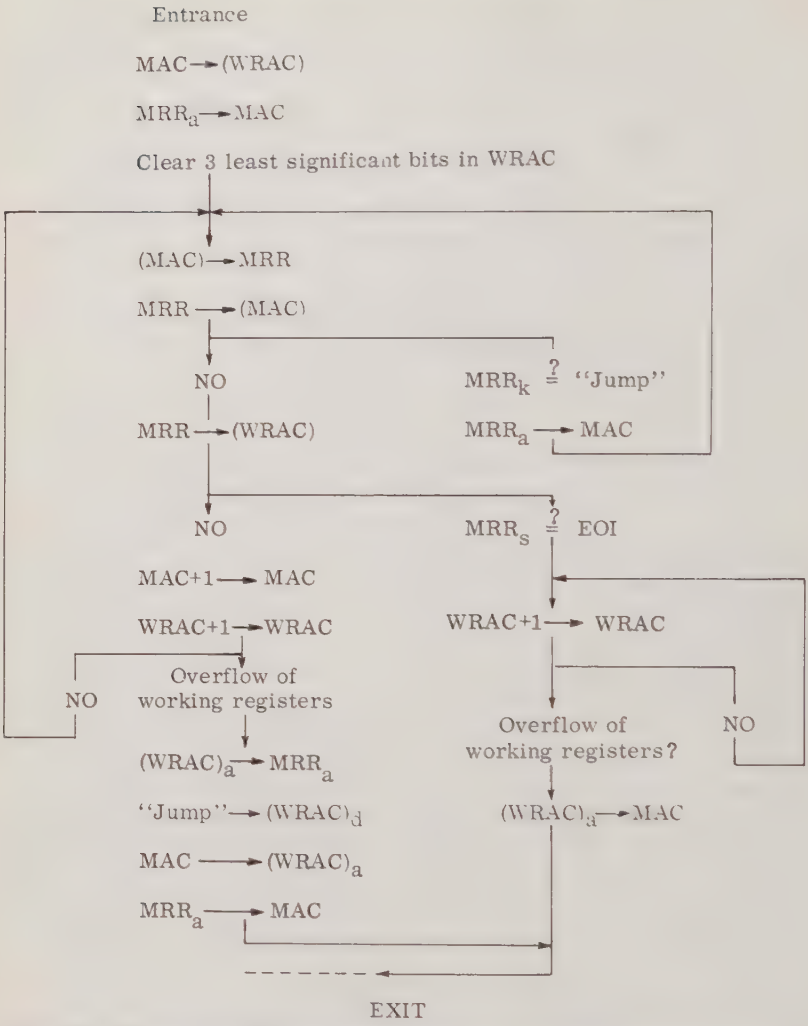


Fig. 7b. "Store item" flow chart, overflow of working register item capacity; read rest of item.

memories. If the "End of Item" symbol of the item to be stored is encountered, the address of the next available catena is placed into the Head of the LAS and the "Store Item" operation is terminated.

"Delete Item" Flow Chart

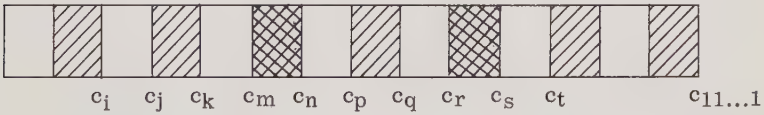
a) Instruction Format

I_c (6 bits): "Delete Item"
I_{wi} (3 bits): Working register address of item the first catena of which contains the address of the item to be deleted. The second catena of this item is used for temporary storage in the "Delete Item" operation.

b) Description of the Flow Chart (cf. Fig. 8)

The execution of the "Delete" instruction begins by clearing MAC. If flip-flop T_1 is reset the access to the LAS for "Delete Item" purposes takes place via the last LAS catena whose address is 11. . .1; consequently MAC has to be so set in order to address this catena. If flip-flop T_1 is set the access to the LAS takes place via the Head of the List of Available Space. Flip-flop T_1 remains reset as long as unused LAS catenae are available, it will be set as soon as the class of unused LAS catenae is exhausted during the "Store" operation.

The following schematic represents a picture of the memory in which the hatched areas indicate memory space that belongs to LAS while the cross-hatched areas represent the memory space due to the item to be deleted.



Let c_i, c_j, . . ., $c_{11...1}$ represent the catenae the addresses of which are i, j, . . ., 11. . .1.

1. If a jump catena, say c_i, is encountered, a comparison between the address to which the jump refers and the address of the item to be deleted takes place.

a) Suppose the address to which the jump refers, j, is smaller than the address of the item to be deleted, which is m; then, beginning with catena j, successive catenae are read out without restoration.

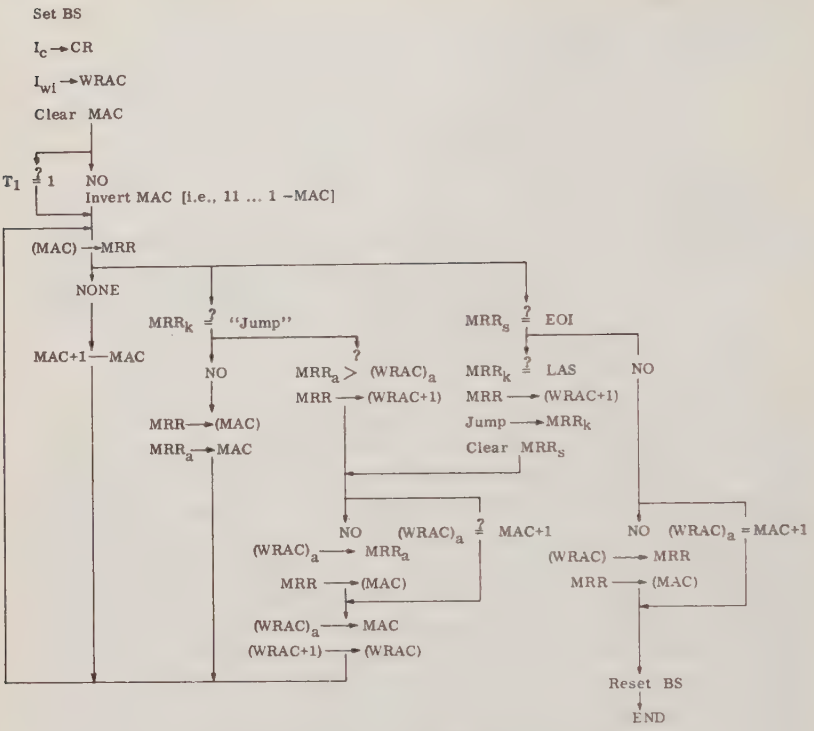


Fig. 8. Delete item flow chart.

b) Suppose a jump is encountered in catena c_k; the address to which the jump refers, p, is then larger than the address of the item to be deleted, say m. In this case, the contents of the jump catena will be stored temporarily in the working register; a jump catena whose address refers to the item to be deleted, i.e., c_m, will be re-stored if catenae c_k and c_m are not contiguous. Successive catenae of the item to be deleted are then read out and not re-stored. If c_k and c_m are contiguous the jump catena will not be re-stored and the read-out of the item to be deleted takes place. A very similar procedure to the one outlined above is followed if a jump is encountered within the item to be deleted, e.g., in catena c_n.

2. Suppose a catena of the LAS, for example, c_q, contains an EOI symbol. Its contents will be stored temporarily in the working register; a jump catena whose address refers to the item to

be deleted, i.e., c_r, will be inserted in its place if catenae c_q and c_r are not contiguous. Successive catenae are then read out and not re-stored. If c_q and c_r are contiguous the EOI catena is not replaced by a jump catena.

3. Suppose the EOI symbol of the item to be deleted is encountered, e.g., in c_s. If c_s and c_t are not contiguous the EOI catena is replaced by the catena that was temporarily stored in the working register, in this case c_q, and the delete operation is terminated. If c_s and c_t are contiguous, the operation is terminated without replacing the EOI catena.

5. APPLICATIONS OF THE GROWING TREE

The process described above translates a key having a fixed format into an address. This process however can be used as the basic building block for the more difficult tasks of translating a partial description of an item, consisting of several descriptors, which may correspond to one or more keys, to locate items of varying length. How this is done is the subject of a lengthy discussion. Generally, an associative memory utilizing an addressable memory is obtained. The range of applications of the tree mechanism is illustrated through the schematic description of the Multi-List System. The applications in this case are in handling symbolically both programs and data.

The flow of information in the system is described schematically in Fig. 9. The functions of the blocks in Fig. 9 are described in the following, thereby describing the entire system.

a) *Inputs*. The inputs to the system come from three sources; these are:

i. *New Data*. These may be in the form of data and associative catenae organized in item units. The associative catenae contain descriptors (or keys), which represent library or file concepts, and blank spaces for the link addresses. These addresses are to be assigned through the automatic memory space assignment discussed previously.

ii. *Programs Defining New Words in a Pseudo-Language*. These programs may each consist of a number of instruction items and may have recursive portions. Addition of such programs to the storage amounts to adding words to the vocabulary in the pseudo-language.

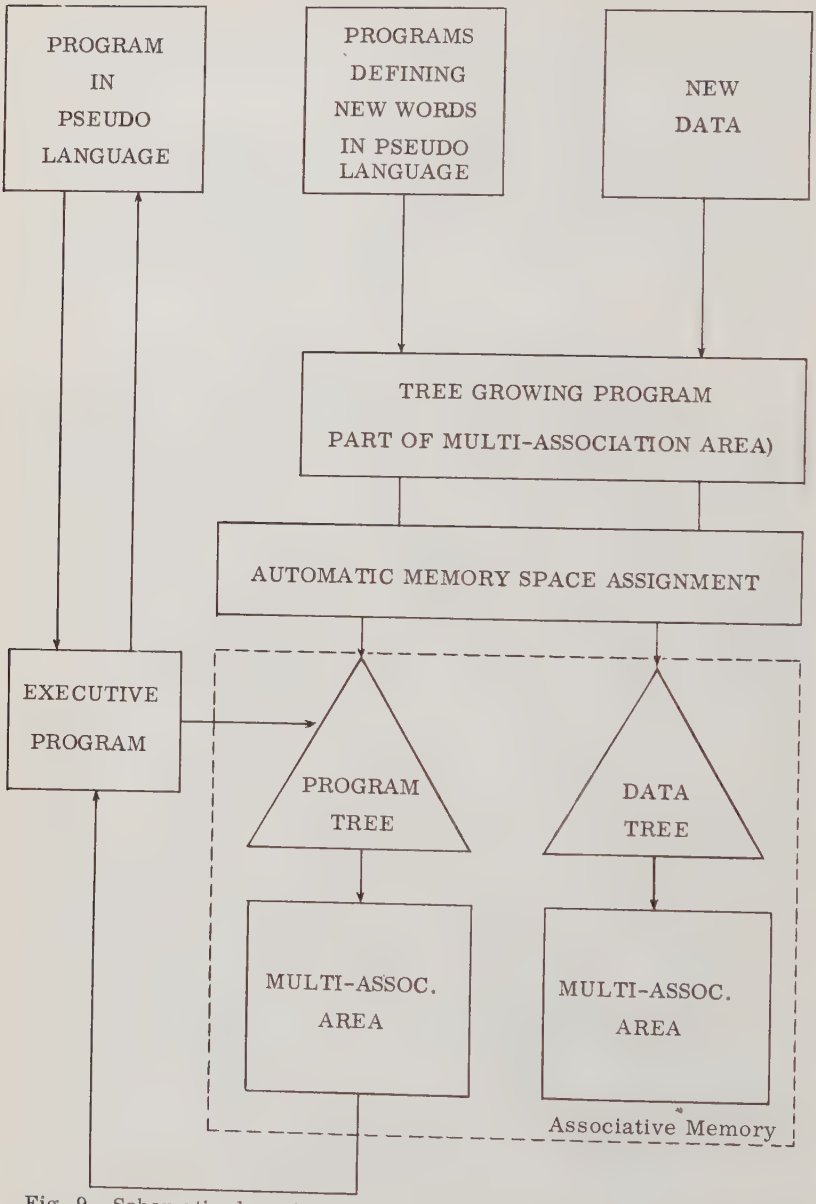


Fig. 9. Schematic description of flow of instructions and data in the interpretive mode.

iii. *Programs in Pseudo-Language.* Such programs consist of a list of command-words and names of operands. Next to each command-word there may be several names of operands. The command-word then is a word in the vocabulary of the pseudo-language. The names of operands are words in the vocabulary of the descriptors.

b) *Tree Growing Program.* The inputs, programs or data, are subjected at this point, to the tree mechanism which assigns them according to their key values to the appropriate location within the program tree or the data tree, respectively. The tree growing program is essentially an encoding and decoding stratagem for the translation of names of lists (that correspond to commands in the programming pseudo-language, or descriptors of concepts prevailing in a library or file) into the addresses of the heads of the corresponding lists in the multi-association area. The flexibility of the tree manifesting itself in the ease of adding or deleting names is its major advantage in the application at hand.

c) *Automatic Memory Space Assignment.* This process is carried out by the synchronizer which has been described.

d) *The Associative Memory.* The complexity of requests made of an associative memory lead to the use of an addressable memory in which the associative memory is simulated. The memory has two parts, the first of which is a Tree structure, described previously. Successive addressing leads from a single entry point of the tree to an appropriate exit that corresponds to a list defined by descriptors mentioned in the retrieval request. The second part of the Multi-List memory is a *multi-association* area where the file, organized in lists, is actually stored. This area is criss-crossed by the lists originating from the tree mentioned before.

Generally, the trees provide the translation between the names of lists (which correspond to commands in the programming pseudo-language or to descriptors in the data system) and an address in the multi-association area where the corresponding list is stored.

In the multi-association area items both of data and of instructions are stored. The addresses that are associated with the keys in the associative catenae provide the links between data items. In the case of instruction items, the memory addresses that accompany the compare and jump instructions provide the links between instruction items. These addresses are put there by the synchronizer while storing the instruction items involved. Re-

cursive processes are possible entirely within the multi-association area, where the instruction items are stored, without having to go out to the executive program.

e) *Executive Program.* The incoming programs are in pseudo-language and consist of sentences each having a command and names of operands. The command word must have previously been stored with the corresponding program. The names of operands also must have been incorporated in the storage previously.

6. CONCLUSIONS

The tree mechanism described in this paper has been preferred over others[3,4,5] due to a) its self organizing feature, which eliminates need for prior knowledge about size of tree and range of key values, and b) its efficiency in allowing an optimum number of branches at a node as well as being "balanced."

A disadvantage concerns the need for shifting keys when the tree grows. This problem has been studied analytically and work on simulation is currently proceeding. If the keys come in a random sequence, the average number of keys involved in a shift due to the addition of a key is 1.8. For the case of 4 branches per node, the corresponding distance in items would be .6. The situation is far worse when the keys come in a monotonically increasing sequence.

REFERENCES

[1]Multi-List System Technical Report No. 1, Vol. I and II. Prepared for the Office of Naval Research, Information Systems Branch, under Contract NOnr551(40). The Moore School of Electrical Engineering, University of Pennsylvania, November 30, 1961.

[2]"The Multi-List Type Associative Memory," N. Prywes and H. J. Gray. AIEE publications S-136. Proc. of the Session on Gigacycle Computing Systems at the AIEE General Winter Meeting Jan. 1962, pp. 87-107.

[3]A. D. Booth and A. J. T. Colin. On the Efficiency of a New Method of Dictionary Construction, *Information and Control*, Vol. 3, No. 4, Dec., 1960; pp. 327-334.

[4]"An Indirect Chaining Method for Addressing on Secondary Keys," R. L. Johnson. *Comm. of the ACM.*, Vol. 4, No. 5, May, 1961, pp. 218-222.

[5]A. Newell and Herbert A. Simon. The Logic Theory Machine, A Complex Information Processing System. *Rand Corp Report P 868*, July 12, 1956.

ON THE DECLARATIVE CONTROL OF THE DATA FLOW BY MEANS OF RECURSIVE FUNCTIONS

*Lionello A. Lombardi**

Euratom—Cetis, Ispra, Italy

1. INTRODUCTION

This note is devoted to the discussion of algorithms for file processing which are represented in terms of a language without command structures, that is, a declarative language. So far, most of the effort in the area of declarative languages has been in the field of the representation of processes whose results, or output, are numbers or small, well defined sets of numbers. Consequently, such effort has been mainly directed towards the development of syntaxes for writing and semantics for evaluating forms whose values range over a set of numbers. File processes do not belong to this category, since their result, or output, are large sets of records distributed in sets of output files, and the main difficulty in achieving this result is the proper organization of such files. Consequently, since declarative languages consist of methods of representing processes as functions of the data whose values are the results, declarative techniques to represent file processes must be based on forms whose values range over spaces of sets of files. Due to the level of repetitivity involved by file processes, such forms should be designed in a way to allow for recursion.

While the application of recursive functions to forms able to represent operations on random access files or tables is discussed in (13), this note is concerned with coordinated or *collinear* file processes (16), that is, with the file processes which take place in magnetic tape processors. The results of the theory of files (7, 16) are extensively used, with special regard to the standardization and optimization of the input and

*Address for all mail and reprints via Ciceri Visconti 10, Milano, Italy.

the predicate-controlled output (7, section 3). In fact, without the tool of the filetheoretical approach, the problem stated at the end of the preceding paragraph would be very hard to solve.

Since this is a report on preliminary investigations aiming to show the possibility and the basic elements of a declarative approach to the representation of file processes, rather than a manual for a declarative file processing language or for a form-evaluating file processing computer, the schemes discussed in (16) have been considerably simplified. In particular, the *stream* and *bundle* level have not been considered, thus implying, among other things, the need for each logical file flowing through an individual input-output unit, which is possible only if there is a large number of such units available. Furthermore, among the file *indicators*, only the existence and the left and right derivatives of the input files are considered, while the input-output and validity indicators, as well as all indicators of the output files, are provisionally dropped. In fact, these useful features can be added later without any need for major theoretical advances, while their treatment here would make the discussion considerably more complicated.

The file processing algorithms considered here provide for coordinating the input by means of the function $f_{\text{input 2}}$ and computing the contents of the output records and coordinating the output by means of the function $f_{\text{output 2}}$. These two functions are cascaded, in the sense that the second one, which gives the timing to the whole system, calls the first one. One of the most important things that are brought up in connection with this, is that, when two functions are cascaded, the first one should have a value with several components, some of which are the data that it was supposed to produce and pass to the second one, and the others have no other purpose then to enable the second one to assign parameters to the first at the time of its following incarnation. In other words, the link between these two particular functions is a prototype of the nature of communication which should exist between independent moduli—or submachines—of a computer with declarative logic.

Input and output buffering is not discussed here; the entering and filing of records is performed by not better specified functions $f_{\text{physical input}}$ of one variable and $f_{\text{physical output}}$ of two variables, whose value are the next-coming record of the input unit specified by the variable and the contents of the output unit specified by the first variable supplemented with the second

one, respectively. These two functions are somewhat eterogeneous with respect to the remainder of the system, because, as it will appear, in order to be consistent, the contents of I-O units should be represented as lists and operated with composition and decomposition operations. This will be done in the near future probably without major difficulties, while drafting a complete recursive file processor, based on six cascaded levels: physical output, output buffering, output computation and coordination, input coordination, input buffering, physical input. However, in this first essay we have bound our investigation to the two central levels, which give rise to the main difficulties, and to the link between them, which is the most critical. No use is made here of the *set inclusion* operator of order three, which has been widely used in (13), and is essentially a particular interpretation of Gilmore's *dynamic load* operator of order two (4). It has been possible here to replace this problemraising operator with composition operations each time the opportunity of using it came up.

Compositions and decompositions of lists are used extensively in order to build composite values and parameters of functions. Such techniques are derived from those developed by McCarthy (3), though they are used here in a different way and for a different purpose. An example of a composite-value form built in this way is the value of $f_{input\ 2}$, the fifth component of whose value is the whole machine status, while the other eight components contain information sufficient to control its subsequent incarnation. This technique is also used by $f_{output\ 2}$ in order to decode the values of $f_{output\ 2}$ that it received through $f_{input\ 1}$

The problem of developing syntactical elements for composition and decomposition operators is not discussed here. The solution of this problem will be a particular case of a syntax able to define elements of new spaces and operations thereon in terms of the previously available spaces and operations, in a way such that operation letters (or function letters) are morphologically independent of the structure of the operands, while an appropriate meaning is assigned to them for each allowable configuration of such structure. A considerable effort is currently being devoted to the solution to this last problem, which is one of the key questions of the theory of computation. This author will soon report on the results of his investigation, based on extensive

development and theorization of some features of the B5000 and KDF9 computers.

Care has been exercized here to use only the particular class of recursive functions called *external recursive* (13, 15), because a much easier semantics to evaluate them, based on the so-called *discharge stack*, is possible, as shown in the second part of (15).

2. NOTATION

Only very short explanations are given here on the notation philosophy used, since the first part of (15) is devoted to its complete discussion. It should be noticed that practical considerations have suggested some very slight deviations from the notations used in the reports (13, 15), written four months earlier than the present one.

All function letters are denoted by the letter f with a subscript. Besides specifying such functions, these subscripts may be mnemonic and connote the practical use of the function letter. However, functions which are usually denoted with special marks, such as + or −, are here denoted as usual, for the sake of readability.

The definition of a function of order n has the layout $f_A \equiv b$, where f_A is the function letter involved and b is a form in the n variables $x_1, x_2, \ldots, x_n$, where x_i always denotes the i^{th} argument (free variable) of the currently defined function. The form b can contain calls for other functions, the one currently defined inclusive. These calls have the layout $f_B(d_1, d_2, \ldots, d_m)$, where f_B is the function letter of the function called, m its order, and the d_j are forms in the n variables x_i, which have the above specified meaning. Whenever the value of any of the d_j for given $\{x_i\}$ is a function of a finite sequence of (λ-bound) variables, the h^{th} of them is denoted y_h. Unlike most languages for representing algorithms, where each variable, free or bound, has a fixed name assigned to it, in this language the names of free and bound variables are always x and y, respectively, with an integer subscript, and the denotation of such names depends on the position where they are used. The reason for the adoption of this method is its orientation towards address-free evaluation logics based on stack operation, (5), while the conventional approach, by means of the standard gimmick of

associating storage areas with names, is oriented towards the execution of commands on addresses (1).

McCarthy's conditional function f_{cond} of three variables (3), whose value is x_2 or x_3, depending on whether or not x_1 is true, is apodictic in this system and referred to sometimes with the redundant notation f_{cond} $(x_1 \rightarrow x_2\ T \rightarrow x_3)$, for improving readability. A *list* of n elements is represented by the integer n followed by the n elements, all separated by commas. The value of the apodictic function f_{comp} of order $n + 1$ is the list which its arguments represent, while the value of the apodictic function of order one f_{dec}, whose argument is a list (we should rather say "has the dimensions of a list"), is the set of the elements of this list. The value of the apodictic function of order 2 f_{el}, whose first argument x_1 is a positive integer and whose second argument x_2 is a list, is the x_1^{th} element of x_2. By using recursively these three basic functions, called *composition*, *decomposition* and *element*, respectively, it is easy to build two further handy list operating functions of order two, f_{first} and f_{last}, whose value is the set of the first or last x_1 elements of the list x_2, respectively. The value of the apodictic function f_{num} of order one will be the number of elements composing its argument, which is a list.

Recursion is allowed in the function definitions, that is, if such a definition has the form $f_A \equiv b$, calls for f_A can be contained in the form b.

If we establish a weak precedence relation in the space of the function letters involved by an algorithm, such that $f_0 \leqslant f_m$ if there is a sequence $f_0, f_1, \ldots, f_m$ such that f_i is used in the definition of f_{i-1}. Then, if this relation yields a partial weak order, we shall say that the algorithm is based on *a class of individual recursive functions* (the case of the strong order would imply the absence of recursion). As it was remarked by Mauro Pacelli, this way of defining algorithms is possibly less powerful than McCarthy's (3), who explicitly introduces sets of recursive functions defined through systems. However, since the logical problem of comparing the two classes of recursive functions to which this difference has given rise is open and goes far beyond the specific scope of this essay, care has been exercized here to base the file processing algorithm on a class of individual recursive functions.

A handy gimmick which makes it easier to read the definitions of recursive functions consists of replacing with a dash (possibly

in parentheses to avoid confusion with the minus mark) some appearances of x_j in d_j in all function definitions of the type

$$f_A \equiv f_B\ (d_1, \ldots, d_m) \tag{2.1}$$

or

$$f_A \equiv f_{cond}\ (d_A \rightarrow d_B,\ T \rightarrow f_B\ (d_1, \ldots, d_m)) \tag{2.2}$$

The letter z with a subscript (possibly mnemonic or descriptive) is called *shorthand*, that is, it replaces graphically the occurrence of an arbitrary writing, which is associated with it by means of a *shorthand definition* having the layout

$$z \equiv d$$

where z is the shorthand and d is the writing that z stands for. Such shorthand definitions have been already introduced in (8) and (16), where they were called *symbols*.

Despite the confusion between function and shorthand definitions that the common layout and the common use of the mark "≡" might yield, one should keep in mind the deep semantic difference between them: they denote functional equivalence and graphical identity, respectively, and the occurrence of ≡ is related to the McCarthy's *label* operator (3) in the first case, while it denoted plain identity or replacement in the second. The limited purpose of shorthands is to simplify the design of algorithms and to cut the size of the (real or simulated) computers which carry them out.

The letter v with a descriptive subscript denotes an off-algorithm datum or a datum which for some reason is left unspecified. In this example, this notation will be used to denote the elements of the programs which the algorithm or abstract machine should execute (see section 4).

The letter w with a subscript is used loosely in this essay in order to build explanatory examples.

In the sequel, Ω will denote an empty set.

3. THE FILE PROCESSING ALGORITHM

Each *currently available* record is represented in the system as a list of seven items, namely

$$f_{comp}\ (7,\ w_{record},\ w_E,\ w_L,\ w_R,\ w_{PK},\ w_{CK},\ w_{FK})$$

where w_{record} is the record represented as list of the contents of its fields, w_E, w_L and w_R are values of the existence indicator and the left and right derivative at this record of the file to which it belongs, respectively, while w_{PK}, w_{CK} and w_{FK} are the keys of the preceding, current and following record of the file involved, respectively. In the sequel, the key constructing function f_{key} of one argument (a record) will be supposed unique for all files, for simplicity. We shall also assume that the last exclusive records of all files will consist of the literal EOF.

Let

$$f_{min} = f_{min\ 1}(-, -, 1, x_1(f_{el}(1, x_2)), f_{num}(x_2))$$

and

$$f_{min\ 1} = f_{cond}((x_3 = x_5) \rightarrow x_4, T \rightarrow f_{min\ 1}(-, -, (-) + 1, f_{cond}((x_4 < x_1(f_{el}(x_3, x_2))) \rightarrow x_4, T \rightarrow f_{el}(x_3, x_2))))$$

Then, f_{min} is a function of order two, which computes the minimum value of the function x_1 of the elements of the list x_2.

Let

$$f_{input\ 1} \equiv f_{input\ 2}(-, -, (f_{min}(f_{key}(y_1), x_3) \neq x_5), 1, \Omega, x_3, x_4, T, (f_{key}(f_{el}(1, x_3)) \neq x_1) \vee (f_{el}(2, f_{el}(1, x_2))), x_6)$$

With reference to (16, section 3), x_1 will denote the contents of the *current key register* (CKR), x_2 the complete set of the records available at the previous *pulse* (the dimensions of x_2 thus being the ones of a list of lists of elements of unspecified dimensions), x_3 and x_4 denote the contents of the lower and upper logical one-record buffers (required to compute the right derivatives), x_5 is the value that the current key register had at the previous pulse and x_6 is the list of all input files.

The function $f_{input\ 2}$ called above is defined as

$$f_{input\ 2} \equiv f_{cond}((x_4 > f_{num}(x_{10})) \rightarrow f_{comp}(10, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}), T \rightarrow f_{input\ 2}(-, -, -, (-) + 1, f_{comp}(x_4, f_{dec}(x_5), f_{cond}(z_2 \rightarrow f_{comp}(7, stop, T, U, U, U, U, U), T \rightarrow f_{cond}(x_9 \rightarrow z_1, T \rightarrow f_{comp}(7, f_{el}(x_4, x_6),$$

$$T, \neg f_{el}(2, f_{el}(x_4, x_2)), x_1 = f_{key}(f_{el}(x_4, x_7)),$$
$$f_{el}(6, f_{el}(x_4, x_2)),$$
$$f_{el}(7, f_{el}(x_4, x_2)), f_{key}(f_{el}(x_4, x_7))))))), f_{cond}$$
$$(7, x_9 \rightarrow f_{el}$$
$$(x_4, x_7), T \rightarrow -), f_{cond}(x_9$$
$$\rightarrow -, T \rightarrow f_{physical\ input}(f_{el}(x_4, x_{10}))), f_{cond}$$
$$((f_{kev}(f_{el}(x_4, x_6)) = x_1) \vee f_{el}(2, f_{el}(x_4, x_2))$$
$$\rightarrow F, T \rightarrow -), (f_{key}(f_{el}(x_4, x_6))$$
$$\neq x_1) \vee ((f_{el}(2, f_{el}(x_4 + 1, x_2))) \wedge (\neg x_8)), -))$$

where

$$z_1 \equiv f_{cond}(x_3 \rightarrow f_{el}(x_4, x_2, T \rightarrow f_{comp}(7, f_{first}(1, f_{el}(x_4, x_2)),$$
$$F, f_{last}(5, f_{el}(x_4, x_2)))))$$

and

$$z_2 \equiv (f_{el}(x_4, x_6) = EOF)$$

Here, x_1 is the contents of CKR, x_2 is the set of the available records at the time of the preceding pulse, x_3 is a signal which is on during the first pulse of each phase [see (7), (16)], x_4 is a standard recursion count, x_5 is the part of the machine status elaborated until the previous incarnation of $f_{input\ 2}$, inclusive, x_6 and x_7 are the contents of the lower and upper buffer, respectively, x_8 is a signal which triggers the entrance of a record which clobbers another one belonging to the same file and having the same key [see (16), pages 152-153], x_9 is a signal which summarizes the conditions under which a new record should not be entered, and x_{10} denotes the set of all input files.

The value of both $f_{input\ 2}$ and $f_{input\ 1}$ is a list of ten elements, of which the fifth is the set of records available to the pulse of which the evaluation of $f_{input\ 1}$ is the start (16, section 3).

We shall utilize the auxiliary function f_{stop} defined by

$$f_{stop} \equiv f_{cond}(\neg f_{el}(x_2, x_1) \rightarrow F, T \rightarrow f_{cond}((x_2 > f_{numb}(x_1))$$
$$\rightarrow T, T \rightarrow f_{stop}(x, x_{2+1})))$$

where x_1 is a list and x_2 an internal count. Furthermore, we shall need

$$f_{\text{list }1} \equiv f_{\text{list }2}(-, -, 1, x_2(1))$$

where x_1 in an integer, x_2 a function of an integer variable, and

$$f_{\text{list }2} \equiv f_{\text{cond}}((x_3 > x_1) \rightarrow f_{\text{dec}}(x_4),\ T \rightarrow f_{\text{list }2}(-, -, (-) + 1, f_{\text{comp}}(x_3, f_{\text{dec}}(x_4), x_2(x_3))))$$

where x_1 is the number of recursions, x_2 is a function of an integer, x_3 an internal count and x_4 the issue of the preceding incarnation, and

$$f_{\text{file }1} \equiv f_{\text{file }2}(x_1, 1, f_{\text{cond}}(f_{\text{el}}(1, f_{\text{el}}(1, x_1))(x_3, x_4, x_5)) \rightarrow f_{\text{physical file}}(x_2, f_{\text{el}}(2, f_{\text{el}}(1, x_1))(x_3, x_4, x_5)),\ T \rightarrow x_2), x_2, x_3, x_4, x_5)$$

where x_1 is the group of FCE related to the current incarnation of f_1, x_2 is the output file involved, x_3 is the list of available records, x_4 the list of internal variables, x_5 the stop indicator (16) and

$$f_{\text{file}\,2} \equiv f_{\text{cond}}((x_2 > f_{\text{num}}(x_1)) \rightarrow x_4,\ T \rightarrow f_{\text{file }2}(-, (-) + 1, f_{\text{cond}}(f_{\text{el}}(1, f_{\text{el}}(x_2, x_1))(x_5, x_6, x_7)) \rightarrow f_{\text{physical file}}(x_4, f_{\text{el}}(2, f_{\text{el}}(x_2, x_1))(x_5, x_6)), T \rightarrow x_4), -, -, -, -))$$

where x_1 has the same connotation as for $f_{\text{file }1}$, x_2 is a count, x_3 is the main issue of the preceding incarnation, x_4 the output file involved, and x_5, x_6 and x_7 have the same meaning that x_3, x_4 and x_5, respectively, had in the definition of $f_{\text{file }1}$.

The definition of the last two functions requires that the file control predicates (16) are organized in the form of a list of as many elements as are the output files, where the i^{th} element is the list of all the file control predicates related to the i^{th} output file. Furthermore, each such predicate should be presented in a list of two elements, of which the first is the flow control predicate as defined in (16) and the second is a form whose value is the record to be filed, represented as list of the contents of its fields.

The basic function used to compute the contents of the records and of the output files, that is, to control the output, is

$$
\begin{aligned}
f_{\text{output }2} \equiv\ & f_{\text{cond}}\,(x_4 \rightarrow x_5,\ T \rightarrow f_{\text{output }2}\,(f_{\text{input }1}\,(f_{\min}\,(f_{\text{key}}\,(y_1),\\
& \quad \text{el}\,(6,\ x_1)),\\
& f_{\text{el}}\,(5,\ x_1),\ f_{\text{el}}\,(6,\ x_1)\ f_{\text{el}}\,(7,\ x_1),\ x_8),\\
& f_{\text{comp}}\,(f_{\text{list }1}\,(f_{\text{num}}\,(x_8),\ f_{\text{el}}\,(y_1,\ x_8))),\\
& f_{\min}\,(f_{\text{key}}\,(y_1),\ f_{\text{el}}\,(6,\ x_1)),\\
& f_{\text{stop}}\,(f_{\text{el}}\,(5,\ x_1),\ 1),\\
& f_{\text{comp}}\,(f_{\text{num}}\,(x_5),\ f_{\text{list}}\ (f_{\text{num}}\,(x_5),\\
& f_{\text{file }1}\,(f_{\text{el}}\,(y_1,\ x_7),\ f_{\text{el}}\,(y_1,\ x_5),\ f_{\text{el}}\,(5,\ x_1),\\
& \quad x_2,\ x_4))),\ x_6,\ x_7,\ x_8))
\end{aligned}
$$

where x_1 is the value of $f_{\text{input }1}$, x_2 is the list of the values of the internal variables at the previous incarnation, x_3 is (CKR), x_4 an internal signal which denotes the occurrence of the last pulse, x_5 is the list of the output files at the previous incarnation, x_6 is the list of forms which assign values to the temporary variables, x_7 is the list of the flow control predicates (16) and x_8 is the list of the input files.

In (16) the concept of pulse as atomic file processing action was presented and discussed. With the present algorithm we can give a simple formal definition of this basic concept of file processing: *a pulse is an incarnation of* $f_{\text{output }2}$.

The general controlling function of this algorithm, which, if we had adopted a tree structure rather than a machine-oriented linear notation, would be at the root, is

$$
\begin{aligned}
f_{\text{output }1} \equiv\ & f_{\text{output }2}\,(z_3,\ x_3,\ O,\ F,\ f_{\text{list}}\ (f_{\text{el}}\,(1,\ x_2)\ f_{\text{physical file}}\\
& (f_{\text{el}}\,(y_1,\ x_2),\ f_{\text{el}}\,(y_1,\ x_6))),\ x_2,\ x_4,\ x_5,\ x_1)
\end{aligned}
$$

where x_1 and x_2 are lists of input and output files, respectively, x_3 and x_4 are the list of the initial values and of the forms defining the temporary variables, respectively, x_5 is the list of the flow control predicates with the corresponding record value associated (see above in this section), x_6 is the list of the contents of the labels of the output files (which, in this algorithm, are assumed independent of the input), and

$$z_3 \equiv f_{\text{list}}\ (f_{\text{el}}\,(1,\ x_1),\ f_{\text{comp}}\,(7,\ U,\ F,\ U,\ U,\ U,\ O,\ O))$$

is the initial machine status.

Remark: The usage of $f_{output\ 1}$ made here is to a certain extent similar to the one of a special purpose supervisory control or monitor in systems programming for conventional, command-structured algorithms or computers. In the case of declarative representation there is no difference between monitors and other functions, since each function monitors the incarnation of those on the basis of which it is defined, and each function which is used as monitor by being put at the root or lowest level in an algorithm can also be on a branch of other algorithms.

4. THE PROGRAMMING LANGUAGE

The implementation of the above algorithm on a computer with wired or simulated stack logic (5) able to evaluate on either wired or programmed basis all apodictic functions mentioned in section 2 consists of writing the functions and shorthands defined in section 3 into its memory or of preparing microprogramming plans consisting of their definitions. A program for this new computer consists of a function definition of the type

$$f_{some\ process} \equiv f_{output\ 1}(v_1, v_2, v_3, v_4, v_5, v_6) \qquad (4.1)$$

while its operation consists of letting it evaluate such function. The meanings of the v_i are the following:

- v_1: list of the names of the input files
- v_2: list of the names of the output files
- v_3: list of the initial values of the intermediate variables
- v_4: list of the forms which assign values to the intermediate variables
- v_5: list of lists of flow control predicates and record evaluation forms
- v_6: list of labels of the output files

The programmer of this algorithm is supposed to write the v_i according to the morphological rules of the system. For example, if he is programming a process involving five intermediate variables, the initial values of the first four of them being 1, $f_{some\ function}$, T and BUMBLEBEE, respectively, while the last one is initially not defined, then, at the place of v_3 he should write

$$f_{comp}(5,\ 1,\ f_{some\ function}, T,\ \text{BUMBLEBEE},\ U)$$

For the design of v_1, v_2, v_4, v_5 and v_6 analogous rules should be

followed. Regarding the lists, v_4 and v_5, these items stand for forms in three bound variables, namely the list of the available records, the list of internal variables, and the STOP indicator. They will be denoted y_1, y_2, and y_3, respectively. Reference to elements of such variables while preparing the lists v_4 and v_5 should be made considering this. For example, the contents of the third field of the fifth input file will be denoted
$f_{el}(3, f_{el}(1, f_{el}(5, y_1)))$,
while the right derivative of the same file is denoted
$f_{el}(4, f_{el}(5, y_1))$.
The sixth internal variable is denoted $f_{el}(6, y_2)$.

The notation rules of the preceding paragraph hold for machine language programming. However, it is easy to write a program for mnemonic translation, in order to enable the programmer to use a notation for fields, variables and indicators (7, 16), similar to the one of the Algebraic Data System Language. It would be an interesting exercise to write such a program in a declarative language similar to the one used in this paper.

In the program (4.1), the function $f_{\text{some process}}$ has order zero. Its function letter plays a role rather similar to the one played by program-identification cards in conventional programming.

CONCLUSION

The main direct advantage of this approach to file processing is the extreme simplicity of the programming language, which depends on the fact that the representation of any file process is independent of the procedure by which it is carried out. Despite the fact that the main field of application of file processing techniques, which are machine accounting and linguistics, involve little mathematics, the design of this algorithm or similar ones requires some applications of formal logic. In other words, the alleged non-mathematical nature of file processes is shown to depend only on the conventional procedural approach to them, while this method, based on the analysis of the inherent logical structure of such processes, allows for fruitful applications of mathematics to their representation. While conventional algorithm and computer design involves the application of nothing beyond than propositional logic, the declarative approach based on recursion requires first order functional calculus. So far, this author has not yet met an algorithm design problem where higher order functional logic

is needed. The need for specialized mathematical tools yielded by this approach affects only the design of algorithms. In contrast, no knowledge beyond elementary Boolean operations is required in production programming, where the level of skill that the programmer should have is considerably lower than the one of a conventional business programmer, such as, for example, a COBOL programmer.

REFERENCES

1. Goldstine, H. H. and von Neumann, J., Planning and coding of problems for an electronic computing instrument, Institute for Advanced Study, Princeton, N. J., 1947.
2. McCarthy, J., A basis for a mathematical theory of computation, paper 5.3, Proc. 1961 W.J.C.C., Los Angeles, Cal., (1961).
3. McCarthy, J., Recursive functions of symbolic expressions and their computation by machine, Comm. Assoc. Comput. Mach., 3, 4 (1960), 184-195.
4. Gilmore, P. C., An abstract computer with a LISP-like machine language without the label operator, (to appear).
5. Pacelli, M., Tecniche di traduzione automatica, Alti del convegno sui linguaggi simbolici, Pisa, January 1962.
6. Church, A., Introduction to mathematical logic, Princeton Univ. Press, Princeton, N. J., (1956).
7. Lombardi, L. A., Theory of files, Proc. 1960 E.J.C.C., paper 3.3., New York, N. Y., 1960.
8. Lombardi, L. A., System handling of functional operators, J. Assoc. Comput. Mach., 8, 2 (1961), 168-185.
9. Lombardi, L. A., Inexpensive punched card equipment, J. Machine Accounting, 12, 8 (1961), 11-18.
10. Lombardi, L. A., Logic of automation of system communications. J. Machine Accounting, 13, 4 (1962), 18-28.
11. Lombardi, L. A., On a problem of punched tape-to-card conversion. J. Machine Accounting, 13, 5 (1962), 31-32.
12. Lombardi, L. A., Nonprocedural data system languages, (invited paper), Proc. 16th. National Conference of A.C.M., Los Angeles, Calif., (1961).
13. Lombardi, L. A., On table operating algorithms, Proc. 2nd. IFIPS Congress, München, Germany, August 1962 (to appear).
15. Lombardi, L. A., Zwei Beiträge zur Morphologie und Syntax deklarativer Systemsprachen, DMV-GAMM Tagung, Bonn, Germany, April 1962.
16. Lombardi, L. A., Mathematical Structure of Nonarithmetic Data Processing Procedures, J. Assoc. Comput. Mach., 9, 1 (1962), 136-159.

PANEL DISCUSSION
Languages for Aiding Compiler Writing

Moderator :	H. D. Huskey	(US)
Members :	R. A. Brooker	(UK)
	J. M. Foster	(UK)
	P. Ingerman	(US)
	J. H. Wegstein	(US)

HUSKEY (U.S.)—I ask the members of the panel to give a five minutes discussion of what they are interested in this particular topic.

WEGSTEIN (U.S.)—As you all know Algol 60 is rather incomplete when it comes to string manipulations, symbol manipulations and character shuffling. Several languages have been invented to do these sorts of things, like the Jovial language. We have published a paper which gives a string language in the January *A.C.M. Communications*. The features of this are that one can take advantage of Algol 60 by adding a very few things to it. One of them is a string declaration. You declare a string thereby giving the string an identifier, and then you have string expressions such as A ⊕ B with the ⊕ symbol indicating a concatenation of the strings A and B, that is the characters in A are thought to have been set up later than the characters in B. There are string statements such as C is equal to A ⊕ B, and then the statement 'shift,' such as shift B ⊕ A right 1. Also, there is a possibility of declaring substrings, so if you indicate K is a string it also indicates substrings L and M. There is also the possibility of string erase, and the possibility of ordering the characters appearing within the string by using a rank decora tion. Now, basing on this, some examples were given in the paper and I want to call your particular attention to example 6 where you are translating from fully print precise algebraic expressions into a polish notation. Well, this language appears to be useful for translating and it has the advantage that it can be rather readily understood by anyone who understands Algol. This is the introduction that I wanted to make:

INGERMAN (U.S.)—I'd like to approach this from a slightly different point of view, and that is, the question of languages to aid compiler writing. I think it can be approached from the point of view of designing a language so that a person can easily write a compiler in it. Or, designing a language so that compilers can be written easily for it. Now, Mr. Wegstein has taken the first point of view, and I would prefer to take the second. I think that it is a productive one. Rather than worrying about specific languages in which it is convenient to write compilers, one should study the problem of designing languages in such a way that once a compiler has been written for one such language, it has been effectively written for all such languages. Now I have been very careful in the way I said that, so that I did not in any sense imply

—I here exclusively deny that I am implying—that I mean by this syntax oriented compilers. What I do mean, however, is that I think it is possible to design a way of specifying a language such that the specification is the structure needed for the design of a compiler, and if a reasonably rigorous way can be found to write such languages, then the problem of writing compilers *per se* is reduced essentially to an afternoon's exercise rather than the extensive effort that it now appears to take.

SCHWARTZ (U.S.)—The statement is that we have found that in writing a language which is in itself capable of being used to write compilers you have many additional gains. The problem of compilation is such that it is a fairly general problem, and once you can cover this class of problems you can cover many classes of problems, so that we feel there is a definite disadvantage to the first way of solving these problems.

HUSKEY (U.S.)—I will surely not argue with the point of view. I accept the point of view as well-taken, obviously.

GARWICK (Norway)—I personally have no sympathy whatsoever for the man who is a compiler writer or is writing general programs which might be used by many people. The only people we must have sympathy for are the ones who are going to use them. So, therefore, if you design a language, it is a language that you so want. Whether it is easy to specify and thereby implement it or not has no interest whatsoever to the user. I think that if we are able to specify it in such a way, so much for the better, but this is not the primary objective.

INGERMAN (U.S.)—I certainly did not intend to imply that the user of the language should be sacrificed on the altar of easier compiler writing, and if I standardized I am sorry. Clearly the person who uses a language is the person for whom a language is designed, but what I am asking to be done is to extend us one step further. I say, fine, if we are going to have people stating algorhithms then let us design a language like Algol; if we are going to have people writing compilers, then let us design a language for the purpose of writing compilers. Now, this is one level higher, but such a metalanguage should certainly not inflict itself on the user of the languages that the use itself describes. It should as a matter of fact never appear as far as the user of the language is concerned. If it is at all possible, and one hopes that it is possible, to design a language to describe this language

in such a way that writing a compiler becomes simple, then I think it should be done, and I think—as I said before—that it means simply applying the same principle of designing a language for the user to the poor fellow who, after all, has to write a compiler.

UNIDENTIFIED (U.S.)—It seems to me that one of the difficulties in writing a language to aid in writing a compiler is that it is not possible to predict exactly in which direction the language is going to go. Professor Gorn remarked yesterday that we would like to have multidimensional languages. Having just emerged from the process of writing a compiler for a two-dimensional language, I can say personally that I have not found any of the two available very useful for specifying the translation process from a two-dimensional language into a machine code.

BROOKER (U.K.)—I'd like to take the opportunity to refer to my lecture of this afternoon, giving an example of a compiler specifying syntax and semantics.

FOSTER (U.K.)—Mr. Ingerman and Mr. Garwick have said a good deal on what I thought to say and so I hope to be forgiven if I state it again in quite a different way. Algol 60 programs have two functions. They can be used to define or to mean a program, but they are also very convenient if they can be used as input to a compiler. Finally and similarly, a language describing a compiler is useful to say what the compiler is, to define a compiler; it is also very convenient if it is used at the same time to actually write the compiler. Now the emphasis of what people have said before has been on the writing of a compiler. I should like to put the emphasis on specifying the language, which seems to me to be a rather important task also.

WOODGER (U.K.)—I'd just like to draw attention to the thought that immediately followed after Mr. Foster's statement, that is that one should be careful as regards the subject matter for a language which is to describe precisely a compiler.

FOSTER (U.K.)—I wish I could speak about this from an experience, but what our policy has been is to say, when you have a new machine you can either write a compiler compiler, and you use this to write compilers, or you can write a compiler and you use this to write compiler compilers. The second of these has some advantages to the practical installation of a com-

piler itself. Compiler compilers are quite easily written in Algol whereas compilers are perhaps not so well described in Algol.

WEGNER (U.K.)—I was very interested in the controversy between Mr. Ingerman and Mr. Wegstein about the two ways of looking at the problem of languages aiding compiler writing and I'd just like to rephrase this controversy in case some different points of view might enlighten the matter.

In both cases we wish to describe a language in terms of another language. In particular, we have a source language, and we wish to describe the source language in some way. We can either write a regular compiler, which involves describing the source language in some machine-like language, and this—I think—would eventually resolve in describing the source language in the universal computer oriented language, something like polish notation (this would be a suitable kind of intermediate language), or alternatively we can describe the source language in terms of a metalanguage. This is the distinction in the way I see it. In the case of Algol, the metalanguage is a Backus normal form. In connection with translators and Gorn's terminology, the normal compiler approach seems to lead to a translator which is a processor, whereas the other approach seems to lead to a translator which is a recognizer in some sense. I am not exactly sure about what we can try, but that would be a starting point for some discussion.

Finally I would like to consider what these languages can supply with respect to the construction of computers. It seems to me that the machine language of the desired computer should eventually include this intermediate language in which terms we wish to write compilers, and possibly basic machine languages. This has in fact happened in the case of the KDF 9 where the machine language was in polish notation. Now the other point of view is also interesting. How would we go about designing a machine whose source language is Backus normal form? I think this is an interesting point to consider.

GORN (U.S.)—Referring to what Mr. Wegner has just said, I think we are deeply interested in the point of view expressed about the intermediate language. In one case the language is specified, and this includes Backus normal form, by a processor called "generator" which tells us how to construct words in the language. And, in the other case, the processor that you

need already to get the translation is the processor called the recognizer which has to be brought down to reading words of a language. Possibly the two points of view may be reasonably put together, let us say, if the language was specified by a recognition grammar instead of by a generator grammar in the first place, but it seems to me that whichever way a language is specified, there is always a problem. If you specify a language by a "generator" processor, or grammar, or whatever you want to call it, you have to be sure beforehand that you can recognize it, and you must have a very restrictive language generator to be so sure beforehand, and maybe you would not want such restrictions. On the other hand, if you have a language specified by a recognizer you may not know how the generator works. In either case there is a problem and I do not think we can get out of it easily.

INGERMAN (U.S.)—I would like to make a comment to Mr. Wegner's observations about Backus normal form. The impression that I have got from the panel discussion of yesterday is that Backus normal form *per se* has little or no semantic content, although possessing perhaps a great deal of syntactic content, and I would like to suggest that the language that I myself would like to have for describing languages to make my life easier writing compilers should also have some way of expressing its semantic content with something approximating the precision with which Backus normal form expresses syntactic content, knowing full well that this may be unrealizable.

HUSKEY (U.S.)—I have a short comment myself. Back in 1958, at the time of the Algol 58 effort, I encouraged some people in San Diego, California, to start a processor, and one of the principles that we stuck to there was that the system would be written in its own language, as far as possible. Actually the resulting language, which has been recorded as NELIAC, is essentially a subset of Algol, except for the addition of features to make it possible to handle characters in an input string and to assemble output commands for a specific computer. This effort has continued, so that there is a bundle of such translators: all of these will process themselves. Now, it is true that to handle these situations such as I described, you can revert to machine language in the respective systems, but the important characteristic here, which I did not hear mentioned, I think, so far in this symposium, is that the actual program is its own documen-

tation, which is a very important factor, and is always up-to-date, because whatever was assembled last to produce the most recent compiler gives a complete description of what the translator does. I think this is a very important feature that has not been mentioned. This is a simple language compared to Algol, which makes it possible for it to compile an extremely large program over a short time, so the standard method of operating is a load-and-go technique, and object programs are essentially never saved. Our binary corrections, for example, are never made, but all corrections are made in source language.

Now, the question I would like to ask of either the panelists or the audience is: why is not Algol, with corresponding extensions and terms of procedures, a satisfactory language to use in describing compilers? Would anyone like to comment upon that?

INGERMAN (U.S.)—I would like to ask a question right back, I think. That is, I observed that Wegstein's extensions to Algol make manipulation of strings visibly pretty, but it is not at all obvious to me, and something I am frankly curious about is: Why really, in order to do any of this manipulation, must you have something other than Boolean variables? Since ultimately you can express everything as a sequence of bits anyhow, it strikes me that any compiler which can minipulate Boolean variables can in effect manipulate sequences of bits, and I do not really see that you need anything more than this except for the admittedly useful purpose of being able to look at the program and figure out what in fact is going on.

WEGSTEIN (U.S.)—The unfortunate thing about Neliac and Jovial and M.A.D. and Mr. Booker's language is that if you want to sit down and compare the different methods of compiling, you have to learn each different language separately, and in some cases a textbook is required.

HUSKEY (U.S.)—I would suggest that if you are able to understand Algol, then the effort to understand Neliac is exceedingly small.

NAUR (DENMARK)—Talking about languages for compiler writing, we should first think of who is going to read this writing. There are two kinds of readers: there are the human readers and there are the machines. Now, the question of the human reader, I think, has quite some importance. There is considerable interest in being able to communicate what we have done

in our compilers. Personally, I consider that the most suitable language for 90% of what goes into a compiler is plain good English language, as long as the human reader is concerned. I claim that 90% of what is really interesting should be and can be, must really be, explained in good language. Then the rest will probably be some suitable mixture of tables describing logic. This is all for the human reader. Then for the machine: the machine, of course, is so different from the human reader that it really needs 10% of what the human reader will demand, in order to understand in its way. Of course there is this 10% left, part of it tables, part of it logic. I found that for this task Algol is an entirely suitable language with no additions whatever. We have the integer facilities handling entities for arbitrary strings, and I found no difficulty whatever in using it.

GARWICK (NORWAY)—Mr. Naur noted a very important point, that is: who should read it, and whereas of course it is very nice to have compilers, if you describe the compilers in the proper language, it is always uncertain if you produce efficient compilers this way. However, many of the processes you try to perform in compilers are very complicated. If you try to describe them in English language, it will be completely unreadable, it may be even worse than reading an Algol report, which certainly has a high standard in this respect—of readability, I mean. I myself worked on this sort of project. I found a lack of language in which to express myself. It is not that I expect to have it automatically compiled, but I just want to write down my thoughts. We cannot speak together without a language. It is very hard to get your thoughts on paper without a language in which to write them. So as to the language in which you write: I think English is very unsuited in most cases.

INGERMAN (U.S.)—I would like to interject a comment here. And that is: once again it seems that a great deal of time and effort has been spent holding the hands of the mathematicians who do not want to learn machine code or the management personnel who do not want to learn machine code, and it has been more and more borne out that very little time or effort has been spent in making life easy for the programmers who make life easy for other people.

BEMER (U.S.)—I would like to thank Dr. Huskey for bringing the topic around to what was advertised, namely languages for

aiding compiler writing. I think your comment was the first which brought us to the subject problem. Now, with respect to Naur's statement, I should like to ask how many Algol compilers in existence today are in fact written in Algol *per se*. Are there any known? I myself do not know of any. Mr. Ingerman may say that it is sufficient to manipulate bits by means of Boolean variables. As a matter of fact, this is in the micro structure and becomes almost impossible to make any sense. True, one must manipulate symbols, but there are symbols which are vastly more complex than the simple bits with which they are represented. Now, with respect to the difficulty of writing anything in, say, Neliac, M.A.D., Algol or other types which have similar manipulations and screen decomposition facilities, I would say it is not necessary that this be intelligible to the layman, because you people sitting here today are very few of the thousands that use computers, and probably not all of you actually write compilers in a production manner. In fact, we can only list, out of the entire world, surely not over a hundred actual compilers in existence that will go beyond, let us say, the test machines of the university. Therefore, it is quite proper, I think, for there to be, not exactly a code, but certainly a specialized jargon that the compiler writer needs to know, but which the user of the processor later on does not need to know. Therefore, it does not bother me that it takes a man a month to learn M.A.D. or Neliac, although I am sure it does not. All I worry about is, he can do a more efficient job than he could otherwise by using machine language. Furthermore, it is not necessary to have a compiler writing language, which in itself is a 100% efficient or does the entire job. This is a matter of economics, of logistics. If you cannot do most of the job, you can always go back to machine language. This is essentially an oversize crutch which we may use: we do not expect the job to be perfect the first time, but at least we have something which we can keep on using.

INGERMAN (U.S.)—I would like to make a small comment. One, I never said that manipulating bits was elegant. I remember I said it was enough. I do not think there is any argument on that score. Two, although it is perhaps appropriate at the moment to consider specification languages to be nothing more than a crutch, I rather think that it is a direction that had best be looked into, because, as fast as new fields develop, and as fast as new types of problems arise, there will be demands for new compil-

ers, and if the compiler writers want to keep up with the demand for their services, I think they are going to have to find some way of doing this more efficiently, with respect to their own time, than is now done.

HUSKEY (U.S.)—Are there any further comments? I think some people are describing their compilers in Algol and so forth. Maybe there is no definitive answer to the questions. Has anybody something to say on that?

DIJKSTRA (NETHERLANDS)—When we made our translator, we discussed how to do it for a couple of months, and we wrote down—we described what we were going to do in Algol, which is for two purposes: to get used to using the language and furthermore because we were convinced that it was very convenient. I must admit that we introduced some additional things which are outside the language, but it did not bother us at all, because it was a documentation, and the translator itself afterwards was written in machine code. So it was not so difficult. Now, in standing here, I should like to make an open remark, which more or less is a sequel to the discussion between Mr. Ingerman and Mr. Garwick. The subject of this panel discussion is languages for aiding compiler writing, and I am really asking myself whether this task is sufficiently important to define languages for. I have a feeling that the better a language is defined, the more obvious it is how you must make your translator. It is not a difficulty at all. Difficulties arise when the language is not too well defined. So, I have a feeling that future languages will be better defined both for the benefit of the user who has to master the language and consequently for the benefit of the translator-maker. So, as far as recognition, etc., is concerned, I think that the task of writing compilers will be reduced considerably. Now, the curious thing is, as we have seen, and you can see it in every journal, that at the present moment manufacturers are able to derive an advertisement from the complexity of their translators, their compilers. This, of course, is ridiculous, because you can invert the statement and say that, if it needs a very large, a very complicated translator then the conclusion is that the machine in question is not too well suited for its task. And, I have a feeling, and a strong hope, that machines will come which, by their very structure, are more adapted to the execution of the object program; and then the second need for making long translators also is lost.

INGERMAN (U.S.)—I would like to ask one question, if I might. It appears to me that Dijkstra's comment about languages being more carefully defined, and that the more careful the definition, the easier it is to write compilers is perhaps on the other side of the fence, in the sense that the language in which the definition takes place is precisely, I think, the language that we have been talking about. I gather that Dijkstra would prefer a natural language rather than a rigorous and formal language, but this is a matter of taste and is not to be disputed. Still, the point I would like to make is that after all the language in which the definition takes place is in itself the language we have been looking for.

STRACHEY (U.K.)—It seems to me that a great deal of time has been spent in discussing the languages in which syntax and various other things—the input to a compiler—are to be written, and it is certainly true that if your syntactic language is sufficiently clear, and the language you want to translate is sufficiently precisely defined, then the analysis section of your compiler will be easier to plan, and presumably therefore easier to write. But in any reasonably competent compiler there is a great deal more than syntactic and semantic analysis. Your job is first of all to take the input string and to analyze it into its parts; then to perform various manipulations on these, in order to analyze them more carefully, to attach meanings to the various symbols you have, and then, finally, to transmit these up to the instruction repertory that you have for the machine for which you are compiling. Now, these last two parts of the operation are very difficult—I mean, they are a very large part of the work, and the specification language gives no assistance at all in this. In particular, I have not seen any sensible approach to match the structure that you get in the machines, for the logical structure of the source program, to the available set of instructions which you have in order to try and get an efficient machine run. And this brings me to the other very important feature about compilers. The thing that you want to hear about most when you hear about somebody's compiler is what is the environment in which the object program is going to run. Now, that is a rather vague statement, but when you compile a program, the resulting object program is to some extent a limitation on the abilities of the natural computer. For instance, you use a conventional linkage for going in and out of procedures and things like this; these are constructed by the compiler. You may use a push-down list for

communication to the various parts of the machine and the description of a compiler shows a very complicated structure superimposed on the structure of the machine. And this is what I call the environment in which the object program has to run. Now, this environment is by no means obvious, is by no means unique and it is at the choice of the compiler designer. The nature of this environment will influence very considerably your judgment on the various values that you put on features to the language. For example, if your machine is a machine in which floating point arithmetic is not a built-in hardware feature, so that this has to be done by subroutines and therefore there are interpreted subroutines, then your judgments of the cost of going in and out of fairly complicated procedures for entering blocks in Algol, will be different—because of the difference between the length of time in doing an arithmetic operation and a red tape operation—from a machine which has a built-in floating point arithmetic; in which case it becomes very expensive to put in an extra call of subroutines. This sort of thing which forms what I call the environment is of utmost importance when writing compilers. It is very difficult to describe in anything except, I think, English, and it seems to me that it is one of the most important features to think about when talking about compilers and having the languages. I do not think you can talk about a language for helping compiler writers, but its ideas and concepts, and if this comes forward we have a language to express them. And I do not think at the moment we have a very clear idea of the possible concepts of environments in which you can compile things.

WILKES (U.K.)—There are two comments I have, one on Mr. Strachey's comments in connection with the environment of a compiler. I believe that Cobol and other commercial languages do have facilities for environment description, which specify the characteristics of the computer, such as memory size, number of tapes, and other characteristics, so that this seems to have been developed, and I wonder whether Mr. Strachey does mean this kind of thing when he speaks of environment, or something else.

The other point that I will take up was one of Dijkstra's. He has indicated, and with good reason, that compiler writing may well become trivial, and is becoming trivial, and he has illustrated this very brilliantly in his compiler. But I feel this is only settled when we have the appropriate concepts in terms of

which to describe the compiler, and, in fact, these appropriate concepts in terms of which we describe the compiler—and due to these concepts compiler writing becomes trivial—these concepts form a language, namely the language which aids us in compiler writing. Now, Dijkstra has reduced the writing of Algol compiler to a triviality, in terms of his notations, but I feel this is partly because he has developed an appropriate set of subroutines or what I would describe as a language in terms of which to consider his translations, and I wonder whether he would comment on whether he considers his set of basic concepts in terms of which to describe the compiler as a language which aids in compiler writing.

DIJKSTRA (NETHERLANDS)—The greatest part of the work, indeed, has been thinking and finding out the ideal structure of the object program, that is, designing the machine, the hypothetical machine which was nice for carrying the object program, which was nice to translate into. Our subject program, in fact, plays around with 100 subroutines tailored to the task. I always regarded this trick as designing a new machine and having this machine imitated by the one built. On the other hand, you can regard a machine also as a language, rather than a language as a machine, so I must state to Mr. Wilkes that if you wish to regard this as the language, O.K., that is all right.

With respect to the remarks made by Mr. Strachey who told that things matter greatly, for instance when you have a machine with floating point arithmetic, or not, indeed it greatly matters. Well, in this direction my hope goes that for future machines, translation will become trivial, amongst other things because structure may be such that we will try to optimize it if it does not work right.

NAUR (DENMARK)—I found that Mr. Strachey said things which were extremely well said and very correct. This involves a question. We have found indeed that it is the most central thing, really. Probably about three times as much work is taken over by this first and decidedly most important part, design of the compiler, while the design of the symbol manipulation is really a very trivial matter, comparatively speaking. So the whole thing of compiler writing, as far as our experience goes, is fitting whatever machine is presented to you with the requirements of your language, and sometimes this is a very painful process, unfortunately.

And then of course I agree with Dijkstra who said that what we may hope is that machines will be designed in the future so that this mismatch will be avoided. On the other hand, it is not surprising that we have a mismatch here. Machines, as we know them today, have been almost exclusively designed with machine coding in view. Why then do we expect that they should be well fitted for an entirely different manner of coding. This is not to be expected at all. But it is already happening in some places. Let us hope that there will be a very much wider movement to design machines which will do really the job we want them to do, and not to present us other problems as soon as they are finished.

CAMION (BELGIQUE)—Les traducteurs sont en général écrits avec un agencement de mémoire en liste. Donc ce qui est nécessaire c'est un manipulateur de listes, c'est à dire un langage qui nous permette d'écrire facilement nos listes, qui nous permette de consulter des listes. Je pense que c'est dans cette direction qu'il faut chercher. Au lieu de traducteurs écrits de façon à traiter facilement le problème numérique, il faut plutôt chercher dans la direction des manipulateurs de listes. J'ai écrit un langage qui permet de traiter des listes. Pour écrire ce langage j'ai donc eu besoin d'écrire un compilateur et j'ai écrit ce compilateur dans ce langage. Evidemment c'est un compilateur assez simple mais l'écriture de ce compilateur dans ce langage n'a pas présenté de difficultés particulières. Ce langage n'est pas encore terminé, il est en cours de texte maintenant et je pense que c'est dans ce sens-là, plutôt que de chercher un problème qui traite facilement l'expression booléenne ou autre chose, qu'il convient de chercher un problème qui traite des problèmes non numériques. Le traducteur que j'ai écrit est spécialement conçu pour traiter des problèmes comme la traduction de langues, les graphes, les manipulations de formules, et c'est dans ce sens-là qu'il faut chercher, je crois.

STRACHEY (U.K.)—I certainly agree with what has been said that list processing languages are extremely useful in the compiler process, and I think indeed that lists themselves or derivations from them are a very useful thing to use in this stage of proceedings. I think that one of the things about the environment of compiler which is worth mentioning, is not only the environment of the object program at run time but a fitting in of your compiler into the system in which your whole computer is being run. This is a much larger problem than that of a compil-

er being put together. As far as running machines which have no floating point arithmetic, I think it would be obviously very useful to have machines which have pushdown lists as easily as computational facilities, etc. But, there are general problems which I do not think are solved yet. I mean problems like storage allocation, that is to say that most computers acquire their programs in consecutive stores and a great deal of these list handling operations give you non-consecutive storage of operands at the end of it, and a very important point is that address modification requires in any simple scheme the use of consecutive storage. One of the advantages of having your list as it works in consecutive program locations, is that you can use a modification for going straight to the sixteenth member of the list, instead of going on sixteen times with the threaded list. And, I do not know what is the solution to this sort of problem and what the solution will be. It is the greatest problem in machine design and in logical design, to design what is the best sort of thing to deal with this very difficult storage allocation. And one problem which you have to face unfortunately, at any rate in England, is that most of the machines available have far too small a core store to make most of these operations at all feasible, or at least at all easy. A very strong tendency of manufacturers is to say "we can't base our compilers on stores of more than 2000 words, or whatever it is, because we can't persuade anybody about it." This is a very different story from machines which have at least 30,000 words in store. If you have a compiler with a limited store, and particularly one which runs at a run time on a store with a machine which has a backing store, drums, tapes, or something like this, it becomes a very difficult problem indeed to make your translation. And most of my complaint with the present day problem languages is that they are fed with so much information that you cannot do in fact the translation mechanically, there being too much unnecessary information.

WILKES (U.K.)—Mr. Naur remarked just now that many machines in existence have been designed with a view to programming in machine code. And he looks forward to the time when machines may be designed with a view to automatic programming. And he says that it would be surprising if the requirements would be the same. Well, it seems to me that it is surprising and if you look at the matter from the point of view of a machine designer you do not in fact find that the requirements of a machine

to be programmed in machine language and the machine to be programmed automatically are different. But if you think of something which is good for a compiler, it is good for a human being as well. There is a number of rather small points of difference you can make but I do not think what you have said is true. I think that the position with regard to the compiler for writing compilers is different, because the writing of compilers is essentially a symbol manipulation and there is—I think—a future for design of machines which can do list processing very rapidly, and this may help not only with the compiling of compilers but also may help us to secure compilers that are in fact rapid. So at that level, I think, there is some hope for us by way of improvement in machine design, but I think it would be a mistake to put too great a hope on getting machines that adapt in some magic way for automatic programming.

INGERMAN (U.S.)—I'd like to make a comment here, and that is that I won't argue on either side of the point that there is a difference or not a difference between machine design for coding in a machine language, and machine design for coding in a compiler language. But it has been my experience that many machines have been designed for the convenience of the machine designer.

WEIZENBAUM (U.S.)—I quite agree with your point of view. I think that if we examine closely the way programming is done, then we see that obviously our people are very much influenced —in fact that is over-determining—by the appearance of the machine with which they have to deal. The fact that we do have four storage levels, for example, is extremely essential for the formation of a habit the programmer has come up with. I think it is entirely foreseeable that, say, 10 years from now no one will attempt to find the 16th item in a file. I think it is foreseeable in this connection that 10 years from now there will be programmers—at least in the United States we are maybe a little bit more luxuriant than elsewhere—who will refuse to work on machines which don't have features which make the kind of program they are working on possible, as we have today in the United States programmers who refuse to work on machines which have latency problems.

HUSKEY (U.S.)—That was very interesting. Is there any question on this?

VAN DER POEL (NETHERLANDS)—I just wanted to go into the question of the appropriate code for serving as an object for Algol. We have quite a bit of experience in this because we have created within our machines an interpretative language which serves as the object language for translating Algol into it, and this has given us a little bit of insight into the requirements of the object language. There are very few requirements of the machine code. The machine code is not so ill-suited for object code at all. The second thing which has given us a little bit of insight is that our machines, in this particular instance, have an accessible microcode so that some of the required operations, for example change searching and list searching, could be incorporated very easily, and this is one of the things which are most required. I still think that the machine code, which we usually work with, is not ill-suited for serving as the object code at all. I should also like to add that we have described our Algol translator fully in Algol itself, with of course some string handling and chopping, but in a normal pure Algol language. So Algol can serve very well as the descriptive language of the translator itself.

BEMER (U.S.)—I should like to make a proposal. We seem to have trained up an Algol processor written in Algol, with of course a few minor modifications with string handling as everyone seems to require. It seems to me that there are a number of computers in the world today for which the people are writing Algol processors and some for which they do not exist. I should like to suggest, as Van Der Poel did, that a program for translation of Algol into basic Algol itself is written in Algol for everyone, in which case we can test this item on a variety of machines and people who do have Algol processors with them now could have one immediately.

VAN DER POEL (NETHERLANDS)—Perhaps I may immediately answer the question of Mr. Bemer. The description we are producing is a full description of the Algol translator and is essentially machine free. I say 'essentially.' Now, there are some minor exceptions but they are not essential at all. So, it is a machine free description which can be applied to any computer. We intend to publish it in a very short time.

KOGON (U.S.)—I would like to answer the final remark by Mr. Van der Poel and at the same time call some attention to the

fact that maybe a little more humility is in order because a processor is after all what gives the program, and if it is possible to use the compiler, say, to write an algorithm for the inversion of a matrix, if this is a good compiler at all, it should also be suitable to write an algorithm for the transformation of one thing into another. I have done this myself and I don't see any reason for many of these discussions here, for many objections. The discussions—I think—could be a little more optimistic because many languages are indeed suitable to write compilers for writing compilers, and this—I think—is the purpose of this Session, or maybe I am mistaken.

In particular, in Algol, with a few minor additions, it is possible to describe compilers. These minor additions, let us say, are the ability to have indirect addressing, which is not in the language, and to compute names. Once one has done so in Algol, one has a language to describe a compiler, and if you have tried that, you are satisfied because it is possible and this is relatively easy for anyone who tries. This is my personal opinion.

HUSKEY (U.S.)—I'd like to say that as there are no other questions this Session is over.

CONSTRUCTION OF PROCESSORS FOR SYNTACTICALLY HIGHLY STRUCTURIZED LANGUAGES

THE ALCOR PROJECT

K. Samelson and *F. L. Bauer*

Gutenberg University, Mainz, Germany

ALCOR (meaning ALgol COnverteR) is the name of a cooperative group of computing centers, and computer manufacturers. The members of this group have agreed to use the same restricted version of ALGOL, based on essentially the same methods of translation. For coding, a single hardware representation (5 channel paper tape) is used. In this way exchangeability of programs for different machines is insured.

The history of the ALCOR group is intimately connected with the development of ALGOL. In a certain sense both could be said to begin with Rutishauser's paper on automatic programming (9) first published in 1951, which for the first time gave a systematic, although somewhat complicated, approach to formula translation. It came too early, however, and remained unnoticed.*

Stimulated by Rutishauser's paper, the computer group of the TH Munich in late 1955 started studying formula translation. This led to the design, by Bauer and Samelson (1), of a formula controlled computer with a very simple and effective control mechanism.

A diagram of the simplest form of this machine is given in fig. 1. The input language, in ALGOL terminology, consists of simple arithmetic expressions (without exponentiation) with numbers only as operands, and with *right* hand side assignments like $(a+b) \times (-a+b) \Longrightarrow e$.

Characters are written on the keyboard 1 and produce coded signals to the predecoder 5. This separates numbers from operators $\times : + - ( )$.

Numbers are sent, as they come in, through numbers converter 7 into the numbers cellar 11. A cellar here is a store of the type which lately has been called pushdown-store or stack, working on the "last in first out" principle.

Operators go to operations converter 8. There, each character coming in from the predecoder is compared with the topmost element in operations cellar 12 (initially empty). The evaluation

*The same happened to two other papers on essentially the same subject which ought to be mentioned here: Lehmann 1953 (7), and Bohm 1954 (4).

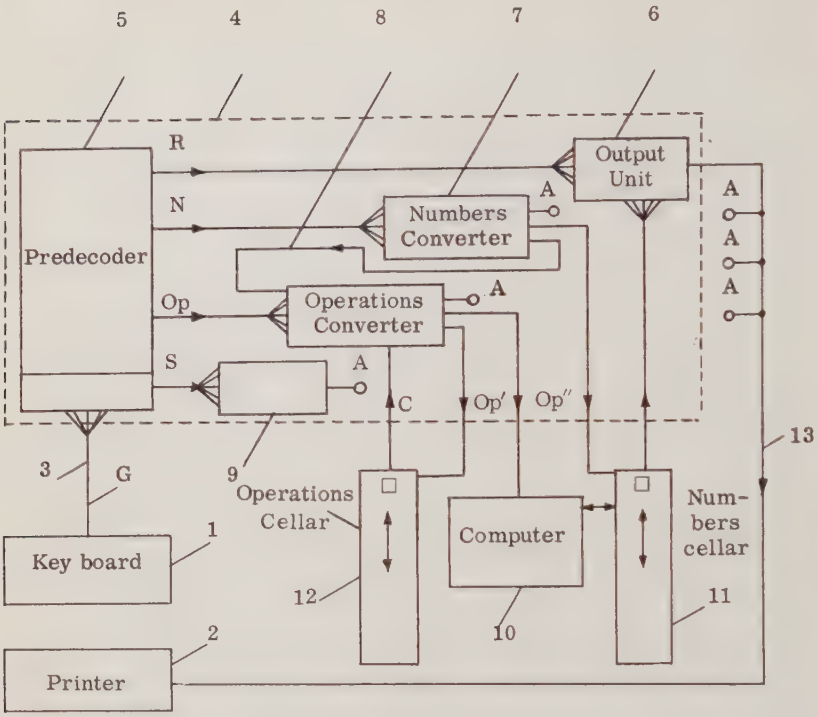


FIG. 1.

is done by means of a decoding matrix. The effects can be described in terms of precedence orders of the characters involved as listed below:

Order	incoming characters	cellar element
0	(	
1	× :	× :
2	+ −	+ −
3	)	(

The closing parenthesis never enters the cellar.

If the incoming character has lower order than the cellar element, it is simply pushed into the cellar. If the order is equal, the cellar element is transmitted to the computing unit 10 for execution, and is replaced in the cellar by the incoming character. If the incoming character has higher order than the cellar element, the latter is again transmitted to the computing unit for

execution. At the same time, contents of the operations cellar move up, and the process of comparison is repeated with the new cellar element. Closing and opening parenthesis simply cancel out upon meeting.

Execution of an operation means that the computing unit extracts the two top elements of the numbers cellar, executes the operation indicated with these operands, and returns the result to the numbers cellar.

An example is given in fig. 2. Here the top field gives the characters of a statement processed from left to right. The two fields below give the corresponding contents of operations cellar OC and numbers cellar NC. c and d denote the intermediate results, o denotes the number zero which is introduced at the beginning of expressions in order to take care of possible unary operators + or −.

The system described can be extended (and has been on paper) in quite a number of ways which we need not discuss in detail. The machine as designed immediately interpreted and evaluated formulae. However, it was clear from the beginning that the process could be adapted easily to generate programs simply by being applied to the names (addresses) of numbers instead of the numbers themselves, and producing instructions instead of executing operations, with the numbers cellar for intermediate results added to the generated program.

The basic principle remains unchanged: The characters of the program are sequentially compared to the cellar elements and moved into the cellar until a complete (explicit or implied) bracket structure is detected. This is extracted, processed, and replaced

S 1	S 2	S 3	S 4	S 5	S 6	S 7	S 8	S 9	S 10	S 11	S 12	S 13	S 14	S 15	S 16	
(	a	+	b	)		×	(	−	a	+	b	)		→	e	
⊗	(	(	+	+	(	⊗	×	(	−	−	+	+	(	×	⊗	⊗
	⊗	⊗	(	(	⊗		⊗	×	(	(	(	(	×	⊗		—OC
			⊗	⊗				⊗	×	×	×	×	⊗			
									⊗	⊗	⊗	⊗				
o	o	a	a	b	c	c	c	o	o	a	−a	b	d	d	e	o
⊗	⊗	⊗	⊗	a	⊗	⊗	⊗	c	c	o	c	−a	c	c	⊗	⊗
				⊗				⊗	⊗	c	⊗	c	⊗	⊗		— NC
										⊗		⊗				

FIG. 2.

by the result of the processing. Then the scanning is continued.

With this translation principle as a background, the ZMMD group (Zürich-Munich-Mainz-Darmstadt), the nucleus of the later ALCOR group, was set up in 1957 to develop a common language, translation philosophy and translation method.

The group was in agreement from the beginning, partly as a result of the experiment in machine design, to devise a language and translator system as an efficient programming tool using available machine facilities to the fullest extent possible. This ruled out any kind of interpretive system, in particular for floating point machines like PERM and ERMETH, and indicated a very judicious use of closed subsidiary subroutines. The target was, and always has been, an easy mathematically oriented language, and fast translation into efficient machine programs.

After publication of the first ALGOL report, in the fall of 58, three people of ZMMD institutes (Schwarz from Zürich, Seegmüller from Munich and Paul from Mainz) were brought together to program a first test model of ALGOL translators for their respective machines (ERMETH, PERM, Zuse Z 22). These first models were interpreters because programming was considered to be simpler. They were finished by the end of 58, were running in January 59, and were soon converted to program generators.

At the ICIP conference in Paris, June 1959, a report on the group's translation methods was given in the symposium on automatic programming (2). This marked the start of the expansion of ZMMD into ALCOR. Several computing centers (Regnecentralen Copenhagen, University of Bonn, Mailüfterl Vienna, Oak Ridge National Lab) showed interest, and decided to join ZMMD on a cooperative basis. They were given the available ZMMD material on translation methods, and began their own work on translators. The first manufacturer, Siemens & Halske AG, soon followed suit.

The cellar method outlined above for arithmetics proved to be suitable for ALGOL in general since practically all delimiters, apart from their operative meaning, implied a bracket structure amenable to the cellar treatment.

The essential feature of this method is, as mentioned in publications (2,11), that each in-coming character is matched with the topmost cellar element, and that the pair determines the ensuing action. The cellar system therefore can be considered to be a kind of automaton, where input (incoming character) and state (cellar element) determine new state and output. Originally a

decoding matrix, with incoming characters on the one side, cellar elements on the other side, as inputs, was used to describe the choice of action for each pair.

In the early days, especially when the figures on the size of the FORTRAN I translator, and its translation times on the fast IBM 704 became known, it was thought that on the relatively small machines at the disposal of the group decoding speed was essential, and a programmed decoding matrix or multiswitch seemed to be the fastest method. For this reason, and on account of its simplicity, ZMMD descriptions of the cellar method in 59 were still in terms of this matrix which had become rather outsized. The publications (2,11), however, which barely mentioned the matrix, should have made clear that the manner of decoding was a question of programming expediency for particular machines, and not a matter of principle.

In fact, all ALCOR members in 59 started looking for ways to reduce the matrix in size, with mostly similar results which only in part were published. First to be mentioned is A. A. Grau (6) who explicitly introduced the concept of syntactic states, and pointed out that the system could be considered to be a set of mutually recursive subroutines. Lucas (8) tried to deduce the translator mechanism from the syntactic structure. Paul and Petry independently introduced the direct use of the precedence order for decoding described at the beginning (unpublished) which can be extended to practically all operative delimiters with the precedence ordering shown in Table I.

The translator mechanism hitherto described (and including resolution of the for statement and conditional statement and expression by means of simpler ALGOL elements) serves to decompose ALGOL statements into a bracket free ordered sequence of instructions of fixed format (e.g. three address or single address instruction).

With a high degree of justification, we could call this the end of ALGOL translation. For the output of decomposition is a fully sequential language of simple macro-instructions. This language could serve as input language for an interpretive system or, alternatively, for a standard compiler system generating machine programs.

In both cases the processing system has nothing to do with ALGOL. However, in a translator designed to produce straight machine programs such a division, although simplifying programming by clearly separating tasks, would be uneconomical.

TABLE I

Order	Incoming character	Cellar element
0	(| [| := | if | for | goto | array | switch | procedure | value | '' | begin | segment begin |	
1	↑	↑
2	× | /	× | /
3	+ | −	+ | − | $-_1$
4	< | ≦ | = | ≠ | ≧ | >	< | ≦ | = | ≠ | ≧ | >
5	¬	¬
6	∧	∧
7	∨	∨
8	⊃	⊃
9	≡	≡
10	) |] | , | : | ; | then | else | step | until | while | do | end.	(| procedure call | standard function call |] | := | if | thenE | thenΣ | elseE | elseΣ | for | for:= | step | until | while | do | goto | begin | begin Δ | begin array | segment begin | array | array[| array: | switch | switch:= | procedure | type procedure | procedure(| type procedure(

Furthermore, what comes after decomposition makes up the bigger part of the detail work to be done for the individual machine. Therefore, we shall consider it briefly.

As mentioned before, operations required in the macroinstructions usually are not directly available in machines, and have to be translated into machine terms. This begins with arithmetic operations on different number types. Type handling very much depends on the machine, and within the ALCOR group different methods are used. The extremes probably are the

translator ALCOR Z 22 on one side, ALCOR S 2002 on the other side.

On the Zuse Z22, arithmetic operations must be done by subroutines. Therefore types are handled dynamically: every quantity stored carries a type characteristic, and the arithmetic subroutines evaluate this characteristic, carry out the operation accordingly, and attach the correct characteristic to the result.

The Siemens 2002 is a floating point machine, and real type arithmetics is naturally identified with floating point arithmetics. As for integers, after long discussions we came to the conclusion that for us they were most important as subscripts. Therefore they are internally identified with machine addresses which necessarily means fixed point numbers. Thus we have two different types of arithmetics which are incompatible, and type handling has to be completely static. This means that the type analysis has to be done completely during translation, with type transfer instructions in case of mixed arithmetic. As a consequence some restrictions, e.g. for actual parameters in procedure statements, had to be made. This we accepted since we considered the running time of the generated program to be far more important.

With respect to relations, and much more so with the Boolean operations, the situation is similar: in part at least they have to be represented by (open or closed) subroutines.

Next to decomposition the major task in ALGOL translation is the storage allocation as induced by the block structure. The block structure itself is a prime example of a bracket structure to be handled by the cellar method, both on the program and on the data storage side.

Conceptually this is very simple and elegant as long as the *own* is excluded: all the variables and arrays declared in each block are put on top of what is already present after the begin, and are taken out again after the corresponding end, and the numbers cellar for intermediate results is always at the momentary end of the sequence.

In practice, however, we have again the problem of static treatment (during translation) versus dynamic treatment (at run time). In a largely interpretive system, fully dynamic treatment as described by Dijkstra (5) is possible and very sensible.

The situation is different for a generating system. For dynamic treatment essentially means that data storage for each block is a relatively addressed sequence, with the point of reference dif-

ferent for each block, and determined by the running program. This can be handled effectively, if a sufficient number of index registers is available so that to each block a register of its own can be assigned. Otherwise, index registers would have to be simulated or shared, which is costly both in computing time and instruction storage, or the block data would have to be readdressed at each entry which also is rather tedious.

Therefore in our ALCOR translators two different data stacks were introduced: a static stack, comprising the simple variables and certain subsidiary quantities, which could be completely assigned and addressed at the end of translation, and a dynamic stack for arrays, (where some address calculation usually has to be done anyway) with a "free storage indicator" or stack pointer handed over from block to block. It should be mentioned here that this dynamic stacking of arrays was standard operating procedure in the subroutine organization of the ERMETH and on PERM since the mid fifties.

Handling of blocks immediately leads us to the delicate problem of how to handle procedures. Conceptually, dynamic stacking again is the immediate answer to the problem of data storage, since it means that at each procedure call, the momentary free storage is available to the procedure, and no data storage need be reserved. However, due to the admission of non-local quantities in procedures, we have the same problem of dynamical relative addressing as with blocks in general. Therefore it was decided, to assign static data storage to each procedure separately within the block containing the procedure, which of course rules out recursive procedures. The waste of static storage, in conflict with our original cellar principle, was considered regrettable.

There was no great concern, however, over recursive procedures. These recursive functions were originally introduced by McCarthy in order to formulate elegantly the theory of computability. They were considered by us to be of very little value in a language intended to describe actual rules of computation since they, in effect, do not describe recursive computation of the function value, but a recursive build-up of the explicit rule of computation for the function value.

In conformity with the general emphasis on efficiency of generated programs, methods to evaluate subscripts of subscripted variables recursively and using index registers were considered rather early (11) since these were deemed to be the most important problem of optimization.

Schemes of this type were incorporated in the translators for PERM by Seegmüller, and for ERMETH by Schwarz.

The extent of the language adopted by the ALCOR group after lengthy deliberations in 1960 was described in the ALGOL Manual of the ALCOR group by Baumann (3).

The main restrictions against ALGOL 60 are:

(1) no conditional Boolean expressions since these essentially duplicate the propositional calculus which is the better known of the two

(2) no "conditional designational expressions" since they are unnecessary

(3) no while-element in the for clause since this essentially duplicates the step until element

(4) no *own* since neither need nor definitions are clear

(5) no recursive procedures.

In addition there are rules for the use of procedures which are essentially programming rules.

(1) type procedures are called only by function designators. They serve only to compute the function value, and no side effects of any kind (recomputation of actual or global parameters, side exits) are permitted.

(2) In proper procedures called by procedure statement the function character is underlined by a standard heading of the form

P(x,y) results: (u,v) exits: (l,m)

separating different classes of parameters. This separation has to be used also in procedure statements.

These rules are intended to keep the notation close to standard mathematical form, and to ensure easy reading of programs. The ALCOR group considered both these requirements to be fundamental postulates with respect to the language.

To conclude, we come back to the ALCOR group to give a short survey of its present status.

The following translators were designed on the basis of ALCOR plans and are now running:
ALCOR-ERMETH, -MAILUFTERL, -ORACLE, -PERM, -S 2002, -Z 22, -Z 22 R.

They are all of the "load-and-go" type requiring no intermediate print-out although none of them except the ORACLE-translator has magnetic tapes at its disposal.

A translator for the Telefunken TR 4 has been programmed in Munich and will become operative with the machine. Smaller translators exist for DERA in Darmstadt and for the Zuse Z 22 (ALCORETTE).

The membership of the group at present is as follows:
Institut für Angewandte Mathematik der Eidgenössischen Technischen Hochschule, Zürich
Rechenzentrum der Technischen Hochschule München,
Institut für Angewandte Mathematik der Universität Mainz,
Institut für Praktische Mathematik der Technischen Hochschule Darmstadt,
Zentral-Laboratorium der Siemens & Halske AG., München,
Institut für Angewandte Mathematik der Universität Bonn,
IBM—Forschungsgruppe Wien,
Oak Ridge National Laboratory, Oak Ridge, Tenn.,
Telefunken GmbH., Backnang,
Zuse KG., Bad Hersfeld,
Dr. Neher Laboratory of the Netherlands Postal- and Telecommunications Services, Leidschendam,
Standard Elektrik Lorenz AG., Informationswerk, Stuttgart
IBM—Deutschland, Sindelfingen

Here, the last three should properly be called associated members since they developed, or are in the process of developing, their translators completely on their own. Their membership therefore concerns mainly use of the common hardware representation, and adoption of ALCOR-conventions concerning the use of ALGOL.

The ALCOR group is essentially an experiment which is not finished yet. Its success so far has been very encouraging although not perfect. In particular, the problem of active cooperation between members has not been solved to full satisfaction. This is due mainly to the severe shortage in manpower and funds in the University institutes which all had a full computing and teaching load to cope with. Thus every institute had to go ahead on its own to build its translator as the manpower situation permitted. The result was that the translators (all essentially one man jobs) show slight deviations. Thus there is still room for improvement, as was to be expected in a group consisting of members as individualistic as university institutes usually are.

In spite of these minor faults, however, we feel that we, in the ALCOR group, have already proved our point: it is possible to set up a uniform system which allows free exchange of programs and information between machines of types as widely different as e.g. Zuse Z 22, PERM, ERMETH and Siemens S2002 without seriously impairing the efficiency of the individual machine.

REFERENCES

1. F. L. Bauer und K. Samelson, Automatische Rechenmaschine und Verfahren zu ihrem Betrieb, DBPA B 4422 IX/42 m, München 1957
2. F. L. Bauer und K. Samelson, The cellar principle for formula translation, Proceedings ICIP Paris 59, p. 154-155 Paris-München-London 1960
3. R. Baumann, ALGOL-Manual der ALCOR-Gruppe, Elektron, Rechenanl. 3 (1961), 206-212, 259-265
4. C. Böhm, Calculatrices digitales, Dudechiffrage de formules logico-mathematiques par la machine meme dans la conception du programme, Annali Mat. pura ed appl. 4, 37 (1954) p. 5-47.
5. E. W. Dijkstra, Recursive programming, Num. Math. 2 (1960) p. 312-318
6. A. A. Grau, Recursive Processes and ALGOL Translation, Comm. ACM 4 (1961) p. 10-15
7. N. J. Lehmann, Bericht über den Entwurf eines kleinen Rechenautomaten an der TH Dresden, Bericht Math. Tagung Berlin (1953) p. 262
8. P. Lucas, Die Strukturanalyse von Formelübersetzern, interner Bericht Mailüfterl Wien, (1961)
9. H. Rutishauser, Automatische Rechenplanfertigung bei programmgesteuerten Rechenanlagen, Mitt. Institut Angew. Math. ETH Zürich Nr. 3 (1952)
10. K. Samelson, Probleme der Programmierungstechnik, Intern. Kolloquium über Probleme der Rechentechnik, p. 61-68, Dresden 1955
11. K. Samelson und F. L. Bauer, Sequentielle Formelübersetzung, Elektron Rechenanl. 1 (1959). p. 176-182
12. H. Waldburger, Gebrauchsanleitung für die ERMETH, Zürich 1957

MACHINE INDEPENDENCE IN COMPILING*

Harry D. Huskey

University of California
Berkeley, California, USA

Since 1958, there has been a substantial interest in the development of problem-oriented languages as a means of controlling computers or data processing systems. All of these efforts have had as a primary purpose the goal of reducing the human effort necessary to prepare a problem for computation or processing on such a computing system.

Perhaps the most significant of these developments is the publication of the international algorithmic language called ALGOL in 1958 and a revised version in 1960. ALGOL can be described as a very general scientific language suitable for scientific or engineering computation. It has two purposes: one of these is for the communication or the publication of algorithms for solving problems, and the other is its direct use as an input language to computing machines. It has perhaps been more successful in the first case than it has in the second. However, there is a substantial number of translators which will accept ALGOL statements and produce machine language for the appropriate computer. In all cases, however, compromises have been made with the complete language. In some cases, these are very minor compromises; in other cases, they are very extensive compromises. Perhaps a more significant aspect of the development of ALGOL is the effect it is expected to have on the future design of computing machines.

Another language development, which has been sponsored by the Department of Defense of the United States, is that of COBOL, a common business-oriented language. In the meantime, the

---

*This research was supported by the Bell Telephone Laboratories under Grant D-603513 and by the Air Force Office of Scientific Research of the Office of Aerospace Research; the Department of the Army, Army Research Office; and the Department of the Navy, Office of Naval Research under Grant AF-AFOSR-62-340.

problem-oriented language which has been used most widely is the FORTRAN system, developed by the International Business Machines Corporation.

Simultaneous with the development of ALGOL 58, the author participated in the development of a problem-oriented language with a restricted field of application, namely that of simulation. In this particular problem there was no need for the various general features of ALGOL so a restricted language called NELIAC was developed and put into operation. This activity took place at the U.S. Naval Electronics Laboratory at San Diego. The unique feature of the NELIAC development has been that the translating system has always been written in the problem-oriented language itself. Consequently, revisions were easy to make and the description of the compiler on punched cards or punched tape was the up-to-date documentation of the system. Since this system was developed simultaneously with the development of the specifications of ALGOL 58, some of its features are more similar to ALGOL 58 than to ALGOL 60.

THE NELIAC SYSTEM

In the NELIAC system, it is possible to modify the command generators so that they will generate commands for a machine B on a computing machine A.

The original NELIAC system was written for a military computer called the Sperry-Rand M-460, and the first version generated commands for the same computer. Early in the development, the system which ran on the M-460 was modified so as to generate commands for the Datatron 205 and also for the Datatron 220. In another effort, the M-460 system was modified to generate commands for the CDC 1604.

With some help in the way of hand operations, the whole translating system was transformed from the M-460 to the Datatron 220 to obtain a NELIAC system which runs on the Datatron 220 and generates commands for the 220. The need for the hand operations arose primarily from the fact that the Datatron 220 is a decimal machine without access to binary bits, whereas the M-460 is a strictly binary machine. In a similar way, the system was transformed onto the CDC 1604 so as to have a NELIAC system which would generate CDC 1604 commands on that computer.

In another effort, the NELIAC system on the M-460 was modi-

fied so as to generate IBM 709 commands, and in a boot-strap type operation, the whole compiler was transformed onto the IBM 709 computer. A version of this was developed which generated IBM 704 commands, and of course the 709 version runs on the 7090. Various features have been added to the 7090 system to take care of the special machine features available.

In each of these cases, the NELIAC system is written in its source language form and can be recompiled on the specific computer. Consequently, if some variation in the translator is desired for a particular purpose, this is easily accomplished with a minimum of man-hour effort.

In all this development, however, whenever a new machine is considered, it is necessary to change the command generators so that they will generate commands for the new machine. If the logic of the new machine is different, or if there are features which are of sufficient interest to be used, then perhaps even the logic of the compiling has to be modified to some extent. Thus, it may require as much effort as six man-months to establish a NELIAC system on a new computer.

TRANSFERRING A COMPILER TO A NEW COMPUTER

Therefore, since new computers are going to replace old computers and it is very important that substantial programming efforts do not have to be done over again, it is of very considerable interest to be able to transform a compiling system onto a new computer without modifying the source language so that the body of the existing programming can be transferred onto the new computer as well. Programs written in ALGOL satisfy the restriction that they do not have to be revised in order to run on a new computer; however, all the current translators would have to be done again for any new computer. Therefore, one of the purposes of the activities described in this paper has been to minimize the man-hour effort required to transform a translating system onto a new computer.

In order to do this, the translating system has been conceptually divided into three major parts. The first part is called a preprocessor; the second part is called a translator; and the third part is called an assembly program. The preprocessor effectively takes care of identifiers and a few other features of the source language in such a way as to minimize the activities of the translator and the assembly program as far as identifiers

are concerned. Thus, one could say that the preprocessor essentially converts general identifiers into relative addresses in an appropriate computer. Along with these relative addresses certain flag information is established, indicating for example whether the identifier is a label, or relates to real, integer or Boolean-type quantities. Thus, the assembly program can quickly compile a correct object machine program; the preprocessor makes almost no changes in the syntax of the source program. The translating portion of the system accepts the output of the preprocessor and effectively passes the identifiers on through without modification. The syntax of the statements, however, is processed completely so as to obtain a sequential list of operations appropriate for a computer in any of a large number of types. In other words, the output of the translator might be said to be a type of Polish string, in which identifiers and operations are developed in the order in which they must be considered in the object computer. The output of the translator is called an *intermediate language.*

CHARACTERISTICS OF THE INTERMEDIATE LANGUAGE

It is clear that if the intermediate language is to serve for a number of computers, then it must be as general as the most general of any of the machine languages of the object computer. Another way to state this is that any information present in the source language about the problem which can be used in assembling the final machine language must be carried over to the intermediate language. For example, if the intermediate language referred to a single accumulator, whereas one of the object accumulators had many accumulators, it would be difficult to write an assembly program which could analyze the intermediate language and decide how to distribute the activity among the several accumulators. Consequently, such information must be developed in the preprocessor and the translator, and must survive in the intermediate language form of the program. In the same way, information about index registers must not be collapsed in any way so that if a sufficient number of index registers are present in one of the object computers, then indices can be handled in an efficient way to solve the problem.

Consequently, the intermediate language is set up on the basis that there may be an infinite memory, that there may be infinitely many index registers, that there may be, in fact, infinitely

many accumulators. In order to allow for extensions of the source language, there is provision for flagging the identifiers in the preprocessor so that the assembly program can be expanded to handle new types of identifiers. In this way more classes of variables may be considered than have been treated in the various problem-oriented languages. This "no limit" restriction on index registers and accumulators requires the possibility of variable length identifiers in the intermediate language. Consequently, simple identifiers, or ones that occur frequently in source problems, are represented in simple form, and as more and more variables are needed, then more lengthy identifiers may be used. Also, since this intermediate language must be machine independent, it is not feasible to talk about words. Thus the intermediate language deals with characters and groups of characters called identifiers, operators or flags. On the other hand, in order to obtain efficient operation, those symbols and operators which occur frequently are simply represented, and the first identifiers that appear in a problem are likewise simply represented. As the list gets longer, provision is made for more complicated representation for the identifiers.

THE ASSEMBLY PROGRAM

The assembly program takes the output of the translator, or the so-called intermediate language, and transforms this into absolute machine code for a particular object computer. Consequently, a different assembly program is required for each computer.

The identifier which come from the earlier stages of the translation process are essentially in relative address form. In such cases the assembly program simply adds a base address to these identifiers and uses this as the address for the appropriate object computer. General sequences of arithmetic operations which appear as output of the translator must be transformed into the appropriate machine language commands for the object computer. In some cases the command specified in the intermediate language may not correspond to simple one-command operations in the object computer. For example, the variables being processed may be real and the arithmetic operations need to be floating point, whereas the object computer mat be a fixed point machine. In this case the assembly program develops the appropriate memory reference commands to obtain the oper-

ands and generates the subroutine transfer into a program which will do the floating point arithmetic operation.

As another example, it may be that there are nested parenthetical expressions (in an arithmetic statement) requiring the use of several working addresses or several accumulators. The intermediate language is set up on the basis that there are infinitely many accumulators, so as the nesting takes place the computation may move from one accumulator to the next. If the object computer has only one accumulator, then it is necessary that the assembly program generates "store in working address" commands and corresponding commands which will assemble these results as the closing parentheses occur. Also the object computer may have none, a few, or hundreds of index registers. Consequently, if the source language problem deals with subscripted variables, then the development of index operations may be quite different in one object computer as compared to another.

On the other hand, there are certain operations which the assembly program must do which are very similar in all computers. For example, there are transfers of control or subroutine transfers which are to absolute locations which cannot be determined in advance. This is true of transfers into future locations (not yet assembled), in that the number of commands down to that future position may vary substantially from one computer to another. Hence, the assembly program must keep lists of so-called future addresses or develop transfer vectors which will take care of these connections. This operation may be done in essentially the same way in any of a number of different computers.

TRANSFORMING THE SYSTEM ONTO A NEW COMPUTER

If it is desired to transform the system onto a new computer, it is only necessary to rewrite the assembly program for the new computer. Once this assembly program has been completed, then the preprocessor and the translator may be processed on any computer for which the translation system exists, so as to obtain them in the intermediate language form. This intermediate language form may then be assembled on the new computer. Then one has a preprocessor, translator and assembly program which will run on the new computer and any problems stated in the appropriate source language may be processed on the new system.

Actually, the assembly program on the old computer may be modified so as to generate new computer commands on the old computer. In this way the "bookkeeping" portions of the assembly program need not be hand written for the new computer. The intermediate language version of the assembly routine may then be processed by the modified assembly program on the old computer. The output is an assembly program which runs on the new computer and transforms intermediate language statements into new computer code.

THE ANALYZER

In the statement of a problem in a language like ALGOL there is considerable redundancy, particularly between declaration-type statements and the way the corresponding variables are handled in the body of the program. This has led to the study of a proposed analysis system which will look at the ordinary statements of a program and will attempt to generate the appropriate declaration statements.

There are two schools of thought about the merits of doing this. One school says that the redundancy in the statement of the problem is of importance because this allows the translator to detect errors in a problem. The other school of thought says that, if a person is to move on to more and more complicated situations, it is necessary that the language that he uses be efficient.

In the current approach to this analysis problem, the idea is to analyze the statements of the problem and generate appropriate declaration statements whenever no ambiguity is involved. In cases where it is not quite clear what declaration should be made, that which would most frequently be correct will be made; the information will be printed out indicating to the operator that such a declaration has been made but that there might be some question about it being the proper one. In this way, an operator can program a large problem, run it through the analysis program, and get a list of questionable declarations which he can review for correctness. This analysis routine will also inspect multiply-subscripted variables and decide how to handle index registers. This allows the possibility of carrying values for a particular index appearing as a subscript in a number of different positions. For example, in a doubly-subscripted variable, a single index in the first position will mean one thing, whereas

the same index in the second position will mean a different thing. In one case, it is the element in the row; in the other case, it is the column in the array. If the analysis system detects such use of the index register, then it can issue index modification statements which will carry both values of the resulting index and for any incrementation increment each one in the appropriate way. In this way, in the current state of the computing machine art, more efficient object programs can be compiled than can be compiled by the more general conventional way of multiplying the indices in the various positions by the appropriate constants.

All the inspection of the source program for consistency and for obeying of syntactical rules is moved up into this analysis program. This permits the preprocessor translator and assembly programs to run at maximum speed.

THE TRANSLATOR

The translator inspects the syntactic aspects of the output of the preprocessor and transforms it into something more or less equivalent to a Polish string.

Thus, the various operators, separators and bracketing symbols are all ranked. These are processed from left to right and, depending upon rank, either intermediate language commands are developed in the output or entries are made in a stack. With each addition to the object program the rank of the last entry in the stack is inspected to determine whether processing continues with the source language or with the last entry in the stack.

The intermediate language has been designed in such a way that the commands for the output are all obtained from a table. This means that the translation process runs exceedingly fast. For example, speeds above 20,000 per minute can be expected from currently available commercial high-speed computers.

In fact, by proper preprocessing and analysis, translations may be combined with assembly.

A PRACTICAL SYSTEM

The conceptual division into analyzer, preprocessor, translator and assembler is convenient for exposition purposes, or for purposes of task assignment in a group effort. However,

since much of the action during assembly is still computer independent, another approach is suggested.

A basic subset of ALGOL is chosen including assignment statements not using multiply and divide operators nor parentheses. Procedures without recursion, with no arguments, are used. Only integral-type variables are used. Boolean quantities are represented by zero and non-zero integers. To this is added a declaration of the number of characters per word. A concatenation operator and a first-character operator are added to the system. Two undefined procedures named READ and WRITE are included. READ brings the next line of input (next card or characters from a typewriter down to a carriage return) and WRITE records on an output medium one line of characters.

Suppose this language is called L. A translator, T_L^A, for this language L is required on some computer, A. An important point is that this translator is also written in its basic language L (subset of ALGOL with extensions). All translators written in this basic language can be transformed into machine language for computer A by translator T_L^A.

In order to transform a body of coding in language L into programs on a new computer, B, a new translator, T_L^B, is required. This is accomplished by modifying the command generators of translator T_L^A so as to generate commands for computer B. The magnitude of this task is less than one expects due to the simplified basic language L, which is such that the command generation is a table-look-up process. Thus, by replacing the tables of T_L^A, a translator T_L^B is obtained which runs on computer A and generates commands for computer B.

The translator T_L^B is processed by translator T_A^A, obtaining a translator T_B^B in the machine language of computer B. This translator will generate programs for computer B.

In this transferring to computer B the procedures READ and WRITE must be hand written in the machine language of computer B.

SUMMARY

Two schemes have been described. The first is a three- or four-stage (conceptually) processor in which only the last stage is essentially computer dependent. In converting to a new computer, parts of the last stage (assembly program) must be hand modified. In the second scheme a subset of ALGOL with extensions is used to describe the translators and other processors. The translator for this simpler system must be hand modified for a new computer system. The system is sufficiently simple so that this modification consists of changing tables and input-output routines.

The following students at the University of California are working on various parts of the above schemes: Jim Spitze, Gary Anderson, Bill Keese, Ralph Middleditch, Ralph Love, Norman Josephson, Niklaus Wirth and Ivan Flores.

THE CONSTRUCTION OF AN ALGOL TRANSLATOR FOR A SMALL COMPUTER

W. L. van der Poel

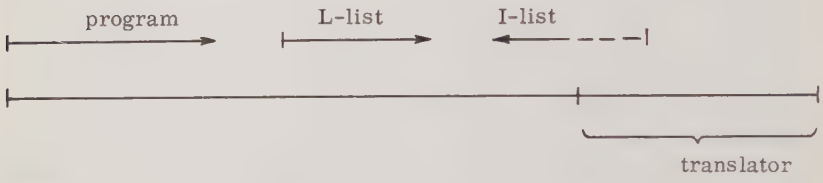
Dr. Neher Laboratory PTT,
Leidschendam, Netherlands

For the computer Zebra we have undertaken a project of constructing an Algol translator with the aim of being as complete as possible (with the only exception of own variable arrays). As a few new concepts are used in this design, it is thought to be useful to communicate them in this form.

Zebra is a two address machine with a very limited fast store and a reasonable drum store of 8192 words. The operation code is a rather peculiar one. Every bit of the 15 bits of the function part of an instruction is used separately for a different elementary function. The operations are in fact micro-operations and more complicated actions such as multiplication, division, list searching, etc. are not built in functions but can be readily programmed. This makes the code extremely powerful for just those operations which are required for an Algol translator. On the other hand it is not the appropriate object language for Algol. Therefore we decided to create a new object language within the machine which is used interpretatively. The structure of this object language (called Intermediate Code or IC) is fully adapted to the requirements of Algol. The analytic code of Zebra helps a great deal in making the interpreter fast and compact. Another reason why an interpretive code is not such a drawback is that the machine has no built-in floating point operations so that these have to be programmed anyway.

The system as a whole works on a load-and-go basis. First the translator is put into the machine, into the upper end of the store and the Algol program is then fed in, translated and built up in the lower part of the store. Then the interpreter overwrites the translator and the program can start working. We shall distinguish these phases by the names translate phase and run phase.

During translation the store is divided into three lists:
(a) the program to be built-up,
(b) the identifier list (I-list),
(c) the L-list.
The arrangement is as follows:



The I-list starts at the high end of the store and has an extension reading into the translator for the fixed (pre-declared) identifier such as sin, read, etc.

The program is built up in the lower part; the L-list is somewhere in between. The reason for making the I-list running backward is a very important one. For every occurrence of an identifier the I-list must be searched for this identifier and this searching process must of course be as fast as possible. In our analytic code this can be done by a repetition instruction, searching in ascending order through consecutive drum location. Thus a fully optimum timing is attained and the last identifier placed in the list is found first. This accounts for the local, non-local concept. For fast searching see (1).

We shall call an Algol symbol an identifier, a number or a delimiter. It is silently understood that the read routine already composes the complete identifiers and numbers and also determines the correct delimiter expressed by a word such as procedure. The read routine also skips all comment situations. We distinguish 3 such comment situations:

(a) the situation after comment
(b) the situation after end
(c) the situation after) followed by a letter.

The third kind of comment also copes with the parameter delimiter but far more than that. It is permitted in our translator to have any string not containing (. All comments are skipped only

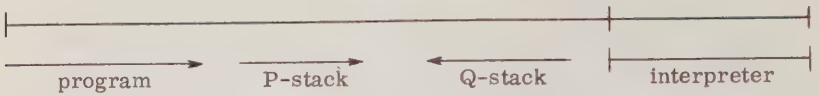
after treatment of the foregoing symbol by the translator thus obviating the difficulty of comment after the last end. Identifiers are read as numbers in radix 37. (26 letters, 10 figures and 1 spare). More than 6 characters give an overflow of the capacity of the word but this is allowed. Only a marking bit is kept which distinguishes an identifier of more than 6 characters from one with 6 or less. Hence there is a risk that two identifiers of more than 6 characters will be represented by the same word but we have taken that extremely remote chance thus being able to place no limitation on the number of characters in an identifier.

The L-list serves many different purposes. In the first place it holds these Algol symbols which cannot be translated directly because of their order of occurrence. E.g. in a + b * c + ... the read routine reads a, +, b* and on seeing * it knows that it has to go on reading until an operator of lower rank than * has been read. On reading the + it translates into take b, multiply c and only then remove a and + from the list by adding add a to the program.

It will be clear from this example that we have only obeyed in our translator to the rules of precedence for operators and we have not adhered to any rule for the order of evaluation of the primaries. The change of order into inverse polish notation is now well known so that I can leave details about this process to be read elsewhere.

In the second place the list L is used for string symbols like begin together with several stack pointers and variables as they were at the moment of reading begin. E.g. in the case of begin there are stored the address where the last instruction that has been translated has been stored, the last location of the identifier list and the last relative address given outside the block just being entered. Then after having translated the completed inside of the block everything what happened inside has disappeared from L and on finding end all pointers and variables are restored to their value before entering the block. When the program meets L or L meets I, the list L is shifted up or down wholesale.

During run time the translated program is already built up in the store. The translator is now overwritten by the interpreter (or for smaller Algol programs both can be held in the store together). The remainder of the store is again used for two stacks. The Q-stack is used for all simple variables and simple intermediate results. These intermediate results are stored in the true



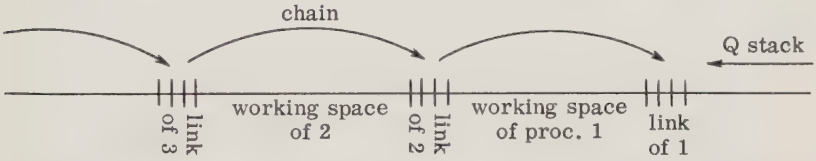
stack fashion on top and are exactly used in the reverse order in which they are put in the stack. Intermediate results are stored in unpacked form. The P-stack is used for arrays, fixed and variable. The allocation is fully dynamic and is only determined during run time.

Constants, own simple variables and own (constant bound) arrays are imbedded in the program itself. For a single constant the instruction using it automatically jumps over the constant placed consecutively. For own arrays the space for it is reserved at the time the declaration is translated.

The greatest advantages from the two stack system come from the allocation system during translation. There are two kinds of locations which cannot be given an absolute address during translation time; one is the variable array, which is dynamically allocated during run time; the other one is the space for formals in procedures.

As procedures can be used recursively a single absolute address is not sufficient for all levels of activation. The mechanism for allocations therefore is staged as follows: Every instruction has an operation part, an address part and a rank. As long as the translator is not translating the inside of a procedure declaration and hence is not dealing with any formals, all simples can be given absolute addresses. They get a rank 0. Also the entering of blocks does not necessitate other than absolute addresses. But as soon as a procedure is translated it is not known when or where or at what level this procedure is activated. Then all formals are given relative addresses, and the instruction referring to a formal gets an appropriate rank $\neq 0$. The first translated procedure gets a rank 1. When inside that procedure again another procedure is declared, this gets rank 2 etc. Hence the system of ranks is a static picture of the level of a procedure, not a dynamic one during run time. The top of the stack at run time when calling a procedure is put into the stack together with the link and some other quantities. Then the called procedure starts work and formals within the body are using the relative addresses given during translation relative to the beginning point determined during run time. When for example in a procedure of rank 20 a non-local quantity of rank 5 is

needed, the interpreter sees that the rank of the quantity needed is not the same as the present rank and then goes through the action of chain searching. The link of rank 20 contains the top of the stack at rank 19 and these quantities again contain the top of the stack at rank 18 etc.



The reason for not fixing down the beginning points of the relative addresses in procedures is that it can happen very well that in a procedure at level 20 a non-local quantity of level 5 is called which appears to be a new parameter-less procedure call rising to level 30 in itself and building up a completely new chain of relative addresses in the top of the stack from 5 to 30. This does not interfere with the old chain which still remains in the stack lower down. When this last procedure has been evaluated, everything needed for the evaluation has disappeared again from the stack and the former level 20 proceeds.

The process of chain searching is a very fast one in our computer as it can be done by underwater programming techniques. For a fuller account of this see (1).

To give a better insight into the coding of a procedure we shall give here an abbreviated form of the structure.
The procedure declaration is coded as:

X : An instruction calling the displacer subroutine. The displacer sets the link and makes the full formal actual correspondence.

D_1 : A specification pattern giving information about type, value, etc. of the first formal parameter

D_2 : Specification pattern of second parameter

⋮

D_k : Specification pattern of last parameter. All D's are positive.

-r : The rank of this procedure put there negatively so that the displacer can see the end.

body: Here follow all instructions of the procedure body in IC or machine code.

Y : The procedure body ends with a subroutine Y called the counter displacer which returns and restores the stack top pointers to their previous values in the calling program.

The procedure call looks as follows:

J : A jump to the X of the procedure to be called

s : A link address to where the procedure must return, which at the same time gives an indication of the place where to find the key of the first actual

Object progr. of Y_1: The object program for the evaluation of the first actual expression when this is not a simple variable. Absent when Y_1 is a simple

Object progr. of Y_2: etc.

⋮

Object progr. of Y_k:

0 A zero which signals the end of the retrogressive list of keys for actuals

Key of Y_k : The key of Y_k gives information about the kind of the actual, whether formal or actual. It can be a simple, an expression, a procedure, a label or switch, an array, a string or a formal

⋮

Key of Y_2

Key of Y_1

The displacer treats all actuals in turn and compares the pattern D_k with the keys, transports the quantities called by value and transforms type when the specified type is not the same as the type of the actual. In fact the type specification and the value list is the only information extracted from the specification. Procedure, label, switch, array, string automatically are clear from the parameter keys.

The structure of the instruction is worth reviewing for a while. The bits are numbered from left to right. Bit $q = \overline{I_0 - I_6}$ from the type of operation. When $I_0 = 0$ the instruction is a real machine code instruction and is not interpreted. When $I_0 = 1$ then $q > 64$.

The operations $64 \leqslant q < 96$ are the arithmetic, relational and Boolean operations such as add, equal, implies etc. Bit I_6 serves

the purpose of indicating whether a non-commutative operation is progressive or regressive i.e whether e.g. a subtraction has to form a-b or -a+b. Hence all operations such as division, implication, greater are only present in one version. The other version is produced by interchange of operands.

The operations $96 \leqq q < 104$ are the extractive operations such as take, take negatively, take inverse).

The operations $104 \leqq q$ are called the non-extractive operations. They contain the storing operations of different kinds (store accu, store address, store factor, store procedure) and many organizational operations as test, jump, pass, adjust (for the dynamic block begin), restore (for the restoration of stack pointers on leaving a block) and verify. The last one is a very peculiar one and is inserted where during translation no information about the kind can be extracted from the quantity itself as for example a constant as an actual parameter of a procedure which is itself called by name. Or in the case that a non-specified actual is a subscripted variable but where it is not known whether this is a designational expression or an arithmetical expression. The verify order is then substituted and during run time when of course the kind is clear from the actual keys the true operation is executed. Some operations are true machine code operations such as X, calling the displacer and Y, calling the counter displacer, restore (for labeled places for restoring stack pointers as entrance from jump leaving from inner blocks), return (for returning from an evaluation of an actual which is an expression), several kinds of for instructions for doing the for, step and while elements.

The bit I_7 indicates the type of the operation. $I_7 = 0$ indicated integer or Boolean type and $I_7 = 1$ indicates real type. It has an advantage to have Booleans represented by the integers 0 ——— 0 and 1 ——— 1. In that case the logical operations of intersection etc. on binary numbers can then be very simply done by applying Boolean operators on integers. But of course this must be hidden in a machine code body procedure looking like Algol on the inside as this sort of tricks does not belong to Algol.

The bit $I_8 = 1$ indicates in an extractive instruction that the type declared is not in correspondence with the type required. Then an automatic transfer of type takes place. For example in formal instructions the type bits I_7 and I_8 are copied from the parameter keys during the formal actual correspondence.

At last the bit $I_9 = 0$ indicates that a formal parameter is used, $I_9 = 1$ indicates that the operation concerns an actual.

$I_{10} - I_{19}$ denote the rank of the instruction which initiates the chain searching when the rank of the instr. at hand is not in line with the rank of the procedure at hand. $I_{20} - I_{32}$ is the absolute or relative address.

We have thus created within a machine not made for Algol an operation code and address system which comply very admirably with the requirements of Algol.

There is one further subject on which I should like to make some explanations. It is not required in our system to give procedure declarations or switch declarations in any prescribed order. It can very well happen that in a procedure A a procedure B is called while B has still to be declared, provided it is declared in the same block. Also a go to statement can lead to a label which is only declared later in the block or even can be declared later or earlier outside the block. (A label followed by colon is considered to be the declaration of a label). The solution has been found as follows: every occurrence of an actual or a label in a go to statement is counter-declared and placed in the I-list, also when already a declaration outside the block has occurred. Only on finishing the translation of a block all counter-declarations are treated and looked up whether they correspond with a normal declaration. They are then cancelled and all instructions concerned are adjusted accordingly. If not, they are shifted down in the identifier list and incorporated in the identifier list of the enclosing block. When then the declaration comes in the outer block they are treated there. At last no counter declarations may be left over otherwise there is an error. It follows that as switches are fully treated in the same way as labels, a switch declaration may in principle be given anywhere in the block.

Acknowledgement. A great many ideas for the realization of this translator have been contributed by Dr. G. v.d. Mey to whom I must express my warmest gratitude. Also to Mr. Mulders who coded a big part of the translator and brought the technique for underwater programming so peculiar for Zebra to a new height.

REFERENCES

1. W. L. van der Poel, Microprogramming and trickology, in Digitale Informationswandler, ed. W. Hoffmann Vieweg 1962.

AN ATTEMPT TO UNIFY THE CONSTITUENT CONCEPTS OF SERIAL PROGRAM EXECUTION

Edsger W. Dijkstra

Mathematisch Centrum, Amsterdam, Netherlands

A machine defines (by its very structure) a language, viz. its input language; conversely, the semantic definition of a language specifies a machine that understands it. In other words: machine and language are two faces of one and the same coin. I am going to describe such a coin. I leave it entirely to you to decide which of these two aspects of the subject matter of my talk you think the most important as it is rather ridiculous in both aspects. The language I am going to sketch is prohibitively difficult for a human user and the machine I am going to describe is of a perverse inefficiency.

Therefore, if my mental construction, nevertheless, has a right to exist it should derive this from other qualities. My machine derives this, to my taste and judgement at least, from its extreme simplicity and elegance, from the uniform way in which it performs its (at a first glance) rather different operations; the justifications for my language are its clearness and the unusually high degree of unambiguity, derived from a strict sequential interpretation and an explicit indication in the program to perform operations, which are usually implicitly understood (and therefore apt to misunderstanding). If one wishes to do so one may regard my machine and my language as being conceived for the purpose of clarification.

Before I really start with my description I should like to warn you of two intentional omissions. The system I am going to present is the result of a careful choice between a great number of "neighbouring possibilities." I shall not give my motivations for these choices, I shall even leave the consciously rejected alternatives unmentioned. In other words, I refrain from introducing my system at least in some respects as, say, a "local optimum." As this diminishes the convincing power of my pres-

entation, I personally regret this omission. I have to omit these motivations, however, for the sake of brevity.

The other question I shall not touch is the question of how to implement this system with the aid of a conventional machine. One might even raise the question—as I did myself to check that I was not thinking nonsense—whether it can be implemented at all, no matter how crudely. You have to take my word for it that it can be done. I have worked out a method of implementation to a degree that could convince, I think, the most suspicious auditor of the possibility. But it is my intention not to show you the particulars of this implementation, because I had to incorporate too many arbitrary decisions in it which, when mentioned, would only divert the attention from the essentials. In particular, the question of storage allocation will remain untouched.

My machine operates on (and under control of) units of information which I call "words." Without loss of generality I can restrict myself to a finite number of different words, each represented by the same number of bits.

The machine distinguishes between different kinds of words, say numbers, operators, variables and separators. For the time being we shall confine our attention to the first two of these, "number words" and "operator words."

A normal arithmetical operation, say the addition or the multiplication of two numbers, has two number words as input and one word, also representing a number, as output. The rules according to which a numerical value should be attached to (i.e. derived from the bits of) a number word are embodied in the workings of the arithmetic unit, which has the usual property that these same rules apply to both input and output: the output of the arithmetic unit can be fed into it again at some later stage of the process. As we assume that the properties of the arithmetic unit are constant in time, we may say that the number words have "a fixed meaning." As the fixed interpretation of number words is coupled to the constant properties of the arithmetic unit it is not so surprising that we shall denote the basic arithmetic operations by operator words (" + ", " − ", " * ", " / ", etc.) the meaning of which can also be regarded as fixed.

The machine works under control of a program which primarily consists of a string of words. For the time being I shall confine myself to pieces of program prescribing the evaluation of arithmetic expressions.

Let us consider the expression that would normally be written down as

5 + 39 / (7 + 2 * 3) − 6 ;

in the usual postfix notation (also known under the name "Reversed Polish Notation") this would give rise to the following sequence of numbers and operators (successive items in this sequence for the sake of representation on paper being separated by spaces)

5 39 7 2 3 * + / + 6 − .

The well known mechanism especially devised for the sequential evaluation of such an expression is what I prefer to call a "stack". (This device has been invented and generalized independently by so many people that it is known now under a great variety of names, such as "push down list," "nesting store," "cellar," "last-in-first-out-memory" etc.) If we regard the above sequence of numbers and operators as the string of words representing a piece of program, the machine reads this string word by word from left to right. If it reads a number word, this number (i.e. a copy of this number word) is added to the top of the stack, if it reads an operator word the operation in question is performed at the top of the stack. In illustration I give on successive lines the successive pictures of the top of the stack where the top is at the right hand side of the line.

```
. . . . .  5
. . . . .  5  39
. . . . .  5  39  7
. . . . .  5  39  7  2
. . . . .  5  39  7  2  3
. . . . .  5  39  7  6
. . . . .  5  39  13
. . . . .  5  3
. . . . .  8
. . . . .  8  6
. . . . .  2
```

and the net result of the execution of this little piece of program is that the value of this expression has been added to the stack.

As clearly shown in the above example the machine starts by copying the program text word by word onto the top of the stack.

Sooner or later this has to be interrupted, otherwise our machine would just be a copying machine In the above system the process of copying is interrupted by the occurrence of an arbitrary operator in the program text. The function of an operator, therefore, is a double one: firstly it indicates that the copying has to be interrupted for a while, because now an operation has to be performed; secondly it specifies this operation. I propose to separate these two completely different functions: from now on arithmetic operators are primarily treated in exactly the same way as numbers are treated, i.e. the operator word is copied into the stack as well. Every time the process of copying has to be interrupted I shall indicate this in the program explicitly by the insertion of a special word, introduced now and represented by "E" (from "Evaluate"). My machine now takes the following form. It reads the program text word by word, from left to right. By "reading" is meant the following: if the word read is unequal to "E" a copy of it is added to the stack, if the word read is equal to "E", it is not copied but, instead, the operation takes place as specified (primarily) by the top word of the stack.

According to these rules the program prescribing the evaluation of the expression of our previous example will now consist of the following string of words:

5 39 7 2 3 * E + E / E + E 6 – E

and under control of this piece of program text (i.e. when this string of words is "read by the machine") the top of the stack will be in succession as shown in the following lines:

. . . . .	5				
. . . . .	5	39			
. . . . .	5	39	7		
. . . . .	5	39	7	2	
. . . . .	5	39	7	2	3
. . . . .	5	39	7	2	3 *
. . . . .	5	39	7	6	
. . . . .	5	39	7	6	+
. . . . .	5	39	13		
. . . . .	5	39	13	/	
. . . . .	5	3			
. . . . .	5	3	+		
. . . . .	8				
. . . . .	8	6			

```
. . . . .  8  6  -
. . . . .  2
```

As said above the machine performs the operation specified by the top word of the stack when it reads the word "E" in the program text. We shall restrict ourselves to such programs that at such a moment the top word of the stack is indeed an operator word (and not, for instance, a number word). Furthermore we shall restrict ourselves to the case that the immediately underlying stack words are in accordance with any requirements that the execution of the operator at the top may set. (For instance, in the case of the binary arithmetic operations illustrated above the two immediately underlying words must be numbers.)

In other words: if an operand of an arithmetic operation happens to be an expression we substitute for this expression its numerical value before the operation is called into action, thus appealing to the fact that, primarily, the arithmetic operations are defined only when supplied with numerical operands.

We regard the replacement of a (sub)expression by its numerical value as a "substitution," and we indicate explicitly when these substitutions have to be performed, although, linguistically speaking, this is rather redundant: "3 + 4" will always be equal to "7", no matter when we perform this addition.

This situation, however, changes radically as soon as variables - in contrast to constant numbers - are taken into account. (In the following we shall denote variables with small letters, reserving capital letters for "special words," such as "E" and others, to be introduced below.) Let us assume that we have to compute the value of the expression

"x + 4"

at a moment that the value of the variable x equals 3. This means that in the above expression we must substitute for "x" its numerical value at that moment; only after having done so we can perform the arithmetic substitution ("3 + 4" being replaced by "7"). Given something dependent on x (viz. the expression "x + 4") we create a result (viz. "7") which, thanks to the fact that we have substituted for x its present value, is made *in*dependent of the future history of x. We have fixed an "instantaneous picture" of the variable x. Obviously I insist upon indicating explicitly *when* this instantaneous picture of the variable x (which is varying in time!) has to be taken.

Now we are going to harvest the first fruits of our labour for the mechanism for this explicit indication is already introduced. The piece of program prescribing the evaluation of the expression

"x + 4"

now takes the following form:

"x E 4 + E"

and under the above assumption the successive pictures of the stack are

```
. . . . .   x
. . . . .   3
. . . . .   3   4
. . . . .   3   4   +
. . . . .   7
```

Our machine invites us to describe the fact that "the value of the variable x equals 3" in slightly other wordings, viz. that the state of the process is such that reading the word "E" at a moment that the top word of the stack is "x" results in the replacement of this top word by the number word "3". The variable on the top of the stack is thus regarded as a variable operator which, upon evaluation, is replaced by something dependent on the state of the process at that moment; in this case it is an operator the execution of which sets no special requirements on the immediately underlying stack words. (The similarity between operators and variables will be further stressed by our next example.)

All words read in the text are added to the stack except the word "E" which causes the machine to perform a substitution. For reason to be explained below we should like to have also the possibility of adding the word "E" to the stack. The framework for this extension, however, is already present. We introduce a special operator, denoted by the word "P" (from "Postponement"), which effects upon evaluation a fixed substitution, viz. its replacement by the word "E". We shall illustrate the use of the operator "P" in the next example.

In this example we have three variables, named "x", "y" and "plinus". Suppose the state of the process to be such that read-

ing "plinus" "E" generates the word "+" on top of the stack. When reading the text:

"x P E y P E plinus E P E"

the top of the stack will show in succession

```
. . . . .  x
. . . . .  x  P
. . . . .  x  E
. . . . .  x  E  y
. . . . .  x  E  y  P
. . . . .  x  E  y  E
. . . . .  x  E  y  E  plinus
. . . . .  x  E  y  E  +
. . . . .  x  E  y  E  +  P
. . . . .  x  E  y  E  +  E
```

and the top of the stack thus contains the string of words which, when read as a piece of program, would effect the evaluation of the expression "x + y". If the value of the variable "plinus" had been "–" we would have generated (the string of words corresponding to) the expression "x – y".

What we have done amounts to a partial evaluation of the expression "x plinus y", the result again being an expression. In our previous examples the final addition to the stack always consisted of a single number. But a number is a trivial example of an expression and generating not only numbers but also more general expressions as intermediate results is therefore an obvious extension of the normal practice.

Up till now we have described the generation of words on top of the stack but not what we are going to do with these words. Furthermore we have assumed that with respect to a given variable the process could be in such a state that evaluation of this variable would give rise to a previously defined substitution, but how this definition should take place is not mentioned in the above. These two gaps in our picture will both be filled by the introduction of the assignment operators.

For the assignment of a single word value, as in "x := 3" we could write in our program

"3 x := E"

resulting in the stack pictures:

```
. . . . .   3
. . . . .   3   x
. . . . .   3   x   :=
. . . . .
```

Upon evaluation of the assignment operator ":=" the machine investigates the immediately underlying word. This must be the variable to which an assignment has to take place; the next underlying word is assigned to this variable (a process, about which more below) and the three words on top of the stack (which have now been processed) are removed from the stack. Until further notice—i.e. a new assignment to the variable "x"—the evaluation of this variable will result in the replacement of the top word of the stack by the word "3".

But for the interchanging of left and right hand side this is closely analogous to the assignment statement as known in ALGOL 60. But we need more than that for, in general, the assigned value will not consist of a single word, but of a string of words and we must therefore have a means of indicating how deep in the stack the assigned value extends. The simplest way to do this is to insert in the stack a marker, say the special word "T" (from "Terminal") at the bottom side of the assigned value. Furthermore we introduce another assignment operator ":-" (called the "string assignment" in contrast to the "word assignment" introduced in the previous paragraph). Upon evaluation of this operator the machine investigates the top of the stack in the downward direction. The first word (immediately under the operator ":-") must be the variable to which a value has to be assigned. Thereafter the machine continues its word by word investigation in the downward direction until it meets the special marker "T": the words passed in this way form together the string that acts as the assigned value.

The simplest way to add a "T" to the stack would be just to insert the word "T" in the proper place in the program under control of which the stack is being filled. This arrangement, however, will not do; for reasons to be explained later we need the possibility of generating a "T" on top of the stack under control of a program that itself does not contain this word. We can do this with the same trick that enabled us to generate an "E" on top of the stack. We introduce a new operator, denoted by the word "S" (say from "Separator" or because it precedes the "T"

in the alphabet) which upon evaluation is replaced by the word "T" and we make it a rule that this will be the only way in which words "T" are added to the stack.

Using all this we have an alternative way to write the assignment statement "x := 3", viz.

"S E 3 x :- E"

giving in the top of the stack in succession:

```
. . . . .  S
. . . . .  T
. . . . .  T  3
. . . . .  T  3  x
. . . . .  T  3  x  :-
. . . . .
```

The net effect of this is equivalent to the previous form using the word assignment ":=".

Let us use the more powerful assignment in an example which is an extension of one of our earlier ones, viz. the one describing the partial evaluation of the expression "x plinus y". The result of this partial evaluation was an expression depending on the variables "x" and "y"; suppose that we want to call this expression "z". For this purpose we write in the program:

"S E x P E y P E plinus E P E z :- E"

When the last "E" of this string is going to be read the top of the stack will be as follows (under the same assumption with respect to the value of "plinus"):

```
. . . . .  T  x  E  y  E  +  E  z  :-
```

and after the execution of this assignment the above words will have been removed from the stack, the word "T" inclusive. Until further notice the evaluation of the variable "z" will imply the execution (the "reading") of the string assigned to it. Upon evaluation of the variable "z" the machine therefore must have access to the first word of this string; when it starts reading this string, however, it must detect the last word of this string. We propose that the assignment operator sees to this by adding again an end marker and for this purpose we can use the very same word "T". Upon evaluation of the variable "z" the string assigned to it will be read as a piece of program, from left to right, until the end marker "T" is met. The new situa-

tion resulting from the last assignment can conveniently be represented by:

"z → x E y E + E T".

In exactly the same way our previous assignments

"3 x := E" or "S E 3 x :- E"

will both give rise to the situation, represented by

"x → 3 T".

One of the most illuminating aspects of this arrangement is that the usual distinction between "numbers" and "instructions" has completely vanished. The value of a variable is defined as a piece of program, evaluation of this variable implies the execution of this piece of program.

Furthermore we should like to draw attention to a certain form of duality between the assignment on the one hand and reading a text on the other. When the machine reads a piece of program text, the top of the stack is filled under control of this program text. In the assignment "readable text" is created under control of the contents of the stack. The duality can also be illustrated by taking into consideration the accessibility requirements. The words in the stack need only be accessible in the direction from top to bottom. If an assignment statement converts the top of the stack into readable text, however, the consecutive words thereby become accessible in the other direction.

Finally, the stack is reserved for "anonymous intermediate results", whereas readable text—in principle, at least—is always "named", for we create it by assigning it to a variable.

The attentive reader will have noticed that, along with the representation of the value of a variable, we have silently introduced two more complications in our machine.

The first one, the occurrence of the word "T" in program text and the machine's "immediate reaction" to it is a relatively simple one. As we have described the organization, the word "T", when read in the text, is *not* copied on top of the stack! Instead, it causes the machine to go on reading at the first word following in the string after the "E" that caused this evaluation of the variable in question. In other words, it acts as a "Return" at the end of a closed subroutine.

But the evaluation of a variable may call for the evaluation of other variables (even for the evaluation of itself): the pragmatic

definition of the evaluation of a variable is basically a recursive one and the mechanism one needs to follow a recursive definition is . . . another stack! I call this second stack "the stack of activations" in contrast to the first which I call "the anonymous stack". One of the functions of the stack of activations is to control the reading process. When the evaluation of a variable starts the stack of activations expands, when the corresponding word "T" is read, it shrinks to its previous size. (In the usual terminology of machine structure: the stack of activations contains a stack of "order counter values", its top element being, by definition, "the present order counter"; in this same terminology its older elements act as a stack containing the "return addresses".)

Note. We could try to merge our two stacks into one. This merging would present itself in a completely natural fashion if the two should expand and shrink "in phase" with one another. In general, however, this is not the case and trying to merge the two stacks into a single one would give a highly unnatural construction.

We shall use the stack of activations for yet another purpose, to satisfy a very fundamental need, viz. the creation of new variables. In the above I have used special words ("x", "y", "plinus" etc.) to denote variables and I have carefully avoided using the term "identifier". I have used the term "variable" in connection with a single, unique object, existing for some period of time and capable of taking on different values in succession. This concept of a variable is to be distinguished carefully from the "identifier" as used in ALGOL 60, because one and the same identifier may be used to point to a host of objects, to a great number of different variables.

First of all we meet the fact that one and the same identifier may play different roles thanks to the fact that it occurs in more than one declaration. A lexicographical rule then tells us which one of these declarations applies everywhere, where the identifier in question may be used. This form of multiple use of one and the same identifier could be removed by a simple process of renaming.

But there is a much more subtle case of "multiple use of one and the same identifier", viz. as soon as a certain block occurs in one or more nested activations (as in the case of a recursive procedure). In other words: one and the same identifier then refers sometimes to this variable, sometimes to another.

In actual fact: the identifier stands for a variable and in order to indicate clearly for which variable it stands I intend to denote explicitly the moment when a variable has to be substituted for an identifier.

For the sake of convenience—to be more exact: convenience for the machine and not for the hypothetical user—I intend to use the same identifiers for the local variables of every activation. (What I call "an activation" is closely analogous to a block or a procedure body, as known in ALGOL 60.) I use for this purpose the special identifier words "L0", "L1", "L2", etc.

When the machine starts the evaluation of a variable, the stack of activations increases by one item. At the start this item also contains a note that up till now no local variables have been introduced in this activation.

If the machine reads the word "E" at a moment that the top of the anonymous stack contains one of the identifier words (say "L2") then it investigates the top item of the stack of activations. If it is the first time that this identifier has to be evaluated in the present activation the machine creates a new variable for it (and may give this variable an empty value) and makes in the youngest item of the stack of activations a note to this effect. Then it replaces the top word of the anonymous stack by the variable just created for it. At a next evaluation of the same identifier at a moment that the same activation is still (or again) the present one, the machine finds in the top item of the stack of activations the note left there at the first evaluation of this identifier and the top word of the stack is replaced by the very same variable.

Now we can show a more complicated example. Let the values of the variables "x", "y" and "complus" be as represented by:

```
      "x → 10   23   T   "
      "y →  5   -2   T   "
"complus → L0   E    :=  E
           L1   E    :=  E
           L2   E    :=  E
           L1   E    E   +   E
           L2   E    E   L0  E   E   +   E
           T    "
```

If we now read the text

"S E x E y E complus E z :- E"

the net effect will be that we can represent the new value of "z" by:

"z → 15 21 T" ,

and what we have done can be interpreted as the addition of two complex numbers.

In ALGOL terminology: "complus" is a procedure with four numerical parameters, all called by value. The simple structure of the process allows the first of these to remain anonymous even in the procedure body. Furthermore, it is a kind of "type procedure", be it one that, syntactically speaking, takes the place of two primaries.

Let me end with a trivial example. Suppose that we want to write "plus" instead of "+". After the assignment

"S E + P E plus :- E,"

which gives rise to the situation

"plus → + E T"

the expressions

"x E y E plus E"

and

"x E y E + E"

are completely equivalent. This example is included to show as clearly as possible the arbitrariness of our primitives.

CONCLUSION

I am fully aware that the sketch is definitely incomplete. In particular conditional reaction and some equivalent of the *go to* statement should be incorporated if one wishes to make a system out of this. For the moment I leave these out and I do so for two reasons. Firstly for the sake of brevity and secondly because I have not decided yet: I know of several possible ways but none of them fully satisfies me.

With some versions of these facilities I have made slightly more elaborate programs. They showed me both the power and the weakness of my Language, its power being its flexibility and its unambiguity, its weakness being the fact that using it intelligently proved to be far beyond at least my powers.

If nevertheless I claim attention for this project I do not do so only because it charms me and may charm others as well. This report is the condensation of my meditations after we had completed our implementation of ALGOL 60. This implementation was conceived at high speed and the main justification for the numerous decisions taken in those hectic months was the recognition that our conceived constructions would lead to our goal and would do the job, in some way or another. The Machine described in this report, however, represents an extreme of the continuous spectrum of possible implementations of an algorithmic language which (as is the case with ALGOL 60) caters for recursiveness. In this quality it has been very clarifying for me personally: it has helped me a great deal in the appreciation of the various (initially disconnected) tricks we have incorporated intuitively and it has clearly shown us a number of alternative solutions. Therefore the hope is justified that translator construction and machine design in the future will benefit from these considerations.

Furthermore, the Machine presented here is so ridiculously inefficient that every practical implementation of a practical algorithmic language in all probability can be regarded as an optimization of it, an optimization which is permissible thanks to certain restrictions in the language. It may be useful to compare a proposed language with my language; during the process of language construction it may be helpful in the timely detection of "expensive features." Whether such an expensive feature will be included or not is more or less a political question but quite apart from how such a question is answered it is nice to know what one is doing.

Finally the language described in this report (or a language devised along similar lines) may prove to be a suitable means for the formulization of the semantic definition of an algebraic language. The lack of such a rigorous semantic definition is one of the recognized shortcomings of the official "Report on the Algorithmic Language ALGOL 60" and having seen the tremendous amount of trouble caused by this defect, I most sincerely hope that this report will contribute to the effort to avoid this mistake the next time an algorithmic language is to be devised.

ACKNOWLEDGEMENTS

A great number of people have contributed to this, consciously or not. Besides all my colleagues at the Computation Department of the Mathematical Centre, Amsterdam, I should like to mention Dr. M. V. Wilkes and Prof. J. McCarthy, who proved to be inspring listeners, and in particular Mr. M. Woodger: his judgment and his comments (I remember his lack of enthusiasm for my first trials in this direction now with gratitude) have been a great help for me.

COMMENTS ON THE ALGOL SYSTEM FOR THE SMALL AND MEDIUM SIZE COMPUTORS

Takeshi Kiyono and *Makoto Nagao*

Kyoto University
Kyoto, Japan

I. INTRODUCTION

The syntax of the ALGOL 60 language is constructed very rigorously but the feasibility of its implementation is scarcely considered. The result is: (i) the size of a compiler becomes large and complicated, and still more it takes much time to generate an object program, (ii) the generated object program is inefficient and time consuming. These are very serious problems for the small and medium size computers and rather unpreferable from a practical point of view.

We constructed, as a preparatory step, the main part of a compiler of the ALGOL 60 language with several restrictions. On account of these restrictions, the size of the compiler became comparatively small and the object program generated could be made nearly as economical as the one written by a good programmer.

We are now constructing a compiler which can accept almost all the syntax of ALGOL 60. However, compared to the former primitive one the size of the compiler becomes fairly large and the generated program seems to be unpractical for the actual computation.

As many kinds of compilers are constructed with as many kinds of restrictions imposed on ALGOL 60, we think that a few grades are to be established to the syntax of the language.

Of course there should be no contradiction among these grades. Then we can construct a compiler for a certain grade of the ALGOL language referring to the ability of the applied computer. Also the confusion arising from the different kinds of restrictions can be avoided by setting up such grades.

II. PROGRAMMING AIDS FOR OUR COMPUTER

(a) Symbolic Assembly System

We have installed an electronic digital computer called KDC-1 (Kyoto Daigaku Computer No. 1) at Kyoto University, where it has been in operation for one and a half years now. KDC-1 is a series parallel transistorized, 230 kc/sec clock pulse computer, with 50 words of core storage, a 4200-word drum, in which 200 words are quick access memories, and two magnetic tape units. The mean access time of the drum memory is 5 m sec. Instructions are single-address form with 95 different types of orders, all of which can be combined with the operation of the accumulator clearing, by adding the symbol "/" after the operation code.

KDC-1 is used by every department of our university, so that with the increase of the population of computer users, more and more keenly is felt the necessity of proper aids in programming. This could have been expected from the beginning the computer was planned, and the first step for the solution was the construction of a symbolic assembly program, which was completed in a comparatively short period and was named SYCS (Symbolic Coding System). The notation in this system is, instead of absolute machine code, the combination of three alphabetic characters for the operation code and the combination of five alphabetic characters at most for the address notation.

For example:

ADD/ 000 A: clear accumulator and add the content of A
MPA 000 X: multiply X by the content of the MD register and add to the accumulator

Further we can add ±n to the address notation such as

ADD/ 000 ADRS ± n

In this case the main address name (ADRS) must be composed of within four characters and "n" must be an integer of one figure i.e. -9 to +9. This system has saved considerably the simple but laborious task of converting the symbolic program into the machine-code program, and now is used frequently.

(b) Automatic Programming System Constructed as a Preliminary Step

Parallel to this system, we were contemplating the construction of an automatic programming system comparable with the

FORTRAN System of IBM. Just then the ALGOL Language, which had been proposed in 1958 and in which we had been much interested, was reborn in quite different, strictly defined structure. After examining the details of the structure we decided to adopt the ALGOL 60 language for our automatic programming system, as we thought on the one hand it has the rigorous structure which can tolerate the fairly complicated process of numerical analysis, and on the other it will become world-wide.

Considering, however, the memory capacity and the operation speed of our computer, we conceded, as the first step towards the complete implementation of the ALGOL 60 language, to the point where the following restrictions are imposed upon the system. This was begun in July 1960 and the main parts of the system was completed in March 1961.

The main principle of this system were:

(i) The scale of the compiler of this system should be compatible to the memory capacity of our computer KDC-1.

From the restriction of the scale of our computer we had to adopt the three pass system. That is, a source program written in the ALGOL language is initially converted into the pseudo-instruction form, which is changed to the machine code program by the symbolic assembly program SYCS. And finally at the third stage the program is actually performed yielding the desired result.

(ii) The effectiveness of the generated object program must be as close to the one written by the experienced programmers, that is to say, the generated program must be as efficient as possible.

The access time of the drum memory and also the operation time itself are not so fast in our computer that the redundant program will be fatal for the practical use of the system and the superiority of the convenient automatic programming system will be diminished to a great extent.

We have succeeded in satisfying these two conditions with the following restrictions and the compiler could generate reasonably efficient object program.

Restrictions:

(i) The type of variables and numbers in an arithmetic expression must be the same. However the types on both sides of the delimiter ":=" need not be the same at all.

(ii) An arithmetic expression must not include in its expres-

sion the conditional part, that is, it must not have the delimiters "**if**", "**then**" and "**else.**"

(iii) We did not adopt the left to right principle in calculation, which was one of the characteristic features in ALGOL 60, but we attempted that the calculation was to be done in the sequence which minimizes the number of object program steps.

(iv) The Boolean expression we took in was the relation alone, and it could be used only between the delimiters "if" and "then."

(v) We could use the arrays of as far as three dimensions. The bounds of the array must be fixed, i.e., the array of variable size could not be accepted. Further the lower bound of an array was fixed to zero. This was rather a severe restriction but the address calculation could be much simplified on that account.

(vi) The controlled variable of the for clause must be equal in type with the for list expression and it can not be the subscripted variable. When the for list expression is the step-until element, the final value of the controlled variable must be equal to the final value of the step until element.

(vii) We did not adopt the concept of the locality of identifiers. A variable defined in a certain position was effective throughout the program, so that the declarator **own** was not necessary. All the variables acted as the own variables.

(viii) All the identifiers, variables and so on used in a procedure body must be listed in the formal location of the procedure declaration, in case they are not declared within the body. No global parameters are admitted. This restriction is very effective, as the unanticipated side effect arising from the existence of the global parameters can be avoided. And if we suppose that the procedure declaration is a subroutine in the usual programming technique, it will be natural to list in the formal location all the parameters which are not declared in the procedure body.

(ix) The formal parameter which is declared as array in a procedure heading can not be called by value.

We think these restrictions were reasonable ones, for we could write fairly complicated algorithms on this language system easily. A few examples of the excellency of the compiler were:

(1) On account of the rejection of the left to right principle the calculation of an arithmetic expression, even if it contained many parentheses, could be done with the minimum steps and with the minimum working storages. Also as it did not contain the if clause, the compiler of the expression could be reasonably small.

(2) The address calculation of a subscripted variable became very simple by the restriction that the lower bound was to be always zero. In the example:

```
array A [ 0 : 9, 0 : 14; 0 : 19 ];
for i : = 0 step 1 until 9 do
  for j : = 0 step 1 until 14 do
    for k : = 0 step 1 until 19 do
      begin . . . . . A [ i, j, k ] . . . . end
```

the address of the subscripted variable A [i, j, k] is calculated by the formula

$$A[0,0,0] + (15 \times i + j) \times 20 + k$$

This calculation must be done $10 \times 15 \times 20 = 3000$ times for the completion of this for statement. This process is very time consuming, if we notice that

$$A[0,0,0] + (15 \times i + j) \times 20$$

remains constant during the change of the controlled variable k (assuming of course that the controlled variables i, j are not changed during the execution of the for statement concerning the controlled variable k), and also

$$A[0,0,0] + 15 \times 20 \times i$$

does not change during the change of j and k.

From this fact we divided the above formula into the three parts as shown in the figure 1. Then

$$W2 + k \rightarrow W3$$

is calculated 3000 times,

$$W1 + 20 \times j \rightarrow W2$$

is calculated 150 times, and

$$A[0,0,0] + 300 \times i \rightarrow W1$$

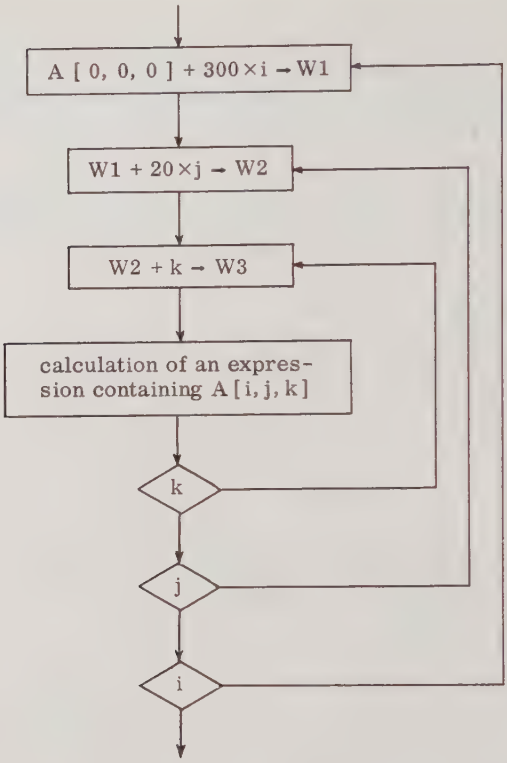


FIG. 1. Iteration loops of a for statement.

is calculated only 10 times. This saves the time of the address calculation very much. Further if the index registers of a computer can be well taken in the process of the calculation the speed of the actual run of the object program can be made even faster. In this example we used an index register for the controlled variable k.

(3) As we did not adopt the concept of the scope there might have been a little waste in the memory space, but the complicated process of the allocation and the cancellation of the memory space was not necessary. This made the compiler simple and the object program economical. This will be especially true when the variable size array is prohibited.

(4) As all the identifiers not declared in the procedure body

were to be listed in the formal location of the procedure declaration, the procedure declaration could be compiled by itself without any reference to the other part of a program. This is very convenient and serves the purpose of preventing the error which can arise from the complexity of the global parameters in the procedure body.

(c) Automatic Programming System Now Being Built

In a word the above-mentioned programming was experimentally constructed by the principle of the practicability for the actual calculation. To our great regret, however, the construction of this system was stopped at the point where main parts of the compiler had been completed and there had been left only a step towards their organic combination. For we have embarked on a new compiler which allows more freedom to the language, that is, the restriction to the ALGOL 60 language is to be made as small as possible. This new system is now under construction and of course the experiences obtained from the former compiler are being taken in extensively. One thing that we are afraid of is whether the new system comes out to be a practical one or not. For our computer is too small to accept such an enlarged system.

The restrictions on the new system are:

(i) unable to write a recursive procedure
(ii) unable to call an array by value
(iii) unable to use *own array*
(iv) unable to write the subscripted variable for the controlled variable of the for clause

Besides, the following problems are to be solved in harmony with the other part of the compiler for its completion. These are

(i) input and output statements
(ii) error detection and proper alarming
(iii) memory allocation of the object program

III. PROBLEMS FOR THE IMPLEMENTATION OF THE ALGOL LANGUAGE

At present, many kinds of automatic programming systems based on the ALGOL language are constructed with as many kinds of restrictions, so that the universality of the language, one of

the main aims of the ALGOL language, seems to be lost gradually. So we think that there must be some reconsiderations for the ALGOL 60 language from a standpoint of the actual computation processes or from compiler builders' point of view. Then we shall be able to have an ALGOL language which has both the strictness of the system and the practicability in its implementation.

When the scale of a digital computer is rather small, the following problems may arise for the realization of an automatic programming system based on ALGOL 60.

(i) The mixture of types of variables, numbers and so on in an expression.

In actual computation there are few examples which can not be expressed without the mixed type expression. This restriction saves the complexity of compiler very much and the generated object program will be more efficient than the one in the case of the mixed type.

(ii) If clause in an expression.

If we prohibit the left to right principle in calculation, the if clause in an expression can be rewritten by an if statement. This makes the compiler simple.

(iii) Left to right principle of calculation in an expression.

This concept will be an object of discussion. I want to take the affirmative side, if a computer is large enough, but can not when a computer is small. Besides we are not accustomed to the usage of this idea and so ambiguity and error may occur.

(iv) The number of dimensions of a subscripted variable.

A computer to which the ALGOL system is intended may inevitably impose some restrictions on the number of dimensions.

(v) Some restrictions to the general arithmetic expression in a subscript part.

For small size computer subscript expression should be simple so that special registers such as index register can be used effectively.

(vi) The array of variable size.

This may be very convenient for ALGOL users but very troublesome for the compiler and the object program generated becomes inefficient.

(vii) Dynamic own array.

Dynamic allocation of storage imposes a heavy burden on both a compiler and an object program.

(viii) The adoption of Boolean expression and logical values.

This will be welcomed but if the Boolean expression is mixed with arithmetic expression intricately, the situation becomes different. This may be beyond the ability of small computers.

(ix) The subscripted variable for the controlled variable of the for clause.

The execution of this may not be so laborious as thought of for a compiler of a large computer.

(x) Some kind of restrictions to the for list element.

This is to be considered for the generation of an efficient object program.

(xi) Declarator own.

This is closely related to the concept of block.

(xii) The concept of block or the concept of scope.

This helps the understanding of a source program and the writing of it very much, but this introduces rather bothersome processes of a storage allocation for both compiler and generated object program.

(xiii) The existence of global parameters in a procedure declaration.

This also is related to the concept of block. In exchanging calculating processes by ALGOL procedures among us, if there is no comment for the existence of global parameters in a certain procedure, the user of the procedure may overlook them. And also the procedure can not be compiled to an object program by itself.

(xiv) Array called by value.

I can not find the inevitability of using this method.

(xv) Recursive procedures.

When this is applied to an ordinary recursive calculation, the object program becomes simple.

(xvi) Input and output statements.

It will be preferable to have some common procedures for input and output.

(xvii) The increase of standard functions and the provision of standard procedures.

To save the inefficiency and complexity of an ALGOL program, it will be profitable to have many standard functions and standard procedures. These are already prepared as subroutines for a computer.

(xviii) The combination of the ALGOL language and the machine-like instructions such as shifting and logical operations.

(xix) The way of error detection.

This is not directly related to the language system but to what extent the detection of error should be done is important. This also depends on the size of a computer memory capacity.

(xx) Effective use of storage hierarchy.

It will be convenient to have some declarators for the allocation of storages.

(xxi) Clarification of the treatment of the term "undefined" which is used in several places in the ALGOL 60 report. For example "the value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array."

To avoid the confusion of the ALGOL language arising from many kind of restrictions, it seems preferable to establish some universally admitted grades to the ALGOL 60 language. For example the following four grades may be reasonable.

(i) primitive ALGOL—many kinds of restrictions such as the one listed above are imposed.

(ii) basic ALGOL—only the most difficult syntaxes to realize, such as (vi), (vii), (ix), (xiv) and (xv) listed above are excepted.

(iii) general ALGOL—accepts the whole syntax.

(iv) advanced ALGOL—accepts many kinds of extension to ALGOL, such as double precision arithmetic, complex variable operation and so on.

These four grades, of course, should not be contradictory to each other, that is, more general ALGOL system must include the syntax of less general ALGOL system.

For the ALGOL language system to be practical, the problems such as (xvii) and (xviii) may be important. The machine-level operations are to be well combined with the ALGOL system. The increase of standard functions and the provision of standard procedures will make the generated object program very economical, because such functions and procedures can be made as efficient as possible with the machine level instructions. We think that it is better to prohibit the existence of the global parameter in a procedure declaration and that the left to right principle in an expression is unpreferable, as it causes much confusion.

The strictness of the language system surely is important, but the practicability is far more so when the problems to be treated become larger and more complicated. Therefore we believe that the problems presented here are worth to be considered.

SEQUENTIAL TRANSLATION OF A PROBLEM-ORIENTED PROGRAMMING LANGUAGE

G. Palermo and M. Pacelli

Olivetti, Milano, Italy

INTRODUCTION

To be strictly sequential the process of the translation of a program written in a problem-oriented programming language, should involve a single sequential "character-by-character" scan of the input program and at the same time output the object program in a machine instruction form.

In the present paper we do not follow a character by character scan but for sake of simplicity we deal with an atom by atom sequential scan.

The translation will be implemented by the use of a stack. This allows a sequential translation of the source program in a parenthesis-free string of symbols and operators according to the rules of a generalized Lukasiewicz's reverse notation(1).

For such a translated program it is indispensable that the machine on which it will run be provided with a push-down list system.

A systematic transformation of the input program into a string which is written in such a notation, is possible if we suppose that every atom is completely self-explanatory and that a hierarchy always exists among the operators, which is reproducible with a suitable introduction of parentheses.

Practically, the translator operates on the source program and rebuilds it with recursive parenthesis structure, on the basis of both the syntax of the language in which it is written and a preexisting hierarchy of operations.

We may now consider a method of sequential translation performed by an abstract compiler on an equally abstract computer.

THE TRANSLATION

Let us translate on a machine M a problem-oriented programming language L, in the machine language L′ of a machine M′.

These two machines are not necessarily identical. We suppose that both M and M′ are provided with push-down list systems, and we will indicate by S the stack of M and by S′ the stack of M′.

We suppose also that the functions of the stack are known.

However we could say that a stack(2) is a list used as a "push-down, pop-up" list; i.e., elements of this list are added and removed from it with the "last-in, first-out" technique(3).

Let us suppose that L is an algorithmic language, more precisely similar to ALGOL, in which the following atomic quantities appear:

(i) arithmetic operators: $\oplus$ $\ominus$ $\uparrow$ $\times$ $/$ $\div$ $+$ $-$
(ii) relational operators: $<$ $>$ $=$ $\leq$ $\geq$ $\neq$
(iii) boolean operators: $\neg$ $\wedge$ $\vee$ $\supset$ $\equiv$
(iv) separators: ( ) [ ] : , ; :=
(v) alphabetic separators: "IF" "THEN" "ELSE" "GO TO"
(vi) identifiers:
  (a) names of real variables, scalar and subscripted
  (b) names of standard functions and procedures
  (c) labels.

It is easy to see that this list lacks some of the components of the ALGOL language. We will treat these features later in a manner which supposes an extension of the scheme of translation presented here.

The two new arithmetic operators $\oplus$ and $\ominus$, have the meaning of monadic $+$ and $-$.

For calling procedures we make the hypothesis that they are functions of only real variables and that their parameters are called by value.

We also impose the restrictions that GO TO statements will only operate upon labels. We have now imposed some restrictions but we also believe that these may be subsequently removed.

Let W and W′ be two indicators associated respectively with the translating program P and the compiled program P′.

W indicates the address of the atom of P at present under

the compiler's control, and W′ the corresponding address for P′.

The translation process is a single left-to-right scan of the program written in L. At every step the atom to which W refers is taken from P and compared with the arguments in a table T of the compiler.

The atoms such as variable identifiers, function and procedure designators, and labels are simply pushed-down in S.

In the table T, the operators have been introduced with respect to the rules of precedence which determine the execution

TABLE T

INPUT ATOM		LIST	
⊕ ⊖	)	⊕ ⊖	(
↑	2)	↑	2(
* / ÷	3)	* / ÷	3(
+ −	4)	+ −	4(
< > = ≤ ≥ ≠	5)	< > = ≤ ≥ ≠	5(
¬	6)	¬	6(
∧	7)	∧	7(
∨	8)	∨	8(
⊃	9)	⊃	9(
≡	10)	≡	10(
IF	11)	IF	13(
THEN	11)	THEN	) 11(
ELSE	11)	ELSE	) 11(
GO TO		GO TO	
(		11(	
)		11)	
[		11(	
]	12)	(	⌋
,		11)	11(
:		SUB "LABEL"	
:=	)	→	11(
;		SUB "EMPTY"	

of the expressions; these precedences are also signified by the number of parentheses associated with each operator.

Operators having the same degree of precedence are grouped under a single entry in the table.

The list associated with any entry of the table may be of the two kinds. One is the appropriate number of atomic quantities to be sequentially operated upon, and the other is a compiler subroutine designator.

Parentheses associated with any atom-operator in the table have the following purposes: (i) to take care of the rules of precedence; (ii) to respect all pre-existing parentheses, i.e., a nesting of operators; (iii) to pop-up from the stack S the left operand of the operator.

All the atoms contained in the list are pushed-down sequentially in S with the exception of the right parenthesis.

The presence of a parenthesis causes the stack S to be emptied of all its contents until a left parenthesis is encountered or the end of the stack is reached.

The number in the table that appears at the left side of a parenthesis, means that this number of parentheses must be successively considered.

In correspondence with the atoms ":" and ";" in the table appear the identifiers of two compiler subroutines.

The subroutine "EMPTY" causes the stack S to be completely emptied.

The subroutine "LABEL" produces the first cell of S and the address contained in W′ to be put together in an element of the Label List.

The Label List, which is built up by the compiler, has its elements divided into two parts: one for the label and one for the address of P′ associated with it.

The atoms drawn out from S are operated upon differently in accordance with another table that is not shown here.

However for each of these atoms we give a broad description of the operations associated with them.

The variable identifier atoms are placed in P′. The same procedure is applied to all the operators which we suppose that in M′ will produce the effect associated with their meaning.

The right bracket "]" produces an operator that as a function of the information contained in P′, the dimensions and memory allocation of the array to which it relates, and using

the actual values of the subscripts, computes the well-known mapping storage function and puts the element found in S′.

The atom "→" is put in P′ with this meaning: there is assigned the value of the operand in the first cell of S′, to the operand in the second cell of S′.

At the end of the operation both cells are deleted from S′.

The atom "GO TO" produces the label on which it operates to be taken from the first cell of S. Now two cases may arise:

(1) an identical Label is found in the Label List; then in P′ is built up a jump to the address specified in the list in correspondence with the selected label.

(2) none of the labels in the Label List is found equal to that of the "GO TO" atom.

This means that the "GO TO" operates on a label not yet examined by the compiler. In P′ is built up a jump (J1) toward an address assigned by the compiler where later there will be built the real jump (J2) to the address which will later be known. This technique for the "jump-down's" is called the "rebound-jump" technique.

Let us consider the simple conditional statement:

$$\text{IF B THEN } E_1 \text{ ELSE } E_2$$

Its atoms will be re-examined by the compiler in the following order:

$$| \text{B} | \text{ THEN} | E_1 | \text{ ELSE} | E_2 | \text{ IF} |$$

Neglecting the expressions B, E_1 and E_2 we now describe the operations connected with the three atoms "IF", "THEN" and "ELSE".

The atom "THEN" causes to be built in P′: the instructions that, will test the first cell of S′ on which the execution of B will have left a TRUE or FALSE result, and also a jump T similar to J* according to a FALSE result of B.

The atom "ELSE" causes to be built in P′ an unconditional jump E, type J_1. Then the compiler completes the rebound initiated by the atom "THEN" constructing at the address contained in T a jump to the address contained in W′.

The atom "IF" allow the compiler to complete the rebound

*See "GO TO" type 2.

initiated by the atom "ELSE" constructing at the address contained in E a jump to the address contained in W'.

When standard function and procedure identifiers are popped-up, in P are put the instructions that provide the call and the execution of the body of the subroutine involved.

We have mentioned before the lack of some ALGOL features. These at present have been excluded because they contain atoms that are not fully self-explanatory; they are without an unequivocal meaning in the syntax of the language and they are ambiguous to the above translator.

These components of the language need a "specifier" that defines their exact meanings. These features, such as FOR statements, must be treated with a scheme of translation in which there will be introduced some indicators whose tests will furnish information about the history of the past program, acting as specifiers for possible ambiguous atoms.

CONCLUSIONS

We think that the above method, in spite of the restrictions imposed, is simpler than other methods of sequential translation, such as the one suggested by GRAU(4) and by FLOYD(5), because of the simplicity of the compiler and its lesser number of operations.

With respect to the above restrictions, the method described in the present paper does not preclude any effort toward its extension to allow the inclusion of the features not discussed.

The main advantage that may be obtained by using sequential translation processes is to have a compilation time depending only on the time of the input of the translating program and of the output of the translated program.

In such a process the disadvantages exceed the advantages. The length of the object program is always and obviously greater than that of one that could be obtained from a many pass translator and the execution time is also not very efficient. A sequential translator is unable to realize an object program with a high degree of parallelism of operations for a program written in the so-called classic "von Neumann languages"(6).

Neither can it prevent the execution of identical operations more than once, or the execution of redundant unnecessary ones.

Typical examples of this, are arithmetic expressions that contain within themselves repetitions of identical subexpressions, and redundant boolean expressions both of which are always completely computed. Also when, as in the case of the example:

$$X := A \wedge (B > C + D \vee E \neq F);$$

with the boolean variable A which is FALSE, it is possible to determine the value of X without ending the computation.

This is all attributable to the use of a Lukasiewicz's notation to write an expression and to the use of a push-down list in the execution of these expressions. In fact the stack automatically supplies temporary storages and volatile variables that no translator would ever manipulate in order to save time in the execution of its translated program.

The last criticism is not only applicable to a sequential translation process but in general to all of the processes of dynamic handling of information and to the use of push-down list systems.

REFERENCES

1. Lukasiewicz, J., Elements of Mathematical Logic, Panstwowe Wydawnictwo Naukowe, Warsaw (1958).
2. Dijkstra, E. W., Recursive Programming, Numerische Mathematik, Oct. 1960.
3. Samelson, K. and Bauer, F. L., Sequential Formula Translation, Comm. A.C.M. 3, Feb. 1960.
4. Grau, A. A., Recursive Processes and ALGOL Translation, Comm. A.C.M. 4, Jan. 1961.
5. Floyd, R. W., A Descriptive Language for Symbol Manipulation, Journal A.C.M. 8, Apr. 1961.
6. Lombardi, L., Theory of Files, Proc. E.J.C.C., Dec. 1960.

THE CONSTRUCTION OF EFFICIENT COMPILERS FOR SMALL SLOW COMPUTERS

K. D. Tocher

The United Steel Companies Limited, Department of Operational Research, Sheffield, England

INTRODUCTION

The original doubts of the pioneers of computer developments about the practicability of efficient automatic programming systems have been swept away in the flood of fast access storage produced by the engineers and by the creation of a feeling of omnipotence by the immense speeds of the new generation machines.

Only now, that the practicability has been established and that advances are being made in generality of language, are a few voices being raised about the efficiency of compilers. It is becoming clear that the size and complexity of tasks which will be tackled on computers will soon catch up with the hardware advances of recent years, and then the creation of fast compiler systems which minimise the storage used will become more urgent.

Fortunately, there still survive sufficient of the old slow small machines in Universities, and the less developed countries to create the demand now for research in this direction already. The difficulties of programming have not decreased as the manpower devoted to it has increased - on the contrary. The various computer centres still dependent on manual coding show the same disappointing ratio of development time to program writing time as of old. The tendency to under-value computer running time on small machines leads centres with such machines to be much more lax about operator control of the machine and too often the machine is used to "find the faults for us". This, coupled with a general habit of writing trick programs even when this is not really required to fit it into the machine, leads to de-

velopment time often exceeding half the total computer running time of a job.

Thus the construction of compilers for small machines will serve two needs—it will supply an immediate demand and by designing these for compiling limited size problems will stimulate techniques for efficient compilers.

This paper is a contribution to this field of research.

EFFICIENCY AND GENERALITY

The exciting possibilities of generalising program languages by the inclusion of procedures and recursive features is one of the principal sources of inefficiency of modern compilers. If every statement has to be considered as a potential element of a recursion, then either the compiler must be very complex to establish that this is not the case (and in many cases this is not possible with infallible success) or the actual compiled program will reserve space for dumps never actually used. This danger has been emphasized by Wilks and Strachey in a recent paper (1).

In that same paper, they also warn of the dangers of side effects and suggest extra types of statements which give the compiler a guarantee that side effects cannot occur. This is a technique by which the author of the source statements imparts implied information to the compiler.

They also raise the similar problem of conveying information to the compiler of the motivation of the set of statements which are not deducible from the statements themselves.

They suggest the use of *notes* which give advice on storage and the status of variables which are clearly in the mind of the author in constructing the source statements but would involve considerable analysis (beyond our power at the moment) to deduce from the statements alone.

Such advice becomes particularly important in a small machine with very limited fast access storage and a backing store such as a drum and we shall consider how this advice can be made mandatory and yet as far as possible explicit statements eliminated.

The key to this lies in limiting the range of problems for which the source language is designed. The tendency has been for this to become more and more extensive - all arithmetic and logical procedures - all data processing systems - general list processing schemes. This increased generality appears at first

sight to sacrifice little efficiency; the basic type of statements are merely slightly extended and the range of objects that can be specified is increased.

The hidden loss lies in the loss of structure in the problems statable in the language. The idea of structure can best be explained by a few simple examples.

The simplest possible example is a *table maker*, for tabulating a set of functions as shown in Figure 1.

Here the notation 'ct z' indicates that z is changed by a unit (direction implied by context) and a test made on the number of changes.

The function boxes enclosed in double lines are the only part which require changing to tabulate a new set of functions and if our language were oriented to only tabulating functions, these would be the only part that would need description. The remainder could be written in machine orders and the storage for it allocated to give this organisational structure maximum speed of operation.

A similar flow diagram could be drawn for, say, the solution of differential equations, and the same remarks apply. Of course, all this is familiar to old-style programmers who have been constructing systems on this principle for many years.

The proposal is to extend this principle to wider and more important classes of problems, limiting the class to those for which the book-keeping and constant component remains a significant part of the total computing time. The variable part of the programs, the procedures, are written in a source language and after translation are incorporated in an executive program which is pre-prepared in machine orders. The source language should be designed so that a range of executive programs can use the same language.

For a complete automatic system, the construction of these executive programs should be undertaken by a compiler. This, however, need not be the original compiler and Brooker et al (2) have been working in this field. However, there are several semantic difficulties which preclude, at the present state of development, any consideration of efficiency of this type of compiled program. For the smaller machine, this objective can be relaxed, without losing the main advantages of the original compiler.

The need to write an efficient executive program partially determines the storage allocation in the sense that the class of

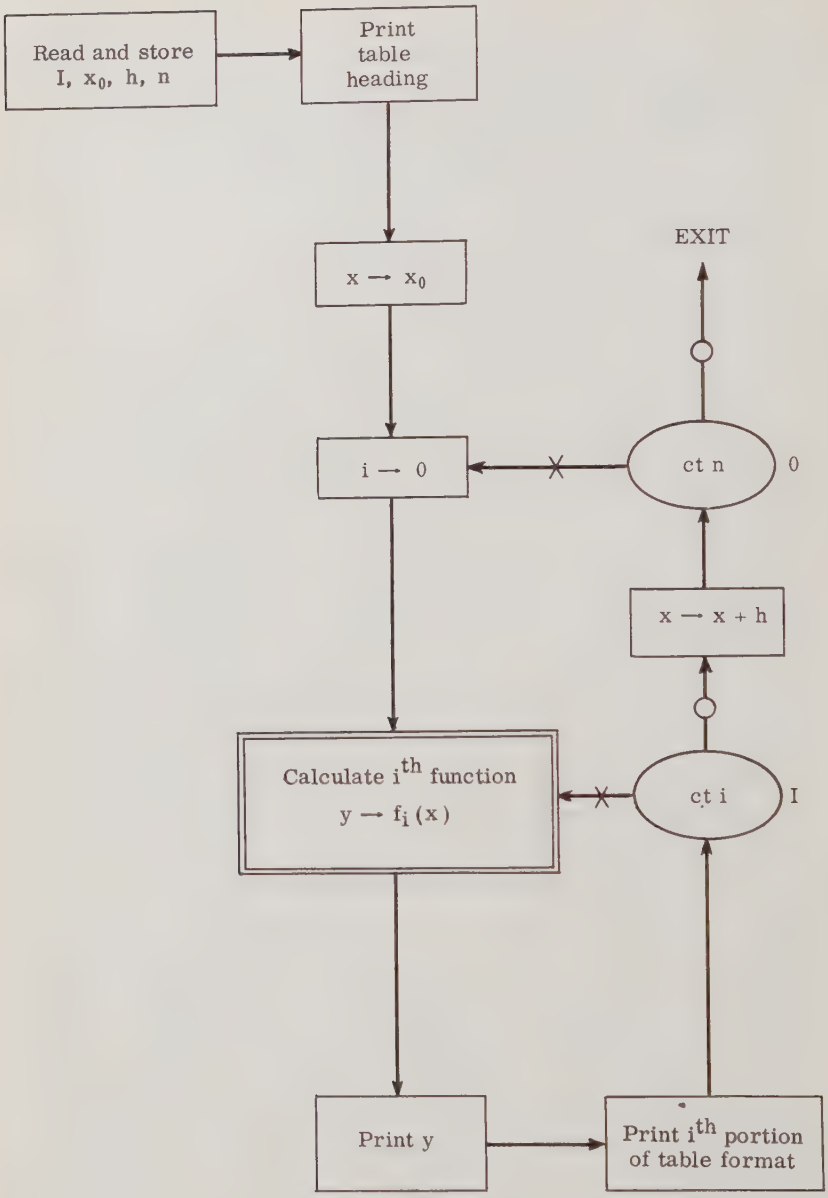


FIG. 1. Table maker.

problem considered requires certain storage for data and this should be in fast access storage if it enters into the variable procedures but can be in the slow access storage otherwise. The nature of the class of problem also determines a reasonable amount of additional data storage for use during the variable procedures. The remaining fast access storage is available for the execution of orders.

Thus the principal division of the fast storage for data and instructions is made by limiting the problem. For example, in the solution of differential equations using, say, the Kutta Runge process, the independent variable, the set of dependent variables and the interval, being needed by most of the differential calculating procedures, will be allocated fast access locations. The successive differences can be stored if space is limited and brought down successively by the executive for the common book-keeping operations of combining them for the final increments.

A further aid to allocation of space is the natural stratification of the storage by the block size defined by the computer code for transfer between the levels of storage. Thus by limiting the number of dependent variables to the number in a block, we can make substantial simplifications to the executive program. If this is not tolerable, then the dependent variables may, for lack of space, be held in the slow access store and a copy of one block of these held in the fast store. Now, the designer of the program can order his dependent variables to maximise the references to the fast store and, by means of a special procedure, change the block copied at suitable times. By giving the author the freedom to name either the slow access item or the copy in the fast store, the minimising of transfers lies in his hands. No arbitrary limitation on the number of variables that can be handled is imposed, and a simple program can be written ignoring the copying feature. This merely enables the author to reduce execution time by judicious planning. To extract such a plan from the simple plan would, even if an algorithm for it could be suggested, increase compiling time enormously.

This proposal is a compromise between the ideal of writing a program completely machine description free and hand coding the problem. The major strategic storage allocation for minimised execution time is left to the human and the book-keeping and exact allocation are done by the compiler.

The provision of hand coded executive material for certain recurring procedures has another practical advantage. As con-

trol passes from compiled material to the executive, or vice versa, it is possible to interpolate optional monitoring orders so that the course of the calculation can be traced during development.

Restricting such monitoring procedures to such points in the program does not materially diminish the effectiveness of the monitoring, but it has important implications for the compiler. If a monitor program can break into any part of a compiled program, either the compiler must write the orders so that a break-in is possible at any point without alteration of the orders (by the insertion of redundant tests) or alternatively a correspondence between the original statements and the coded form remembered so that temporary changes to the compiled material may be made.

It is desirable that the instructions to the monitor program should be in the same language as the basic program elements so that the same compiler can be used for both purposes.

Each program element has associated with it the following information.

(i) the storage allocated to the program;
(ii) the executive or program element to be entered after executing the element;
(iii) if an interruption is required for monitoring and if so, which monitor program to obey:
  (a) on entering the program;
  (b) on making a jump in the program;
  (c) on entering a subroutine program;
  (d) on leaving the program.

Executive elements can be treated in exactly the same way even if they are hand-coded, providing there is at least one element whose function it is to determine the next program element.

Subroutines are treated as program elements with the restraint that (at least in an elementary form of the scheme) they do not contain subroutines.

Both subroutines and executive elements require the same information as program elements and in particular interruptions on them are possible.

The monitor elements can be treated in the same way except that they cannot be monitored. Such a facility would be of little value and if provided, cyclic behaviour could be engendered.

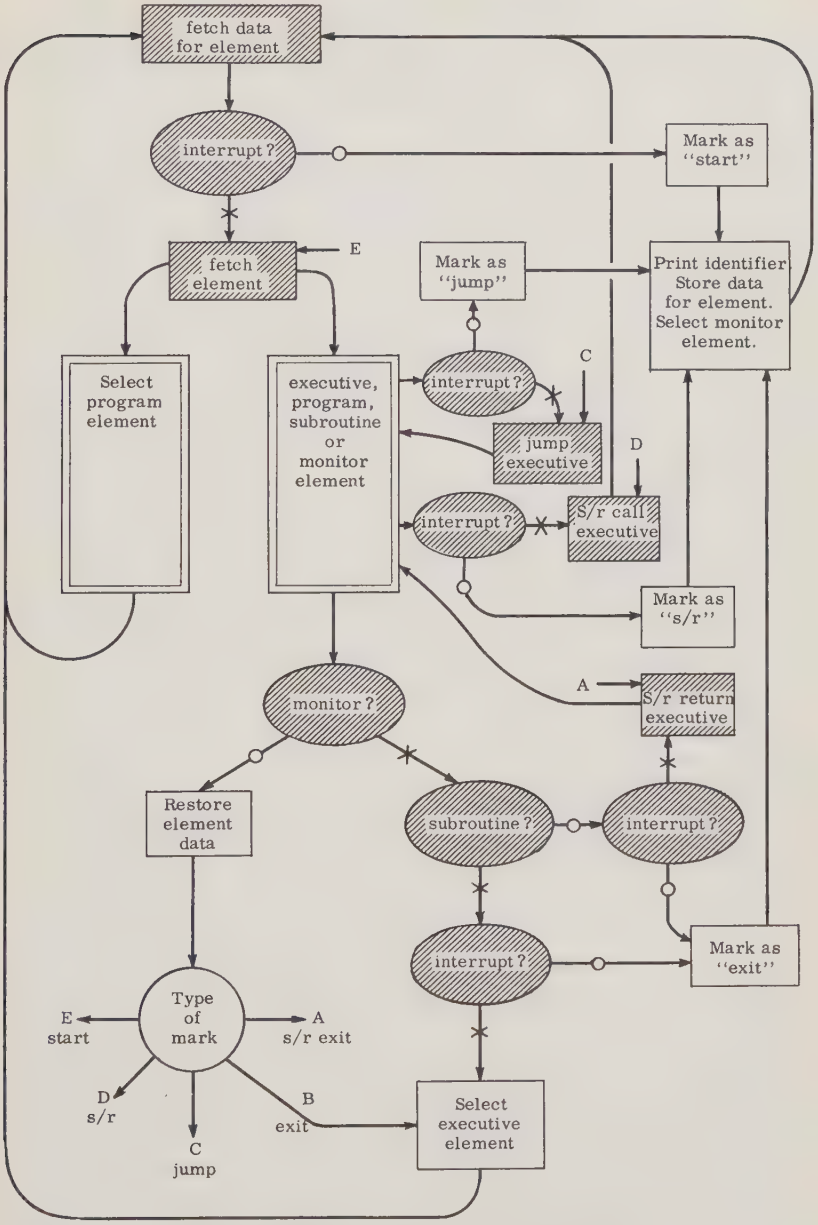


FIG. 2. General form of structured program.

Thus the types of element can be treated in a common way and Figure 2 illustrates a typical master flow diagram. Elements of any kind are brought into a common part of the store reserved for program execution by the master executive order "fetch element".

On exit, if not a monitor or subroutine, the data specified above is used to determine if an interruption is required. If not, a new executive (or program) element is selected and the data associated with that element is obtained by "fetch data for element".

If an interruption is required, the type of interruption "exit" is remembered and a monitor element selected by reference to the original data (which is stored). After printing an identifier to name the element which caused the interruption, the monitor data is obtained by "fetch data for element". No interruption can be called and "fetch element" causes the compiler to enter the monitor program. On exit, the original data is restored and a return made (by consulting the remembered type of exit) as if the interruption had not occurred.

If a program element calls for jumps or subroutines, a similar procedure applies to interpolate a monitor program or not according to the data provided. On exit from a subroutine, a return to the executive order "s/r return" is made which returns to the original element.

It should be noted that with this scheme, jumps are permissible in all types of elements but subroutines are only available in program elements.

The shaded elements of this flow diagram indicate the components which it is profitable to have available permanently in the fast access store. Any interruption calls down a common block of program which uses the automatic link to distinguish the type of interruption. By this device the number of orders executed to test for interruption is reduced to at most one for each test. The data concerning an element can be packed in a single word with a little judicious trimming of the number of possible monitor programs available and the sign bit used as the indication if an interruption is required or not. With this convention, interruption takes place at all or none of possible interruptions points of an element.

The limitation on the possible number of monitor programs can be relaxed by providing a monitor program which can change the programs associated with each possible code number

according to input data. The same program can be used to change the monitor data associated with program and subroutine elements. If the monitor feature is removed, the flow diagram reduces to Figure 3.

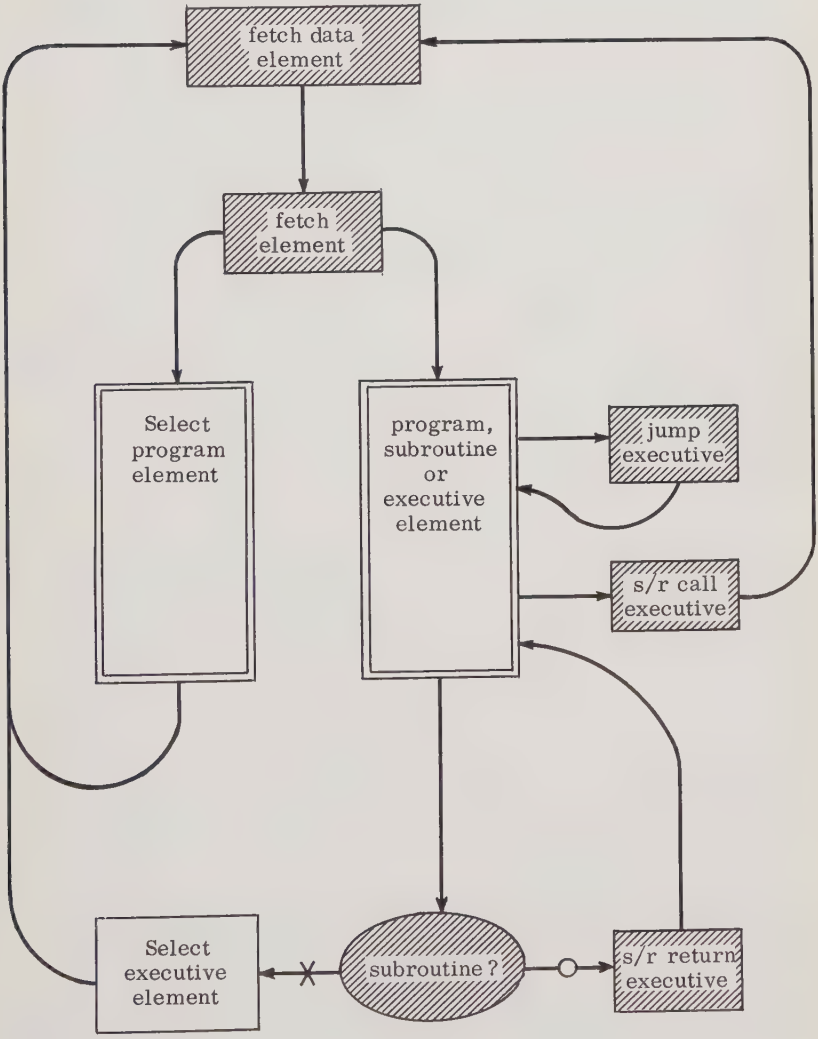


FIG. 3. Non-monitored form of structured program.

This general scheme may be made clearer by two examples from the allied fields of simulation and data collection.

A GENERAL SIMULATION PROGRAM

This program has been described in more detail elsewhere (3) but the following brief description is sufficient for our purposes.

A simulation is a dynamic representation of a system in which the successive states in time are generated by the rules of connection of the system and the laws concerning the operating times of its components. To be more specific, we consider the most important case of an industrial plant. This can be regarded as a set of machines each capable of taking, at any time, one of a set of specified states. The time any machine will remain in an active state is determined by an associated process and often, but not invariably, this is represented by a sample from a statistical distribution. At any moment of time, the state of the plant is completely specified by a list of the states and times of all the machines. Certain passive machines (such as stores, queues etc.) do not have times associated with them for any of their states and a division is made into time-dependent machines and others.

In practice, it is advantageous to allow several variables to be used to represent a state. In the General Simulation Program, a block of storage is associated with each machine to hold the states, and if necessary, the time. These can be arbitrarily ordered except that the time-dependent machines are grouped together before the others.

In practice, however, a reorganisation of the machines may help in describing the rules for changing the state of the plant.

A more compact description of the plant can be given in terms of activities but these are merely short-hand names for a subset of the machine state specification. Such an activity only endures till the least time of those specified.

The machine state description only changes at one of the stated times, and the first change takes place at the least of these times which is greater than the present instant. At this time, the activity (or activities) which terminates at this time, releases machines to undertake further activities.

Two kinds of activities can be distinguished; those associated with the previous activity and which will immediately follow it

regardless of the state of other time-dependent machines, and those which depend on the states and availability of other machines. These latter are called co-operative activities or C-activities. The former are bound activities or B-activities. They are frequently book-keeping activities which merely log in a special machine - a report machine the fact of the previous activity being concluded.

The program associated with an activity specifies the necessary states of machines and changes the state of machines and the associated times as appropriate.

A B-activity is associated with a machine and can be so associated with it at the start of the preceding activity. Thus in one activity, we may *engage* a machine to an activity.

If a complete list of possible activities is drawn up, specifying the necessary machine states for them to start, the description of the plant is complete.

The master program now divides into three phases.

(i) a time-advance phase in which a clock is advanced to the least time associated with a time-dependent machine ahead of the current value of the clock. In this phase, an ordered list of the returning machines is made.

(ii) a B-phase which examines every machine in this list for an engagement and for each engaged machine initiates the B-activity specified.

(iii) after the list is exhausted, a list of C-activities is scanned and executed in order. In practice, the initial machine state tests heading these activities often fail & so no state changes are made.

The cycle is now repeated indefinitely.

A compiler is required to translate the B- and C-activities expressed in a simple algebraic code into machine orders; the remaining program is hand-coded.

Each B-activity has associated with it a common following executive element (the B-phase executive element) which picks the next machine from the list, tests for engagement and sets the number of the engaged activity. This is then entered via the fixed executive.

There is a special dummy machine always added to the end of the list which is permanently engaged to an executive element which brings down the data for the first C-activity. This element, the C-phase executive element, is associated with each C-activity and brings down the data for the next C-activity.

The list of C-activities ends with the T-phase executive element which advances time, forms the list of returning machines and is associated with the B-phase executive element.

Provision is made for interruption at the beginning and end, and at jumps or entries to subroutines for each activity and at entry to and exits from subroutines. Ignoring the complications to the flow diagram of this feature, the programming system can be illustrated as in Figure 4. This gives in an expanded form precisely the structure of Figure 3.

EVENT DATA ANALYSIS PROGRAM

Simulation depends for its success on representing the behaviour of machines by a statistical distribution of certain process times and these must be extracted from records of actual plant behaviour. Ordinary works data is usually not accurate enough and special data has to be collected.

One approach to this has been through the development of an automatic recording device which enables a series of messages to be recorded, one for each event occurring on the plant. This gives a history of the plant event by event and corresponds to the successive states of the machine state description used in a simulation.

The problem arises of analysing this history to give the required process times. Information concerning congestion delays on the plant can also be abstracted from the history. Another type of analysis will give the histories of successive batches of material as they proceed through the plant.

A general method of analysis of such historical data is required and the key to this lies in realising that the input of this program will be of the same form as the potential output of a simulation. The language which will enable statements to be made about changes to the plant in terms of process times is sufficient to make statements on how to calculate process times from a record of plant changes.

However, a new executive program is required to use this common language (translated by the same compiler).

The recorder operates on a cycle of 3 secs. Corresponding to each machine or group of machines is a recording station on which the current state of the machine is set either manually or by automatic signals from the machine itself.

When the information is set correctly, a marking button is

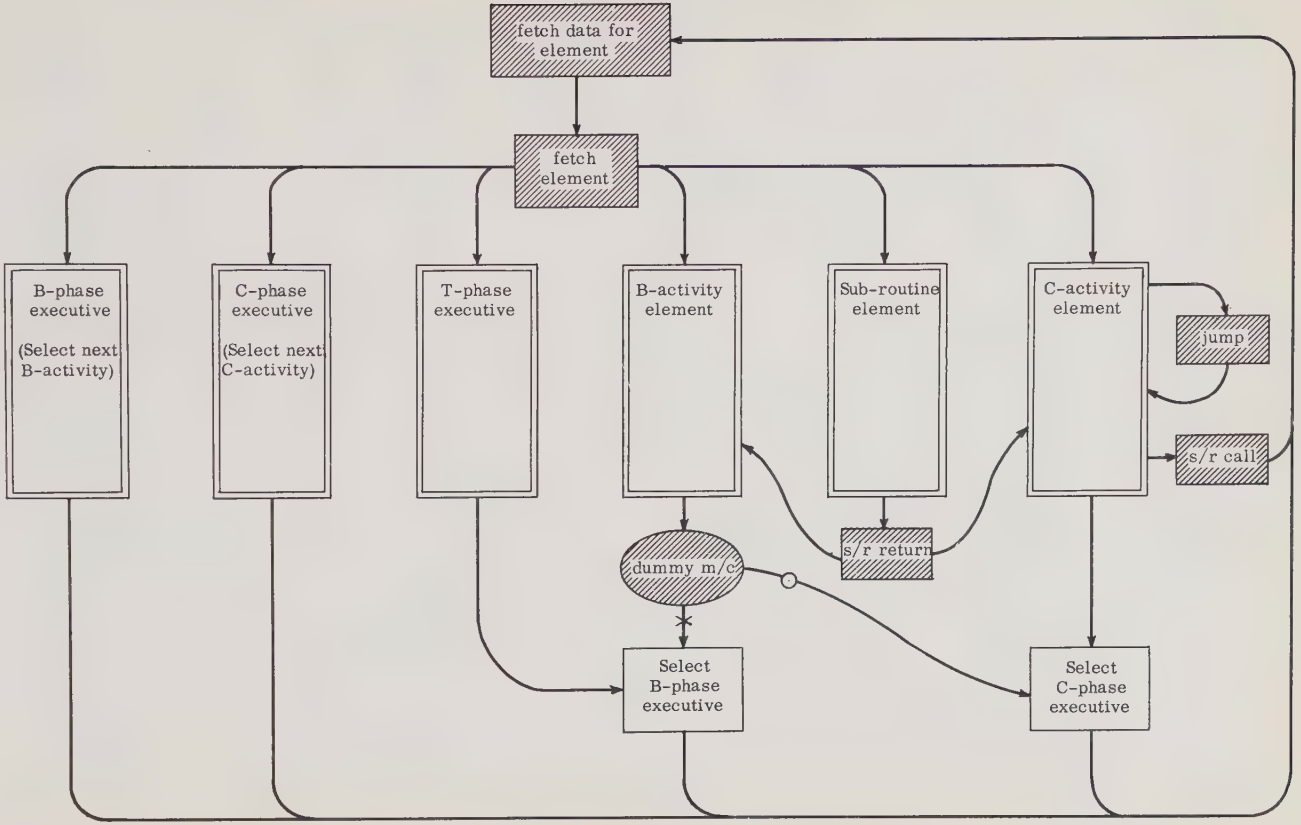


FIG. 4. Structure of general simulation program.

pressed (also possibly automatically). At each clock pulse, the stations are scanned and if any are marked, a time message is punched out from a master clock, followed by a message from each marked station.

Each message consists of a starting symbol, two symbols to identify the station (the clock is regarded as a special station) a separating symbol and the state(s) of the machine.

In the analysis program, a block is assigned for each station, and when the tape is read, each message is decoded. If it is not the clock the states are entered after decoding into the cells of the block.

The new message is compared with the old state for the cells relevant to the analysis and if a significant change takes place, the station number is added to a list. When the clock message is read, this is also decoded and the time stored. The master clock of the analysis is set to the old recorded value and the analysis proper starts.

This takes the stations in turn and tests if they have been changed. If not, the next station is taken. If a change has occurred, then a program element is brought down and the necessary calculation made. Before leaving the element a change is made, if necessary, to the element which will be brought down by the next change in this station.

This continues until the complete list of stations has been scanned when the input program is resumed, and the cycle continues until the tape is exhausted.

The function of the decoding routines is to enable the message to be recorded in the most convenient way and yet the analysis to proceed on data which is in an acceptable form for the natural language.

Figure 5 illustrates the general scheme which again fits the general pattern of Figure 3. Monitoring programs can be incorporated in the usual way already described.

A COMPILER FOR STRUCTURED PROGRAMS

The value of structured programs is that the compiling task is broken up into individual sub-programs which can be translated without fear of side effects.

A knowledge of the structure enables a programmer in the source language to predict exactly the order his elements will

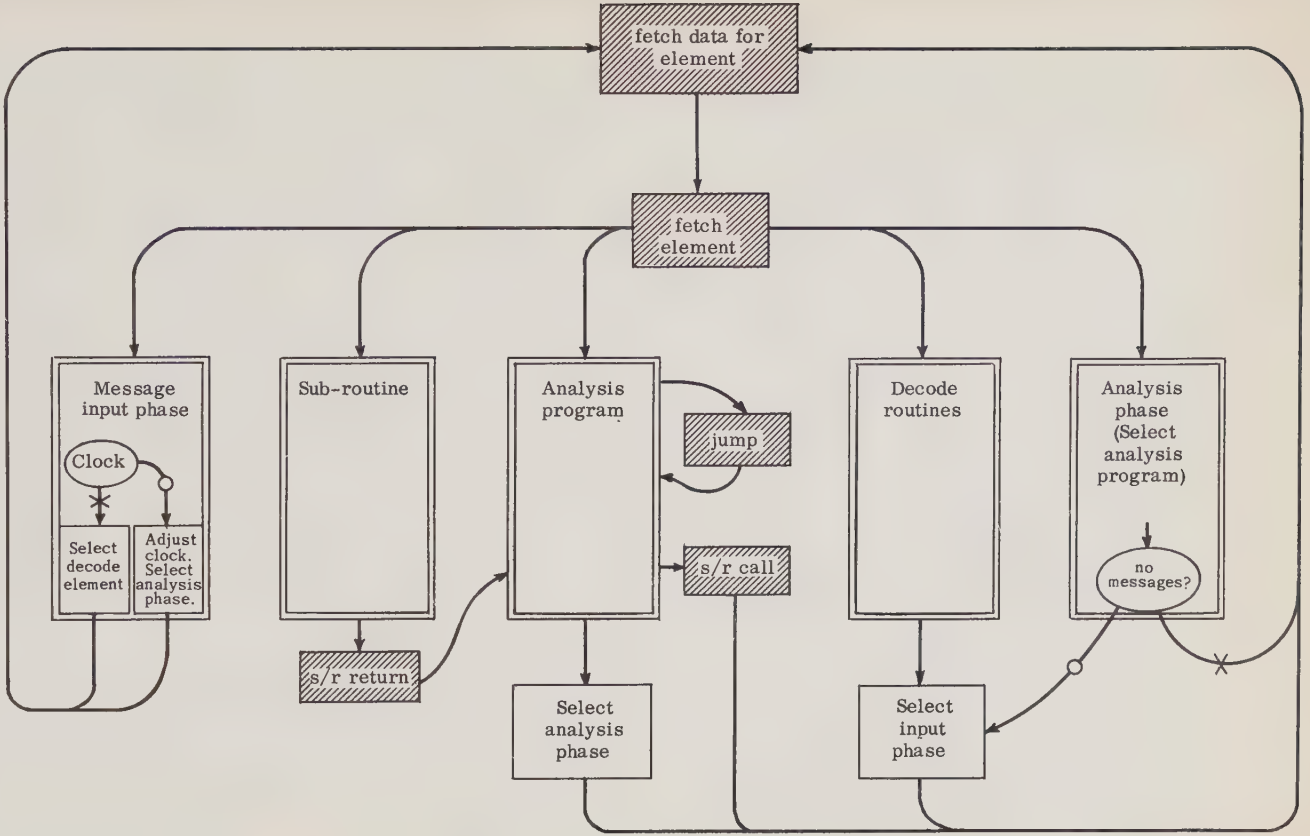

fetch data for element
fetch element
Message input phase
Clock
Select decode element
Adjust clock. Select analysis phase.
Sub-routine
s/r return
Analysis program
jump
s/r call
Select analysis phase
Decode routines
Select input phase
Analysis phase (Select analysis program)
no messages?

FIG. 5.

be used and hence to determine the contents of his data cells at the time of execution of each element.

The structure imposes a common approach to all problems of a class which far from being restrictive enhances the power of the programmer who brings to bear all his past experience of similar problems. The temptation to invent new methods for each new problem is frustrated.

For the two systems, discussed above, the form of arithmetic required is simple non-floating integer arithmetic with some Boolean functions.

The next sections will deal with the construction of a compiler for such a system but the method to be described is not limited to integer arithmetic and a compiler to deal with floating point fractions is under consideration. The central problem for this compiler is to provide a meaningful structure to it.

THE METHOD OF COMPILATION

It is generally accepted that the ideal translation procedure would read successive symbols of the statements in the source language and immediately write the necessary orders in the object language. The difficulties in the way of achieving this ideal stem from the connected nature of the problem being solved and the reflection of these connections in the statements to be translated.

These arise primarily at two levels:

(i) one statement refers to another statement or group of statements. Examples of these are jump statements and calls for subroutines or procedures.

(ii) The syntactical meaning of a symbol can only be made unambiguous by symbols read later.

The first type of connection raises problems of storage allocation. If material has not yet been translated, it is not possible to refer to it directly, since the storage to be allocated to it has not been fixed. This can be overcome by an indirect referencing scheme.

The second type of difficulty involves remembering the ambiguities, and delaying the completion of translation until the ambiguity resolving symbol is read. It is always possible to eliminate the ambiguities by restricting the source language or by increasing the alphabet of symbols and complicating the syntax. Neither of these possibilities is very attractive, as the first

case leads to clumsy and lengthy formulations of problems, and the second makes the source language difficult to learn.

Several attempts have been made to systematise the memorising of the sequence of symbols in a statement. The most important of these is the table look-up method which effectively translates the statement into reversed Polish notation and applies the rules of this language to signal when each object is complete and to cause the correct combination of successive objects.

The mechanics of this is now described since the method to be developed is based on it. The symbols of the language is divided into connectives, brackets and object symbols. In what follows, it is assumed that the set of symbols defining an object has been replaced by a single symbol. Denoting a connective by θ and an object by x a statement becomes a string such as

$$\theta x \theta (x \theta (x \theta x) x \theta x) \theta$$

The connectives are given values in a table. A simple example is given below

θ	$, \rightarrow + x /$
v	0, 1, 2, 3, 4

This is arranged so that the connectives of higher value correspond to those of higher order in the syntax. A count is made of the brackets, up for) and down for (, in a register called b. A second register s is used to control the row of the table used and corresponds to an object count.

A three entry table of bracket level, connective and object is used to store the symbols read. If a bracket is read, no entry is made in the table, but the bracket count is adjusted. If an object is read, the object count is increased and the object is entered in the new row of the table. If a connective is read, this is entered in the current row of the table.

The program shown in Figure 6 is then entered. In what follows, the tests will be abbreviated to

$$(b, \theta)_S \leq (b, \theta)_{S-1}.$$

As an example, consider the statement

$$x_1 \rightarrow (x_2 + x_3 \times x_4)/((x_5 + x_6) \times x_7) + x_8 \times x_9$$

The succeeding states of the table are given below:

s	b	θ	0
0	0	$\rightarrow$	x_1
1	1	+	x_2
2	1	$\times$	x_3
3	0	/	x_4

b	θ	0
0	$\rightarrow$	x_1
0	/	x_2'
2	+	x_5
1	$\times$	x_6

b	θ	0
0	$\rightarrow$	x_1
0	/	x_2'
1	$\times$	x_5'
0	+	x_7

b	θ	0
0	$\rightarrow$	x_1
0	+	x_2''
0	$\times$	x_8
0	,	x_9

b	θ	0
0	,	Z

$x_2' = x_2 + x_3\, x_4$; $x_5' = x_5 + x_6$; $x_2'' = x_2'/x_5' \times x_7$:
$Z \equiv x_1 \rightarrow x_2'' + x_8 \times x_9$.

It should be noted that the table reaches a depth of 4 and that the first entry is not disturbed until the end of the translation.

If the order of terms is altered in this expression to say

$$x_1 \rightarrow (x_8 \times x_9) + (x_2 + x_3 \times x_4)/(x_5 + x_6) \times x_7$$

the depth is increased to 5.

A method of anticipation has been introduced by Samelson & Bauer (4) who utilise the fact that if the bracket count goes down a combination is bound to be called for by the table conditions. It is also possible to combine an object directly it is read and not enter it into the table. By considering the connections in

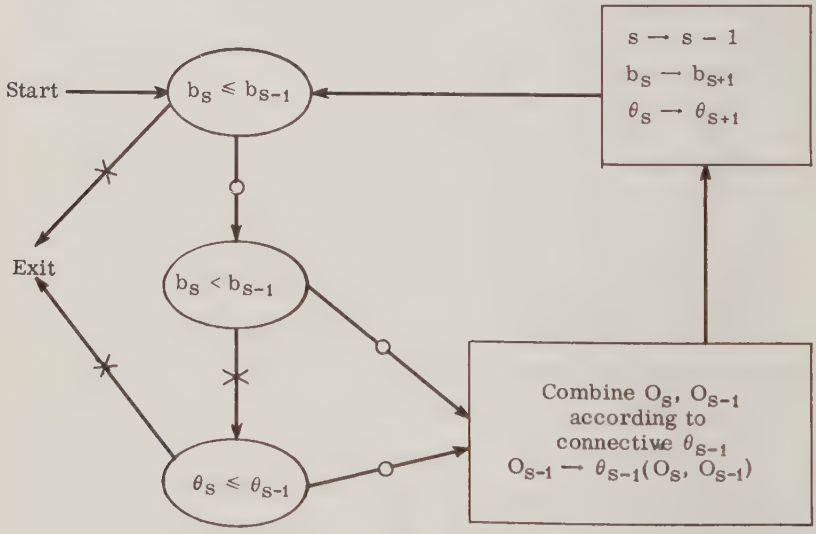


FIG. 6. Object combination logic.

pairs, a further reduction in the table entries is possible. These reductions in the table size are only achieved at the expense of a more complicated logic in the compiler.

At this point, we must consider the storage allocation for the compiler itself. In a two-level machine with a small fast store, it is desirable to make the logic threading part of the compiler and the temporary storage for the connection table (or its equivalent) as small as possible.

The compiled program will have to be kept in the slow access store and this space can be used to hold material prior to the actual compilation of orders.

A further consideration is the problem of recovery if a mistake or inadmissible statement is made in the source language. With paper tape input, immediate correction of the error is difficult and a new run is required. Thus a special recovery procedure can be dispensed with without great loss. This makes it desirable to detect errors at an early stage before the statements are subjected to processing. The space reserved for the final compiled program can be used to hold an intermediate form of the source statements.

We are led to a two-pass program for the compiler. The first pass from paper tape detects as many inadmissible statements as possible and translates into a form which enables the second stage to use the intermediate form with the minimum of logic unravelling and the minimum amount of temporary storage.

The intermediate form can be regarded as both data on which the final program acts and as a set of instructions to reduce the bulk of the logical program.

The method to be described, in fact, constructs a program from the source statements and this program is obeyed by a simulated computer at the second stage.

Another advantage of the two-stage process is that the ending of statements can be detected and the program read in reverse order so that the name of the subject in each statement is not revealed until the object has been formed. This gives a further reduction in the necessary table size.

To illustrate the method, consider a language with objects x_i, brackets and connectives,; $\rightarrow$; +; x; / only. This would, of course, be useless for a real compiler but this simple example will enable the principle of the compiling method to be seen without a clutter of detail.

At the first stage, the brackets are regarded as two extra connectives and then every statement is of the form

$$\theta_1\ x_1\ \theta_2\ x_2\ \theta_3\ x_3 \ldots x'_n\ \theta_{n+1}\ \theta_1 = \theta_{n+1} = \text{`,'}$$

Some of the x's may of course be null, e.g. the first and penultimate terms of $((x_1 + x_2)/x_3) + x_4$.

It is more convenient to describe next the pseudo computer simulated in the second phase.

Let b, s denote the count registers for brackets and objects, O_S denote the object (usually a cell name) at s^{th} level, b_S, θ_S denote table entries of bracket count and connectives.

The program consists of an ordered sequence of word pairs A, S. The word A denotes an object and S is a word of the form

$$\varphi\colon\ t_1,\ t_2,\ t_3,\ t_4, \ldots t_n$$

the limit on n being placed by the word length of the computer and the size of the bytes t and φ. The symbols t denote tasks according to the following table:

t	Task
0	Bring down next word pair
1	$b \rightarrow b + 1$
2	$b \rightarrow b - 1$
3	$s \rightarrow s + 1$
4	$s \rightarrow s - 1$
5	$b_S \rightarrow b$, $\theta_S \rightarrow \varphi$
6	$\theta_S \rightarrow \theta_S + \varphi$
7	See Figure 7.
8	$0_S \rightarrow \theta_S(0, 0_S)$
9	$0_S \rightarrow 0$
10	Book-keeping for next statement

The pseudo computer starts with the first word pair and executes the tasks $t_1, t_2, \ldots t_n$ in order. The words t are so arranged that A and φ are only used once in the succession of tasks. After exhausting these tasks, the next word pair is brought down and the process repeated.

To construct the sequence of order pairs, the source statement is read in phrases $\theta_1\ x_1\ \theta_2$, $\theta_2\ x_2\ \theta_3$, $\theta_3\ x_3\ \theta_4$ and a key word fabricated by assigning values to each θ and to the type of object involved. In our simple example, there are only two types of object—null ones and cell names. The key-word is used to extract an S-word from a table. The A word is constructed directly from the object symbols.

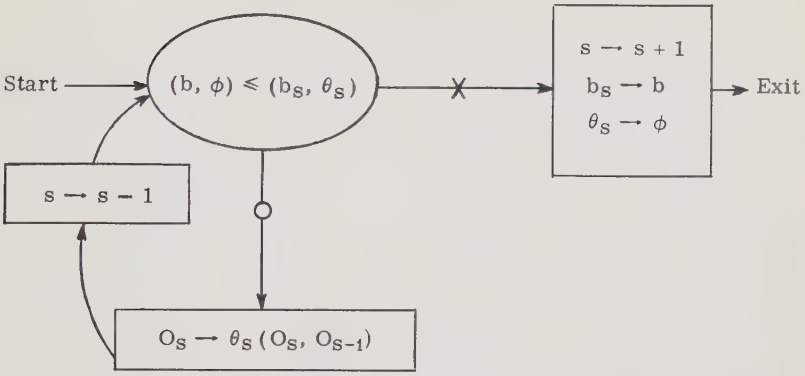


FIG. 7. Task 7 for pseudo-computer.

In this and in more complicated examples a comma or line-feed is used to denote the end of the statement. For phrases of the form θ_X, in addition to the normal process of adding an (A, S) pair to the list, the location of this pair in the list is noted in a separate list.

In the second phase, the scanning and obeying of the program is done backwards until a terminating symbol is used. The location list is then consulted to skip forward to the end of the next statement.

Certain phrases will be inadmissible and the corresponding S words are marked in some way (a negative sign bit is usual) so the reading program can detect it and halt the translation. In an acceptable program of A, S pairs, all the S's will be unmarked.

This two-phase structure has an important influence on the correction of source statements. For an amendment, the offending statements are named by sequence number and the alternatives provided. The first pass will form a piece of program for these.

The second phase then merely needs simple book-keeping to enable it to switch from one program segment to the other as required.

Table I gives the S-words for the simple language described, but the φ symbols are left uncoded to values for ease of interpretation. The code symbol m is used for forming subjects in a manner described later.

TABLE I

List of Key Words for Simple Compiler

,x,	Alarm		, ,	Alarm	
,x+	m	6,8,10	, +	Alarm	
,x×	m	6,8,10	, ×	Alarm	
,x/	m	6,8,10	, /	Alarm	
,x)	Alarm		,)	Alarm	
,x(	Alarm		, (	Alarm	
,x→	0	6,8,10	, →	Alarm	
+x,	+	9,5	+ ,	Alarm	
+x+	'	8	++	Alarm	
+x×	+	8,5,7	+×	Alarm	
+x/	+	8,5,7	+/	Alarm	
+x)	+	1,3,9,5	+)	Alarm	
+x(	Alarm		+(	+	2,7
+x→	Alarm		+→		
×x,	×	9,5	×,	Alarm	
×x+	×	3,9,5	× +	Alarm	
×x×		8	××	Alarm	
×x/	×	8,5,7	×/	Alarm	
×x)	×	1,3,9,5	×)	Alarm	
×x (	Alarm		× (	×	2,7
×x→	Alarm		×→	Alarm	
/x,	/	9,5	/,	Alarm	
/x+	/	3,9,5	/+	Alarm	
/x×	/	3,9,5	/×	Alarm	
/x/		8	//	Alarm	
/x)	/	1,3,9,5	/)	Alarm	
/x(	Alarm		/(	/	2,7
/x→	Alarm		/→	Alarm	
)x,	Alarm		),	0	4
)x+	Alarm		)+	0	0
)x×	Alarm		)×	0	0
)/	Alarm		)/	0	0
),)	Alarm		))	0	1
)x(	Alarm		)(	×	2,7
)x→	Alarm		)→	Alarm	
(x,	Alarm		(,	Alarm	
(x+	0	8,4	(+	Alarm	
(x×	0	8,4	(×	Alarm	
(x/	0	8,4	(/	Alarm	
(x)	Alarm		()	Alarm	
(x(	Alarm		((	0	2
(x→	Alarm		(→	Alarm	

TABLE 1 (continued)

→x,	→	3,5	→,	Alarm	
→x+	→	8,5,7	→+	→	0
→x×	→	8,5,7	→×	Alarm	
→x/	→	8,5,7	→/	Alarm	
→x(	Alarm		→(	→	2,7
→x)	Alarm		→)	Alarm	
→x→	Alarm		→→	Alarm	

Consider the statement used as an illustration earlier. This generates the program:

	S					
Phrase	φ:	t_1	t_2	t_3	t_4	A
× x_9 ,	×	9	5	0	0	x_9
+ x_8 ×	+	8	5	7	0	x_8
) +	′	0	0	0	0	0
× x_7)	×	1	3	9	5	x_7
) ×	′	0	0	0	0	0
+ x_6)	+	1	3	9	5	x_6
(x_5 +	′	8	4	0	0	x_5
((	′	2	0	0	0	0
/(	/	2	7	0	0	0
)/	′	0	0	0	0	0
× x_4)	×	1	3	9	5	x_4
+ x_3 ×	+	8	5	7	0	x_3
(x_2 +	′	8	4	0	0	x_2
→ (	→	2	7	0	0	0
, x_1 →	m	6	8	10	0	x_1

The successive states of the 3-entry table as the program is obeyed are shown in the top entries of Table II.

If the terms in the statement are reversed

$$x_1 \rightarrow x_8 \times x_9 + (x_2 + x_3 \times x_4)/((x_5 + x_4) \times x_7),$$

a different program, shown below, is generated and the table entries become as in the lower entries of Table II.

	φ	t_1	t_2	t_3	t_4
),	′	4	0	0	0
x_7)	×	1	3	9	5
)×	′	0	0	0	0
+x_6)	+	1	3	9	5
(x_5+	′	8	5	0	0

	φ	t_1	t_2	t_3	t_4
((	'	2	0	0	0
/(	/	2	7	0	0
)/	'	0	0	0	0
$\times x_4)$	$\times$	1	3	9	5
$+x_3\times$	+	8	5	7	0
$(x_2+$	'	8	4	0	0
+(	+	2	7	0	0
$\times x_9+$	$\times$	3	9	5	0
$\rightarrow x_8\times$	$\rightarrow$	8	5	7	0
$,x_1\rightarrow$	0	6	8	10	0

Thus the table entries have been reduced in either case and the fast storage needed during compilation is reduced.

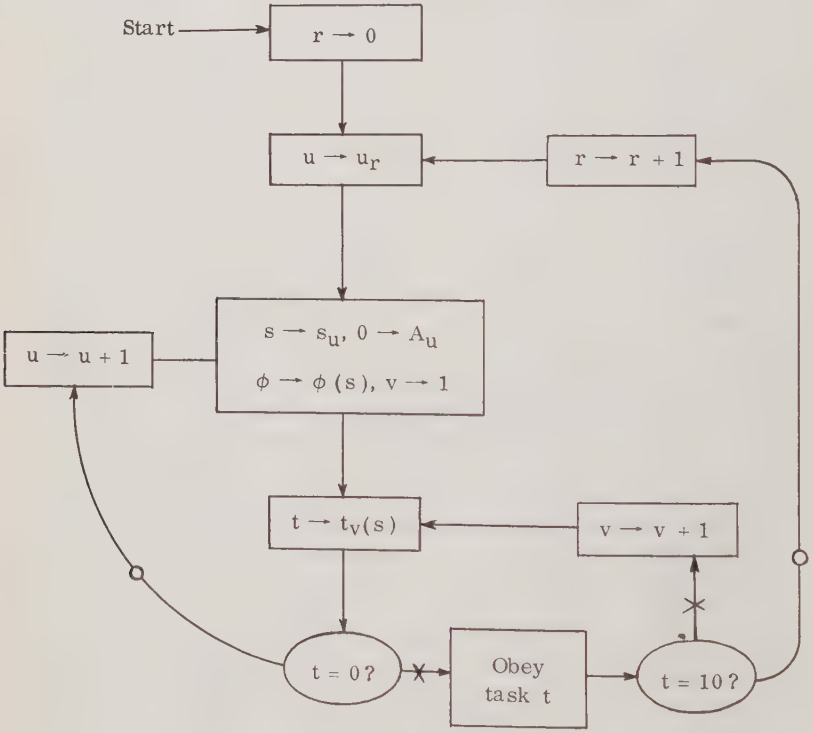


FIG. 8. Pseudo-computer compiler logic.

TABLE II

Successive States of (b, θ, 0) Table for Sample Expressions

b θ 0	b θ 0	b θ 0	b θ 0	b θ 0	b θ 0	b θ 0
$0 \times x_9$	$0 + x'_9$	$0 + x'_9$	$0 + x'_9$	$0 + x'_9$	$0 + x'_9$	$0 \rightarrow x''_7$	$0 \rightarrow^* X$
	$1 \times x_7$	$1 \times x_7$	$0 / x'_7$	$0 / x'_7$	$0 / x'_7$		
	$2 + x_6$	$2 + x'_6$	$1 \times x_4$	$1 + x'_4$	$1 + x''_4$		

$x'_9 = x_8 \times x_9 \quad x'_6 = x_5 + x_6 \quad x'_7 = x_7 \times x_6 \quad x'_4 = x_3 \times x_4 \quad x''_4 = x_2 + x'_4 \quad x''_7 = x''_4 / x'_7 + x'_9 \quad X = x_1 \rightarrow x''_7$

b θ 0	b θ 0	b θ 0	b θ 0	b θ 0	b θ 0	b θ 0
$1 \times x_7$	$1 \times x_7$	$0 / x'_7$	$0 / x'_7$	$0 / x'_7$	$0 + x''_7$	$0 \rightarrow x'_9$	$0 \rightarrow^* X$
$2 + x_6$	$2 + x'_6$	$1 \times x_4$	$1 + x'_4$	$1 + x''_4$	$0 \times x_9$		

$x'_6 = x_5 + x_6 \quad x'_7 = x_7 \times x'_6 \quad x'_4 = x_3 \times x_4 \quad x''_4 = x_2 \times x'_4 \quad x''_7 = x''_4 / x'_7 \quad x'_9 = x_8 \times x_9 + x''_7 \quad X = x_1 \rightarrow^* x'_9$

The significance of the value m of φ can now be explained. It is obviously desirable both from the point of view of the user and the efficiency of the object language translation that statements like

$$x_1 \rightarrow x_1 + x_2;\ x_1 \rightarrow x_1 \times x_2;\ x_1 \rightarrow x_1/x_2$$

should be abbreviated to

$$x_1 + x_2;\ x_1 \times x_2;\ x_1/x_2$$

Now, however, the combination rule for +, ×, / are not as usual and the value m is added to the value for +, ×, /, so that the appropriate combination rule is used when it is recognised that the new object x in the phrase ,$x\theta$ is actually a subject.

The executive program for the compiler is particularly simple.

Slow Access Storage

Pseudo-computer Program.

$A_1\ S_1$
$A_2\ S_2$ } 1st statement
:
$A_{u_0}\ S_{u_0}$ } end of 1st statement

$A_{u_0+1}\ S_{u_0+1}$ } 2nd state-ment
:
:
$A_{u_1}\ S_{u_1}$ } end of 2nd state-ment
:
:

$A_{u_n}\ S_{u_n}$ end of $(n+1)^{th}$ statement

List of statement endings.

u_0
u_1
u_2
:

Fast Access Storage.

b, s, φ, t, 0, S
$b_0\ \theta_0\ 0_0$
$b_1\ \theta_1\ 0_1$
:

Pseudo-computer Simulation Program

See Figure 8.

Before this scheme can be used as a practical compiler, it must

be shown that certain necessary compilations can be accommodated within this framework. These can be categorised as:

(i) a full range of possible objects can be used;
(ii) parameters of subroutines can be specified;
(iii) tests and jumps are possible;
(iv) declarative statements may be made;
(v) negative quantities can be handled.

The objects which are needed are:

(i) specified numbers;
(ii) variables - contents of the registers or cells;
(iii) variables calculated by subroutines or procedures.

The usual device of demanding that the name of a variable must contain at least one non-numeric symbol can be used. The block structure of the slow access store where the majority of the variables are stored gives a natural two-dimensional array. It is convenient to letter the registers of a block and number the blocks. Thus in the Ferranti Pegasus which has 8 registers to a block, eight letters S - Z (to include the conventional X, Y, Z of algebra) are used, and a cell is denoted for example as U123, being the third register of the 123rd block.

It is also advantageous to be able to specify fast access storage. This can be done by holding a copy of one block in the fast store, and/or by using special names for working space. The copy may be named as So . . . Zo or without loss of unambiguity as S, . . . Z. For working space, other capital letters may be used, or better, the lower case letters on the input mechanism may be used. With a normal 5-hole teleprinter code, n and v are available and these can be followed by a number if more than two cells are required.

Subroutines can then be specified by single capital letters not already allocated and by letter combinations both of which can be followed by a distinguishing number, if required.

Thus the type of objects may be classified as

Type	Description
0	digit 0 or no object.
1	digit x ($\neq 0$).
2	fast access store (L = S, . . . Z).
3	ordinary register Lx.
4	subroutine M or LL . . . L ($M \neq S \ldots Z$).
5	subroutine Mx or LL . . . Lx.
6	working register nx, or vx.

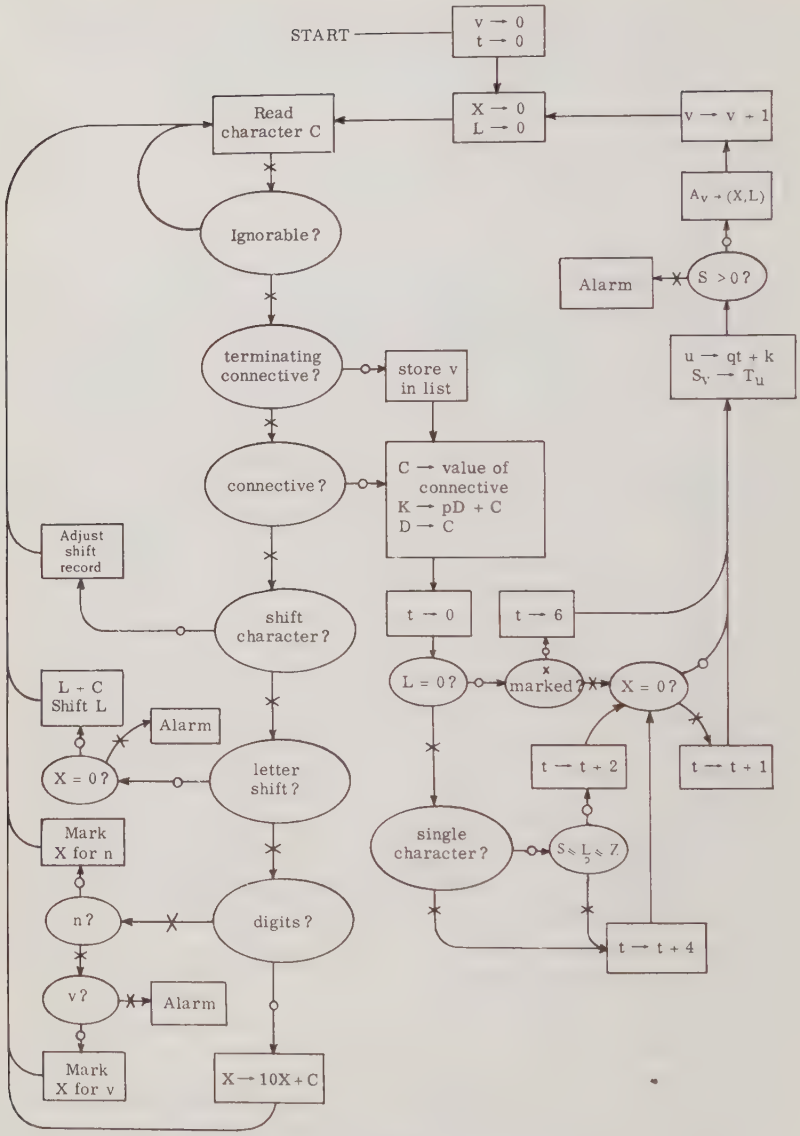


FIG. 9. Typical reading program.

It must now be established that no ambiguity can arise in class 0. This follows since no two operational connectives can appear without an object separating them, no object can appear between an operational connective and an opening bracket, and a closing bracket cannot immediately follow an operational connective. Treating a terminating connective as a form of bracket, this exhausts the possibilities.

A typical reading program to distinguish these types of objects and to form the pseudo-program is outlined in Figure 9.

X and L are used to collect the letters and digits of the object, D remembers the preceding connective, v is a pseudo-program counter, k, u, t are working spaces and p, q are suitable constants (usually binary powers). T_u are a table of all possible S words for all possible phrases.

For the creation of iterative loops and general economy in program writing in the source language indirect naming of cells is essential. For complete flexibility both the letter and numerical designation of a cell should be indirect, but the restriction to fixed letters is not found very restrictive in real problems as for mnemonic reasons different letters often represent functionally different quantities.

Thus the compiler must be able to cope with expressions like
θ T (U3 + n + 5) θ'
The reading program breaks this into phrases
+5)θ'
+n+
(U3+
θ T (
The last phrase is normally unacceptable.

If we now insert a new operation ρ which combines the object T with the object U3 + n + 5 by finding the quantity in cell T of block U3 + n + 5, the phrase θ T (can be rewritten as θ Tρ (and the S word corresponding to it made an amalgamation of those for the two acceptable phrases θ Tρ and ρ (. The S words must now be modified to contain two φ values or the order code extended to deal separately with the special value ρ.

The latter solution introduces a new task 11, say

11 $\quad 0_S \rightarrow \rho(0, 0_S)$

The phrase , T (needs special treatment. As usual, it is treated as an amalgam of , Tρ and ρ (. Now , Tρ must cause 0_1, the re-

sult of task 11 to be combined by the modified θ_0 with 0_0. This involves a new task 12

12 $\quad 0_S \rightarrow \theta_S(0_{S+1}, 0_S)$

and so the S-word for ,T(becomes

m: 11, 4, 12

When only one term defines the block number, it would be desirable to omit the brackets. This is not possible in T(S3) say since TS3 is the name of a potential subroutine. However Tn($\equiv$ Tn$_0$) or Uv3 are permissible at the expense of a further class of objects and an extension of the basic table of possible S-words.

The extension to greater depth e.g. U(n+W(T5+3)+7) does not create any new problems.

We now turn to the specification of parameters in subroutines. Provision must be made in the executive program for storage of the parameters in fixed locations or alternatively a single parameter must be provided to specify the location of the parameters. In either case, the compiler must prefix the call for the subroutine by orders which dump the parameters in the cells specified or selected.

A natural notation would give rise to an expression of the form:

θLL . . . L(x_1, x_2, . . . , x_n) θ'

The comma previously used as a terminating connective now has a second meaning. Ambiguity is avoided, however, since the bracket count is non-zero in this later context. The reading program has to be modified to detect a ",," or a "line feed" at zero bracket count as the end of a statement.

The breakdown of this expression gives rise to four new phrases θLLL(; (x;,;, x,;, x). None of these has had a previously assigned meaning so no ambiguity is introduced. However, additional tasks are required in the order code to deal with writing the object dumping orders. Also a task for comma counting is required so that the location for the successive parametric objects can be remembered. Since the number of parameters is not revealed until the phrase (x, is read, the parameters are stored in reverse order to that specified but this is no disadvantage since the subroutines can be easily written to take this into account.

There is no limitation on the objects. If a cell is named indirectly, the phrase ,L(is generated and the comma count must be used to distinguish from the earlier interpretation. The phrase), cannot now have a null S-word but will introduce a comma count.

If the parameter is specified as a subroutine, no further difficulties arise and this may itself involve parameters. Now, however, provision must be made for specifying in the executive program where the parameter list for the inner subroutine starts. This is easily calculated from the comma count, providing the comma count at each phrase), is remembered.

An additional task is required in the pseudo-order code to perform this function. A limit on the total number of parameters must be made in the syntax of the source language to prevent the space allocated in the executive program being overfilled. Alternatively a count of space required can be left and the executive adjusted. This is usually not practical if the fast access store is limited.

The specification of the parameters of a subroutine by subroutines must not be confused with calls for subroutines within subroutines (if these are written in the source language). The feasibility of this depends on the storage space for parameters. With limited storage, the parameters of the outer subroutines will have to be stored in the slow access storage of the computer and restored at the conclusion of the inner subroutine. An address for this storage will be an additional hidden parameter of all inner subroutines. A count of depth will be needed and any more complex structure than simple nesting will cause considerable complications.

Unless some automatic link feature is built into the object computer, a conventional dump for a link is required and the compiler must set this prior to calling in the subroutine. Similarly, a convention as to where to dump the object calculated by the subroutine must be established.

The call for the subroutine written by the compiler must specify the location of the subroutine material. If this specification is indirect in the sense that the required address is found in a location specified by the *name* of the subroutine, then the cell so specified may have the location inserted at any time, before or after the compilation of the statement involving the subroutine call.

Thus the compiler need only code the name into a number (by consulting a list and augmenting it, if necessary) and write orders to form this number and pass control to a common sequence of executive orders which collect and dump the link and use the number provided to consult the directory and bring down the subroutine orders. The subroutine must pass control back to another sequence of executive orders which obey the link. The two sequences may also contain storing and restoring orders for the current block of program if high speed access storage is at a premium. The subroutine is then called in to the normal program space. These groups of orders have already appeared in the master flow diagram, Figure 1.

We must now consider tests and the associated jumps. These require the addition of the Boolean relations $=, \neq, <, \leq, >, \geq$ to be added to the connectives list. A minor difficulty is raised by the absence of $<$ and $\leq$ on standard five-hole equipment, but this is overcome by using $>*$ and $\geq*$ to represent them. A further modification of the reading cycle is now required to detect * and to adjust the connective coding accordingly.

Ideally, we require to write the test in any form we please, but this is difficult with the present method of compiling, which is only simple if the left hand side of the test relationship is restricted to a single element.

To illustrate this, consider the test

$$, x_1 + x_2 > x_3 + x_4,$$

Two new phrases are created $>x_3+$ and $+x_2>$. The first is treated as $\rightarrow x_3+$ with the appropriate combining orders associated with the connective $>$. The phrase $+x_2>$ indicates that the subject involves more than one term and so a new object level is created. The phrase has an S-word

$$+x_2> + 3, 9, 5$$

The b, θ, 0 table now reads

b	θ	0
0	$<$	$x_3 + x_4$
0	+	x_2

The phrase $,x+$ has already been assigned an S-word m: 6, 8 and the task 6 will modify the + sign incorrectly and cause task 8 to change the object x_1 incorrectly. To deal with this task 6

must be modified to add m to θ_0 and the new task 12 used to unconditionally combine 0_S and 0_{S+1}.

6 $\theta_0 \rightarrow \theta_0 + \varphi$
12 $0_S \rightarrow \theta_S(0_{S+1}, 0_S)$

Then the phrase ,x+ has an S-word m: 6, 8, 4, 12, 10

It remains to specify the statement to which a jump is made on the success (or failure according to convention) of the test. The usual convention that on the alternative outcome of the test no jump takes place eliminates a second address. The object statement must be labelled and this is most naturally done by prefixing it with (x) where x is a number. Thus

, S(n+5) → U,

becomes ,(5)S(n+5) → U,

The phrases , xθ are replaced by)xθ. These have no other interpretation and so can have the same S-words as ,xθ. The new phrases (x) and ,(are introduced. The former identifies the label and the compiler must then note the starting address in the coded form of this statement. Since the orders for the statement will be written by then, a note of the starting address must be kept.

This can be avoided if the notation

, (5) ,S (n+5) → U,

is used. The phrases ,(5), are read as a separate statement; and then the current address (advanced by one instruction) can be noted as the first of the next statement.

The S-words for the new phrases are

(x) 0: 1, 13
,(0: 2, 10

where task 13 is the address noting sequence.

The directive to jump to a labelled statement must be appended to a test statement. Brackets are required and these must enclose an object distinguishable from a label. If the matrix letters are chosen to exclude J then the natural mnemonic notation is illustrated by

S < T + V, (J5),

The phrase (J5) is distinguished from a label since J5 is regarded as a subroutine object. The intermediate comma is essential since V(J5) is a possible name for a cell. The separa-

tion into two statements is slightly inconvenient since the combination corresponding to the Boolean relation must provide the test quantity and then after the phrase (Jx) has caused the compiler to set the name of the label, to write the test instruction. This can be achieved most conveniently by noting that the table will not be used by the phrase (Jx). We create a new combination task 14 for writing test instructions jumping to a fixed location and use task 6 associated with a φ value t to transform the previous value. If φ can be allowed to take negative values $t = -m$ is a convenient value. The S-word of the phrase (Jx) is

t: 14, 6, 8

The new combination functions do not use A so its value Jx is immaterial.

Once again an indirect form of address must be used by the labels to enable a single pass process to be written and a further sequence of executive orders used to process the name of the label at run time. These also appear in Figure 1.

The alternative of prefixing the test by the jump address is not so attractive either notationally or from the point of view of the compiled program. The jump label would either have to be held by the program throughout the construction of the test quantity or alternatively the compiler would have to remember it giving an unnecessary connection between statements.

Declarative statements are a form of subroutine which initiate a procedure rather than form an object. Thus the single phrase ,LLL, can introduce them and this has not been used before.

If parameters are required, the natural notation is

,LLL(x,x,x),

This also arises if a parameter is set by a subroutine but the distinction can be made by testing the bracket count on encountering the phrase ,LLL(. A non-zero bracket count implies object dumping after calling in the subroutine. A zero count does not.

The introduction of subtraction and negative numbers needs a little care since the sign '–' may refer to the unitary operation of negation or the binary one of subtraction.

The difficulty is to correctly interpret the statements without adding unnecessary orders to the compiled program.

A new connective - of equal value to + must be introduced.

The new phrases which arise are $-x\theta$ and $-($. All others are non-admissible.

The phrases $-x$, $-x)$ introduce a new task.

$$0_S \rightarrow -0$$

while $-x+$; $-x\times$; $-x/$ will need the task

$$0_S \rightarrow \theta_S(-0,\ 0_S)$$

The phrase $-L($ will be interpreted as $+L($ except that task 12 will be replaced by

$$0_S \rightarrow \theta_S(-0_{S+1}\ 0_S)$$

These new tasks can be avoided if the task

$$0_S \rightarrow -\ 0_S$$

is introduced and then the phrase $-($ is easily interpreted.

However, this solution involves the orders to negate an object when this can be avoided by slight changes of the other combining orders written.

The difficulty centers on the fact that + and − must be assigned equal values φ and since this in due course becomes the appropriate θ_S, only addition can be achieved directly. If different values are assigned to + and − this is avoided at the expense of enlarging the table entries.

The solution lies in adding a second parameter byte ψ to each S-word. One byte is used for the comparison $(b,\varphi) \leqslant (b_S,\theta_S)$. The second is used for setting θ_S; $\theta_S \rightarrow \psi$.

This basic language can be adapted for the various specialised languages by using the non-cell letters as names of special directives which add the necessary facilities.

For example, the directive to bring a row x of the matrix into the high speed location can be denoted by Rx or if the name is only known indirectly by e.g. R(n + W3 + 2).

Before this directive is obeyed, the current row must be restored to the matrix if any changes have been made to it. This must also be done prior to a test and jump. Thus task 10 will need orders to count statements which change the content of the fast access copy of a row. A standard location must be reserved for the name of this row and an order written by the compiler to restore this row. The alternative of reconstructing the value x from the previous R-statement is not satisfactory since if an

indirect naming was used, the cell(s) containing the variable component(s) of the name may have been changed.

A useful directive to add to all specialised languages is one to introduce a cycle. The directive

,C(r,s),

will mean set the conditions for starting a cycle repeated s times on the index nr starting at nr = 0 (There is no loss of generality in a fixed starting point as the addresses of the cells involved can be adjusted.)

The statements following constitute the cycle whose end must be indicated. This can be done by terminating the last statement of the cycle with a '.' instead of a ','. The new phrases introduced have in their S-words an additional task that writes the cycle count and return orders. If test statements are included in the cycle whose failure indicates return to the start of the cycle this can be provided by the convention of leaving such statements unlabelled and omitting the label from the first statement of the cycle.

Providing distinct indices are used, cycles may be nested. More complex logical connections must be synthesized from strings of test statements. The only reason for introducing cycles is to enable special facilities in the basic order code to be utilised.

Another useful directive is a compound test statement. For example,

L(s, Sx = 3, T > 3, n = 0), (J7),

is interpreted to mean that the truth or falsity of the three statements Sx = 3, T > 3, n = 0 shall be tested and the s^{th} of a set of logical truth tables consulted to determine if a jump to the statement labelled 7 shall be made.

Since all Boolean expressions may be expressed in a canonical form, the same effect can be achieved by using a directive 0

0(Sx = 3, T > 3, n = 0), (J7),

meaning that only if all of the three tests are not satisfied is the jump made. This represents the logical form 'OR'. A succession of test statements represents the logical function 'AND'. Thus a succession of OR statements can give the general Boolean expansion.

With the extended language, the phrases ($x\theta$; ,$x\theta$ occur in new contexts but using the bracket count, these can be distinguished from the former cases.

The preceding analysis indicates that this technique of compiling will cope with a reasonably complex language and extensions will not change the central structure of the compiler but merely add to the set of tasks which must be provided and increase the set of possible phrases. The limit is reached when the complete table of phrases occupies too much of the slow speed access store to leave room for the input program.

The logical elements involved in creating the necessary objects and making the correct combinations of them are contained in this phrase table. The tasks induced by the pseudo-computer program fall into two kinds.

i) elementary book-keeping (counts of brackets, commas and objects, entering items in the table);
ii) order writing tasks.

These latter tasks involve other ancillary book-keeping; a record of location of the next order, the number of orders left in each block (so that program transfers can be inserted) indexes for the lists of indirect addresses for subroutine calls and jumps, search routines to identify subroutines and directive names, etc.

There is considerable opportunity for errors in the programming of these tasks, and this is largely independent of the correctness of the logic of the compiler.

A useful ancillary program for the development of a compiler constructed by this technique is a simulator which works identically to the real compiler except that the order-writing tasks are replaced by programs which print the effect of the orders which should be written. Thus for example task 8 is replaced by a program which prints

$$0s \rightarrow 0\ \theta s\ 0s$$

If 0 is an object of subroutine type the relevant phrase in the S-word of +LLL(($0 \equiv$ LLL) will print

$$0s + LLL'$$

The dumps for subroutine parameters must be denoted by new symbols such as Pr and the intermediate dump for subroutine objects given a name D, say. This is usually the accumulator or a fixed one of them in a multi-accumulator machine.

As an example, consider the statement

$$W(n+6) \rightarrow 2 \times W(n+5) + (Z3 \times (SAM(n4, PIG(S, T4, v1)) + W1) + W3$$

This would lead to a printed statement

$O_o \rightarrow W3$.
$O1 \rightarrow W1$
$P_o \rightarrow v1$
$P1 \rightarrow T4$
$P2 \rightarrow S$
$D \rightarrow PIG$
$P_o \rightarrow D$
$P1 \rightarrow n4$
$D \rightarrow SAM$
$O1 \rightarrow D + O1$
$O1 \rightarrow Z3 \times O1$
$O_o \rightarrow O1 + O_o$
$O1 \rightarrow 5$
$O1 \rightarrow n + O1$
$O1 \rightarrow W(O1)$
$O1 \rightarrow 2 \times O1$
$O_o \rightarrow O_o + O1$
$O1 \rightarrow 6$
$O1 \rightarrow n + O1$
$O1 \rightarrow W(O1)$
$O1 \rightarrow * O_o$

The * in the last printed statement indicates that this is a subject dumping operation and not the normal object forming operation.

The program reveals the steps that will be taken by the compiled program to execute the statements and in addition to detecting failures of the logic of the compiler reveals redundant orders that will be generated.

For example, if the computer order code contains an order for accumulated multiplication, the steps

$O1 \rightarrow Z3 \times O1$
$O_o \rightarrow O1 + O_o$

can be compressed to

$O_o \rightarrow O_o + Z3 \times O1$.

This can be achieved by changing the S-words and adding a new task. Similarly the statement

$O1 \rightarrow 2 \times O1$

can be obtained by a binary shift and does not need a true multiplication. This will involve a more complex task for the multiplication combining task.

Since the construction of this program involves very little extra programming effort, it offers a good return on labour at the development stage.

THE MINIMISATION OF COMPILED ORDERS

The process outlined above will, with a minimum of logical program, initiate order writing routines to effect the creation and combination of objects.

For a slow machine, it is desirable that in the compiled program the creation of objects shall be achieved in a minimum time and with the minimum of orders.

It is generally accepted that time and space are inter-changeable commodities in program writing and to present a well formed objective for the compiler these two objectives must be scaled relative to one another.

For the type of two-level machine under consideration, this is comparatively easy, since the executive orders must be punctuated by block transfer orders to bring successive parts of the program into the fast store for execution. Since most of the storage variable must, in a real problem, be in the slow store, it is unlikely that a block of program will be written which does not include a storage transfer order for data. Most slow access stores are magnetic drums or have other similar access characteristics. Thus the interlinking of data and program transfers effectively randomises the access time to the slow store and the delay to be added to each executive time of a transfer order is, on average, half a revolution time (or half a cycle time for other forms of storage).

Now the program executive time can be divided into two components.

(i) the actual execution time of the orders (including waiting time if necessary);

(ii) the execution time (including waiting time) of the program transfers.

Since each block will contain an equal number of orders (for most ordinary codes) the appropriate fraction of the second

component can be added to each order executive time and these augmented values used for timing the program.

Although it is possible that two equivalent programs whose running times calculated in this way are equal could occupy a different number of orders, the variation of length is likely to be small.

Treating the space occupied as constant, the problem is reduced to producing a program whose running time calculated in this way is a minimum.

The alternative ways of combining objects are limited and for each type of object the quickest way can be selected beforehand. The choice will not depend on the actual object once its type is known. For example, there may be fast multiplication by small integers, but unless it is known that the multiplier is a small integer (i.e. the object is a number) no advantage can be taken of this by the compiler.

Thus the opportunities for economy only exist in the creation of the objects. The strategy here is to realise that if for a variable normally stored in the slow store, a copy exists in the fast store, this should be used instead. If a record is kept of the contents of the available fast store locations (i.e. the variable storage space, the working space and the accumulators) during compiling, this record can be scanned prior to creating an object and the object selected or constructed from fast storage if possible.

There must be some limit on the complexity of the contents of the locations contained in the records and it is convenient to limit this to a sum of terms. Beyond this form the contents are taken as unknown.

Thus the contents of a location are either

(i) a number;
(ii) the contents of a fast store location;
(iii) the contents of a slow store location;
(iv) the sum or difference of preceding forms;
(v) unknown.

The slow store location may itself be named indirectly. Because the language will not allow indirect letter mention, only a range of names of rows need be remembered. A convenient limit is a sum of terms of types (i) and (ii).

Thus we require to remember names like W5, T(n+S+4) but not those like U(n+S4+U) or U(n+S(n+5)).

There must be some further limit on the number of terms that can be remembered for any location. A convenient limit is that only one slow access location can be remembered and that only one occurrence of a high speed location is allowed. Thus

W3+S+n+v2+6, W(n+S) – S+n3 – 5, S+T–U+v – 6

are allowed but

W3 + S + n + S, W3 + W1 – S, W(n + S + 2 + S) + T

are not.

Although it is unlikely that a term S + S will be written in the source language, it is possible for such quantities to be held in working storage during the execution of the compiled program.

It is now possible to remember the contents of a location in four words:

(i) the value of the fixed part (stored in normal form);
(ii) the fast store locations occurring positively (in the form of a key-word);
(iii) the fast store locations occurring negatively (in the form of a key-word);
(iv) the slow store location (the letter and fixed part of row number in decimal code, the variable part of the row in a key-word, sign bit to denote sign of term.

The locations which have a name in the source language now can be represented in two forms - its own name e.g. S or its contents in terms of the other locations.

The degree of sophistication used in this memory sensitive compiler depends on the speed required in the compiler. If the machine is fast enough, all fast access locations can be remembered, but for a faster compiling process the memory can be restricted to the working space not nameable in the source language.

The first essay in this technique has accepted the latter simpler procedure. The question of aliases does not then arise.

Since the memory allows sums of terms to be remembered, the combination of objects by addition is delayed until the complete set of objects to be added (or subtracted) is known.

When the object is finally defined, a selection of the location in which to built it up must be made. Two objectives arise:

(i) to choose a location which will need the minimum time to convert to the required form;
(ii) to choose a location which will destroy forms of least value in later constructions.

A possible approach to this is through the notion of the value of the current state of the locations with memory record.

For each location, the time necessary to execute the orders to fill the locations with their current contents is calculated. The sum of these is taken as the value of the current state. In this calculation, locations whose contents are unknown are taken to have zero value.

For each possible selection of location for the object, the time to execute the orders to change the contents to the required value is calculated. For locations of unknown content the first order sets a value in the location. For other locations, the first order adds or subtracts a value to or from the location. The values of the resulting configurations are then calculated. All the individual values necessary are available. The changes in value of the state are calculated. We now have to balance loss of value of the state against the actual execution time for the change and in doing this some account must be taken of the fact that none of the present states may actually be required. In a crude way, this can be done by discounting the change of value and adding execution time to it to give a single measure. The location with minimum value for this measure is then chosen.

The choice of the discount factor is arbitrary and can only be found by experiment. In an ambitious system, the performance of the compiler could be recorded by the program itself and trial adjustments made to minimise the execute time of the orders written. Although this adaptive scheme is attractive at first sight, this success would depend in some ill-defined sense of the "sameness" of the successive programs compiled.

The ideal scheme would look ahead to determine the actual forms which will be required and to use this in selecting the location to be used in the creation of any object. This would mean a multiple pass process and considerably slow the compiler down. It would also raise questions of the depth ahead which should be explored and the route to be followed at tests and this would also lead to the necessity of adaptive features.

The actual processes under construction use the principles of heuristic programming.

The locations are numbered 1 to m, say, and a fictitious location O is introduced. This is always recorded as containing zero.

The factors to take into account are:

(i) the actual execute time;

(ii) the likelihood that the contents of the location will be useful;
(iii) the value of the destroyed contents of the location.
(iv) the likelihood that the location will be used for dumps and fixed purposes.

We assume the locations can be ordered by the likelihood of being used for other purposes. Since fixed locations are used for each type of dump, this assumes the statistical frequencies of occurrences of subroutines and tests can be estimated for the class of problems under discussion. If competing locations cannot be distinguished on other criteria then this last will be used, to ensure that on average, the contents are preserved for later use as long as possible. (The contents for dumps and fixed purposes, being usually constants, have the lowest value possible).

The actual executive time is calculated for the location O and all known locations. If the location is a current object awaiting combination, the execute time is adjusted to allow for the extra transfer to a non-object location.

The primary judgement is on actual execute time. Suppose there is a unique location with minimum execute time. If this is a known location and not a current object, we choose this. If it is a current object, we must choose a location in which to copy it. If the minimum is achieved by location O we must choose a location in which to build the required value up. If there are any non-object locations of unknown content, in both cases we choose the last of these. If there are no locations of this kind, we must choose a known non-object location. Failing this, the compiler cannot proceed.

Now consider the possibility that there are several locations which achieve the minimum value. If one of these is location O, proceed as if this were unique. If not, we choose a non-object location. If none exist, we choose the last location of unknown content and start to build up the quantity required by dumping any candidate in it. Failing this, the compiler cannot proceed.

It remains to decide how to select one of a set of locations of known content. We now require to estimate the likelihood that a term will be useful. To do this, we appeal to the nature of the source language. If a location contains a large number of terms, the author of the statements, being naturally economical in his writing, is likely to define it by collecting the quantity in a working space and use the working space for later references.

Thus the number of terms rather than the value is relevant in the likelihood of its re-requirement, and this diminishes as the

number of terms increases. If this criterion does not resolve the choice, then the value of the remaining candidates are used to discriminate. Finally the ordering is used, if necessary.

The structure of the compiler pseudo-computer code has to be altered quite considerably. Associated with each object level there is a set of four locations, called the Q set. Phrases of the form ± x ± ; ± x) ; ± x , call in tasks to adjust the appropriate Q cell according to the type of object. The phrases (x ± ; → x ± ; ≤ x ± ; , x ± etc. call in a task which selects a location for the object in accordance with the scheme outlined above and writes the orders to execute the creation of the object.

Phrases such as × x ± ; / x ± raise the object level and form the new object, as before. The phrases ± x × ; ± x / now demand the combination of two objects, the result of which is to be added to an object which may not exist. The level associated with the combined object is reduced and then when the Q set for the object is finally completely specified, it is added to this object. If an object combination is called a second or later time, the addition must be made immediately.

Thus the location selection procedure must be modified to take account of this possibility. If an object has already been allocated a location, the choice by the more complex rules is ignored.

The process of storing the terms of a sum in a Q-set can lead to an impossible adjustment. If a slow access location is added to a Q-set which already contains such a term, or if a particular fast access location is added when this term is already present, it cannot be stored (because of the agreed limitation of memory of contents of locations). In this case, a partial object is formed consisting of the unacceptable term. If the partial object already exists because of a previous unacceptable term or a higher level combination, the new term is added to it.

In principle, a whole sequence of Q-sets could be generated, but apart from practical difficulties, the optimum partitioning of the terms into easy calculable sets poses a theoretical problem of great complexity.

There are several further facilities necessary to complete the scheme. Tasks must be provided which up-date the records of the locations following each set of orders which are written. The status, object - non-object, of the locations must also be adjusted.

When a named location is altered by any orders written, the description of the contents of other locations also may change.

If a named fast access location is altered, then all locations with known content referring to the named location either in direct form or in the name of a row of a slow access location will either change or become unknown according to whether it is possible to specify the new value in terms of the old.

Thus if the statement S + T + n is obeyed, the expression S in any memory must be changed to S − T − n. This may generate an unacceptable form and the contents then are marked as unknown. The statement S → T + n destroys the old value and all locations containing this term in the memory must be marked unknown.

Turning to changes to slow access stores, if the row is named explicitly, we cannot be sure that a slow access name described in indirect form is not an alias for the name and all locations with components of this kind must be marked unknown. So must any location containing the exact name. Conversely if the row is described indirectly, all directly named slow access elements become unknown.

It may be thought that this is an expensive expansion to the original simple compiler, but in practice, the only extra apparatus needed are the Q-cells, a value calculating program, and a series of memory adjustment programs added to each order writing program. The location selection programs are also needed in an elementary compiler, although they need not then be so complex.

The final justification for this complication is the added efficiency of the compiler program compared to the extra compilation time and no experience is yet available to make this judgement.

ACKNOWLEDGEMENT

I should like to thank the United Steel Companies for permission to publish this work.

REFERENCES

1. M. V. Wilkes and C. Strachey. "Some proposals for improving the efficiency of ALGOL 60", Communications of the A.C.M., Vol. 4/No. 11/Nov. 1961. pp. 488-491.

2. R. A. Brooker and D. Morris. "An Assembly Program for a Phrase Structure Language", The Computer Journal, Vol. 3/No. 3, Oct. 1960, pp. 168-175.
3. D. G. Owen and K. D. Tocher. "The Automatic Programming of Simulations", Proc. Second International Conference on Operational Research, (Aix-en-Provence), 1960.
4. K. Samelson and F. L. Bauer. "Sequential Formula Translation", Communications of the A.C.M., Vol. 3/No. 2 Feb. 1960, pp. 76-83.

NON-DYNAMIC ASPECTS OF RECURSIVE PROGRAMMING

U. Picciafuoco and M. Pacelli

Olivetti, Milano, Italy

Stack processing procedures, whether executed by special-purpose hardware or by a suitable program that simulates the operation of such hardware, are already well known. The use of a stack makes possible the automatic addressing of the memory cell corresponding to the unfixed, moving end of the stack without knowing the address of this cell. Thus it is possible to program operations involving quantities the locations of which are not known explicitly, provided only that they appear in some appropriate sequence below the quantity at the head of the stack. We shall call the position at the unfixed end of the stack the *privileged* position. Except as they are at some time located in this position, quantities stored in the stack are not addressable. This restriction is not, however, a practical limitation on the usefulness of stacks. On the contrary, its systematic exploitation makes possible the proper interpretation of expressions written in many currently used programming languages, in which parentheses are used to specify logical structure, either during their translation or during the subsequent execution of a program.

But this single characteristic of stacks is no longer sufficient to ensure the unambiguous interpretation of expressions in which function designators occur, since these in turn involve the use of procedures. Ordinarily, in fact, the use of a procedure entails the use of a working-storage area in memory in which are placed its formal parameters which are to be called by value or its local value.

These working-storage areas must be set up in appropriate position in the stack. Now, however, it will be necessary to introduce an addressing system for the stack in order to make it possible to have access to any of these temporary storage areas when their contents are required for the execution of a procedure. The absolute position of such a storage area will remain *a priori* undetermined.

The addressing system will make use of a non-constant ref-

erence point, which coincides with the location of the stack's privileged position before the execution of the procedure call. If the program is expressed in a language, like ALGOL, having a block structure, the addressing system described here for procedures must be extended to every program block in relation to the local variables that each of them defines locally.

Working-storage-area assignments for every block are made dynamically and the absolute addresses of these areas are determined at the time of storage variable definition. A programming system based upon such an extension of stack methods has recently been described by Dijkstra, who has given it the name "Recursive programming", because it appears to be particularly well adapted to handling recursive processing.

To use conventional static storage-assignment procedures it is necessary to make use of a particular area organized according to the push-down list principle in which it is possible to store temporarily the variable values associated with recursive procedures. Dynamic assignment of working storage areas using a stack makes the push-down list useless and enables one to treat recursive 'reincarnations' in the way simple procedure calls are treated.

Both theoretically and practically, recursive processing is of great importance; and since stack systems are also very useful, recursive programming is likely to become more and more important.

In such a scheme the stack areas occupied by local procedures or block variables are not used as operative areas but for declarations only. During the execution of a nest-structured program the stack is alternately occupied by operative and declarative zones.

It is possible to construct various systems which make possible the addressing of working storage areas situated in declarative zones related to nested program blocks.

The effectively used operative zone is the one corresponding to the running program block. A particular program block can refer only to working area defined in its operative zone. A block cannot address intermediate result belonging to operative zones related to other blocks. This system makes it difficult to transfer information from a procedure to another one at a higher level. If we wish to handle procedures in a way equivalent to the method suggested in the Algol 60 Report, it is essential that we be able to transfer to the called procedure a list of addresses.

Every address on this list is related to a general expression, which realizes a particular formal procedure parameter.

In order to ensure, in a simple and coherent way, the possibility of this data transfer, it is necessary that stack operative zones not be defined dynamically; that is, not depend on the program's block structure. For this reason it is necessary that the two stack functions be separated by the introduction to the system of two different stacks, which we will call the declarative stack and operative stack.

Since the operative processes have been removed from the declarative stack, it need not have the dynamic characteristics which the declarative stack must be provided with during a nest-structured program execution.

In particular, data transfer between two procedures is remarkably simple. This is because the operative stack, owing to its static location in memory, can be referred to by any procedures interested in the transfer.

The dynamic mechanism typical of the declarative stack is not equally well adapted to the handling of OWN variables. This mechanism in fact destroys systematically the data contained in working storage areas related to variables defined by already executed program blocks. Before every new execution of the same block, new working storage areas will be defined for local variables. These new areas, however, need to be those assigned to the same variables during previous runs.

Since the previous contents of the working storage area have no significance at the beginning of a new execution, this mechanism is acceptable and it makes possible the best solution of the data allocation problem.

On the contrary, however, at the beginning of every new block execution, defining OWN variables, the values of these variables must be identical to the ones they had at the time of the previous block's execution.

We must preserve the contents of OWN storage fields during the time interval that elapses between two successive block executions.

Naturally this cannot be done if the OWN field is defined on the declarative stack, as for not OWN variables.

This OWN field content has to be moved on the declarative stack according to a position that is not known *a priori.*

It follows that defining the OWN variables on the declarative stack, would destroy its dynamic character.

In order to avoid doing so, it is necessary to set up non-dynamic storage area to be used for OWN variables definition.

If the program block, defining the OWN variables, is involved in recursive processing, the related areas' contents must be preserved in a special push-down list.

We have observed that the declarative stack is equipped with the same list mechanism.

For appropriate handling of OWN variables:

(1) to every OWN variables we must relate associated storage areas in a static OWN region. These will be defined for the entire program execution.

(2) the OWN variables must also be defined on the declarative stack, like the non-OWN ones.

This condition enables us to eliminate the need to provide a push-down list reserved for OWN variables during recursive processing.

For every entry operation in a block defining OWN variables, we must transfer these areas' content related to OWN variables from the static zone to the declarative stack.

The opposite transfer will be executed at every exit from the same block.

The area reserved to static OWN variables cells is divided into two new parts:

(a) A static, or first, OWN zone.

(b) A dynamic, or second, OWN zone.

This distinction is made necessary by the possible presence of OWN arrays having variable dimensions.

In fact it is not the same as for simple variables and for fixed dimension OWN arrays, which have related unmovable working storage areas in the OWN zone. While the contents of a variable dimension OWN array must be also preserved, as it is necessary, for all OWN declared quantities, their positions in the OWN zone is never the same.

Their positions are actually related to:

(1) The values of the variables which define their actual dimensions.

(2) The execution sequence of program blocks having declared variable dimension OWN arrays.

These two factors may differ for every new program execution.

It follows that the position of this class of OWN arrays cannot always be the same during the execution of the program, if efficient space utilization is desired. In consequence, for this class

of arrays the addressing must be done in a way not related to the particular position actually occupied by every array in the OWN second zone.

To obtain these results we must use:

(1) Special storage cells allocated in the OWN first zone. These cells will be referred to as *flag* storages and every one of them will be associated with a particular variable dimension OWN array.

(2) Reference cells containing the addresses of the flag storages.
Every reference cell immediately follows the last element of the array to which it is related and contains the corresponding flag-storage address.

Accordingly, since all of the flag-storages are defined in the OWN first zone, the addresses contained in the reference cells will always be the same.

Summarizing what has been previously said, the OWN first zone will be used for storing the following pieces of information.

(1) Values of OWN simple variables. Every stored value will always be defined, since the first execution end of the program block declaring the particular OWN variable considered.

(2) Sets of OWN constant-dimension array elements.

(3) Flag-storages related to OWN arrays defined in the OWN second zone.

In the OWN first zone, quantities calculated or simply defined in the same program block have associated consecutive storage locations.

The second or dynamic OWN zone is used for variable dimension array elements only.

The rules that control the movement in OWN second zone are those we have described for the generalized stack mechanism, and we shall call the OWN dynamic zone and its moving mechanism an OWN stack. The set of the same array elements will be thus called an OWN stack row. We can also define an OWN stack indicator as a register which will indicate the address of the first free OWN stack position.

To address that part of the OWN first zone where its OWN quantities are defined, each program block must have at its disposal information about the address of the storage cells set related to it in the OWN first zone. In the same way the address of every flag storage in this set must be known, like the values

of variables which allow the actual array dimension computation. These last values usually are not OWN declared quantities, and, consequently, they must be addressed on the declarative stack.

The data required may be numerous and not all of them will be found readily.

We can, however, imagine a specialized computer, in which all the data required for own variables and arrays handling, are recorded together in an automatic addressing list.

The list content will be from the beginning defined for every program block which has own quantities declared.

Not all required information will usually be known before program execution begins. Then the addressing list will require a system which enable the program to make the list content usable.

Beside list up-to-dating, other automatic operations will have to be accomplished before an entry is made in an own quantities declaring program block. They are:

(A) First, that part of own first zone which is related to the block being considered will be copied in a storage area reserved for this purpose is the declarative stack. This operation will assign its correct starting value to every own variables of the block.

(B) Each flag storage found among the copied data will be considered. A "new edition" of the related own variable-dimension array will be created. The array elements number that will be defined in this "new edition" will be chosen accordingly to the actual values of the variables that influence the array dimensions.
Every new edition array will define a new row for the own stack.
The address recorded in the own indicator will be the starting address of a new row. This means that the content of the own indicator will have to be copied in the flag storage cell on the declarative stack associated with the array.

Just after the construction of the "new edition" of the array, some of its elements will have an undetermined starting value. This will occur for those elements that, not being defined in the last edition, are without any starting value. Into the position of the other elements will be copied the content of the related ones of the last edition array, still defined in an old stack row. This data transfer is made possible by means of the address contained

in the old array related flag storage, and that contained in the newly defined declarative stack flag storage, which is related to the new one.

The arrays defined on the OWN stack will not be processed on the declarative stack, as it happens for all other variable quantities.

This will be done in order to avoid the transfer of all the array elements to the declarative stack. By the other hand the declarative stack can not be used by quantities having variable lengths and structure.

The elements of every "new edition" array will be addressed on the OWN stack by means of two references points.

(1) The addressing reference value defined on the declarative stack for the program block that will be executed.

(2) The value contained in the flag storage related on the declarative stack with the desired own array.

This reference address computation will be executed once for every block entry, and the result will be recorded on the above-mentioned list.

If the program block will be involved in a recursive process the sequence of successive "new editions" of the arrays will be created on the OWN stack, in the same time that proceed the dynamic new definition mechanism of the declarative stack.

Every "new editions" array defined during a recursive process will make reference directly to the array edition whose flag storage call is still found in the OWN static zone.

In fact this array edition will not be loose up to the block exit corresponding to the first executed entry.

However the program cannot address its elements because any reference to this array will appear on the declarative stack. The operations related to the OWN static and the OWN stack zones that must be executed for each non-recursive stack exit are, respectively: Global transfer of the contents of the OWN working storage areas and flag storages from the declarative stack to the OWN static zone; and deletion of the old edition of the array from the OWN stack zone.

This last operation will be preceded by a general move of all of the OWN stack rows that follow the array edition that is to be deleted. It will then be deleted, and the contents of the flag storage cells related the moved arrays will be adjusted accordingly. This deletion must of course be accomplished before any trans-

fers to the OWN static zone are executed. This sequencing of these operations is necessary to allow the automatic master control operator to address the deleting OWN row.

The interested flag storage content changes must be really done only after the above mentioned transfer. Using the techniques described here, the authors are preparing a general ALGOL 60 experimental compiler for the Olivetti ELEA 6001 scientific computer.

REFERENCES

1. Dijkstra, E. W., Numerische Mathematik, Vol. 2 (1960), pp. 312-318.
2. P. Z. Ingerman, Communications of the ACM, Vol. 4 (1961) No. 1 - pp. 59-60.

ON STATIC AND DYNAMIC TREATMENT OF TYPES IN ALGOL TRANSLATORS

Klaus Wohlfahrt

Münster, Germany

INTRODUCTION

This lecture deals with the treatment of arithmetic expressions in sequential translators for algorithmic languages similar to ALGOL 60.

The subdivision of types in the ALGOL—report into real and integer appears to offer two methods as principally possible:
(1) Dynamic type treatment: The differentiation of types is effected in the generated programme in this manner: The logical circuitry or programmes for each arithmetic operation test, at run time, the operands for their types before performing any operation in accordance with the types found.
(2) Static type treatment: The translator generates different programme structures for arithmetic operations in accordance with the type of operands as found at compiling time.

In the construction of translators with dynamic type treatment the generating process for arithmetic operations (Ref. 1.) presents no specific difficulties: Arithmetic operations are, in the generated programme, represented without regard to the types of their operands. As soon as in the course of translation the uppermost symbol "+", for instance, has to be deleted from the symbol cellar, the corresponding part of the generated programme will have to transfer both operands, one after the other, to the operand registers of the machine (provided they are not found there already) and close with an order uniquely determined by the sign "+".

On the other hand, static type treatment requests of the translator to act differently according to the different combinations of types in a pair of operands. If g designates a number of the type integer, ξ one of the type real, there are four different combinations:

(a) (g, g), (b1) (g, ξ), (b2) (ξ, g), (c) (ξ, ξ)

and the representation, for instance, of the operation plus in the generated programme in case (a) is essentially different from that in case (c). Moreover, in cases (b1), (b2), according to the ALGOL-report, transfer functions have to be used to reduce these cases to case (c).

To show how this can be done, (b2) is chosen for an example: Here the generating process can be subdivided into the following steps with A and B designating operand registers, A having the property of an accumulator and B being filled only through A.

(1) To the part of the generated programme already completed an order is added to transfer the number ξ_1 of type ξ to A.

(2) Another order is added to transfer the contents of A to B.

(3) By means of a suitable transfer function the translator changes the number g_2 of type g into a number ξ_2 $(=g_2)$ of type ξ.

(4) To the generated programme an order is added to transfer ξ_2 to A.

(5) Another order is added to effect the addition of the real numbers in A and B.

Each of the steps (1) through (4) is a transfer of one operand either from one register to another or from one form into another. Also, (5) can be regarded as the transfer of a pair of operands into a single operand, the result of the arithmetic operation.

Of course, the example just given is purely hypothetical as here all the steps—not only (3)—could be dealt with by the translator. It is useful, however, in showing the kind of steps into which the process can in general be disintegrated.

Static type treatment now will appear as the control of a sequence of operand transfers in accordance with the operation at hand and effectively depending on the types of the operands. The assignment statement may here be included beside the arithmetic operations $+$, $-$, $\times$, and $:$. For shortness of notation the concept of arithmetic complex is introduced: an arithmetic complex (a.c.) is a triple (ω, α, β) with ω standing for one of the operators $+$, $-$, $\times$, $:$ or $:=$ and α, β for the operands belonging to ω.

Either kind of type treatment, dynamic and static, has been used in existing translators for ZUSE—computers

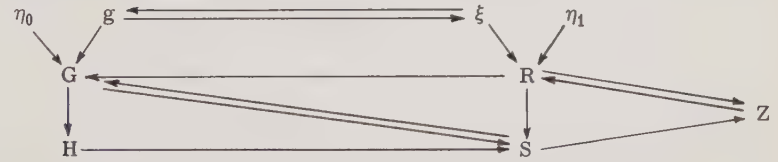
(Z 22, Z 23). An outline of the method used in the static case is given in this note.

1. THE TRANSFER DIAGRAM

In deleting an arithmetic operator from the symbol cellar the translator has at any moment to note the states of the operands as given by the part of the programme generated up to that moment. It is suggested to classify the possible states of each of the operands of a pair as follows:

Symbol	Description
g	integer
ξ	real number
η_0	address of integer variable
η_1	address of real variable
R	real operand in A
G	integer operand in A
S	real operand in both A and B
H	integer operand in both A and B
Z	real operand in one of two interim-registers

The necessary operand transfers are contained in the following transfer diagram:



Both the transfers between g and ξ are effected by the translator, at translating time, by means of transfer functions. All of the others require in general the addition of (normally) one order to the generated programme under construction. Which of the two registers provided in the state of Z is relevant depends only on the position of the operand concerned relative to the operator.

Each a.c. determines a certain operand transfer of the transfer diagram leading to a new a.c., and these transfers are with the aid of a table so controlled as to lead to one of a few permissible final situations. If at least one of the operands originally was of type real, such a final situation, for instance with the operation minus, will have been reached if the first operand is in state S, the second in state R. For then the operation symbol — can, after addition of one more order to the generated programme, be

deleted from the symbol cellar. If as with Z 22 and Z 23 the operation is, at run time, performed with the result in both A and B, the translator eventually has to cancel both the operands from the operand cellar and to add to it the symbol for the state S.

As the transfer diagram applies to either of the operands in an a.c. likewise, it now appears appropriate to provide in the translator for another transfer V, a substitution of each operand by the other, with no immediate consequence for the generated programme. This allows to find the appropriate transfer in any given situation from a transfer table, which in the case of the operation minus may have the following form, similar tables applying to the other possible operators ω in an a.c.

–	η_0	g	G	H	$\eta_1 \mid \xi$	R	S	Z
η_0	G /	G /	Ω_0 / Ω_0	Ω_0 / Ω_0	G / G	V /	V /	G / G
g	G /	G /	Ω_0 / Ω_0	Ω_0 / Ω_0	ξ /	ξ /	ξ / ξ	/
G	V /	V /	/	/	S / S	/	/	S / S
H	V /	V /	/	/	S / S	/	/	/
$\eta_1 \mid \xi$	V /	R /	V /	V /	R /	V /	V / R	R /
R	Z / Z	S /	/	/	S / Z	/	/ Ω_1	S /
S	Z / Z	V /	/	/	V / Z	/	•	V / Z
Z	V / V	/	/	/	/ V	/	/ R	R / V

Here Ω_0 and Ω_1 respectively designate final situations for the subtraction of integers and reals. The first column contains the states of the first, the first row those of the second operand.

The principle underlying the table may be made clear by an example: Suppose an a.c. $(-, \eta_0, g)$ is given. The square in row η_0, column g gives the first transfer as $\eta_0 \rightarrow G$.

The next transfer is found to be V according to the square in row G, column g. This transfer has no consequence for the generated programme while, for the translator, it means that the table has to be transposed, exchanging rows with columns, and that the transfers have now to be taken from under the diagonal in each square. So the third transfer is Ω_0 as stated under the diagonal in the square of row g, column G.

2. THE TRANSLATION PROCESS

A translator implementing static type treatment by the method of control outlined above will have to contain subroutines and tables with the following functions:

(a) Subroutine to test operands for their types.
(b) Subroutine to find the appropriate information from table (c) for any given a.c.
(c) Transfer table with three inputs corresponding to the three parameters of an a.c.
(d) Subroutines for each of the possible transfers of the transfer diagram.
(e) Subroutine for the transfer V with parity check of the number of calls.
(f) Subroutines for each of the final situations.

A certain complication may be mentioned here, namely the possibility of empty operands, as in -1. By a suitable subroutine not included above, care is taken not to treat this like a subtraction zero minus one, and an analogue provision is made for Boolean expressions like E 'less' 0, where the explicit subtraction E minus zero is unnecessary.

Deletion of one of the symbols +, −, ×, : or := from the symbol cellar, at translating time, will then in general run as follows: First, subroutine (a) finds out the parameters of the a.c. to be dealt with, whereupon by means of subroutine (b) the appropriate information is drawn from (c). Next, in general, one of the subroutines of (d) or subroutine (e) is called, with return

regularly to (b). The process ends as soon as table (c) renders one of the final situations of the operations concerned, in which case the appropriate one of the subroutines under (f) is called, whence an unconditional jump leads to the reading subroutine of the translator to continue the sequential translation of an ALGOL-programme.

Finally it may be mentioned that the manner of treatment of arithmetic expressions in the static type treatment allows for a certain freedom, which may be utilized, without altering any of the subroutines, by merely exchanging transfer table (c). From this point of view, dynamic treatment of types can almost be regarded as a special case of the possible ways of static treatment.

REFERENCES

1. Samelson, K., Bauer, F. L.: Sequentielle Formelübersetzung. Elektron. Rechenanlagen 1 (1959), S. 176-182. (Engl. Transl. in: Communications ACM 3 (1960) No. 2, S. 76-83.)

EFFICIENT HANDLING OF SUBSCRIPTED VARIABLES IN ALGOL 60-COMPILERS

U. Hill and H. Langmaack

Institute for Applied Mathematics
University of Mainz, Germany

H. R. Schwarz

Institute for Applied Mathematics
ETH, Zürich, Switzerland

G. Seegmüller

Computation Center
TH, München, Germany

I. GENERAL REMARKS

The following discussion is of special interest for ALGOL 60-users who have problems which lead to algorithms with a great deal of linear algebra and the machines of which have built-in floating point arithmetic. In this case, the running time of generated programs depends considerably on the way in which subscripted variables are treated by the compiler. This is especially true, if the subscripted variables occur in connection with for statements. We shall deal with this important case only. The formula language is ALGOL 60 in the (slightly) restricted representation which is defined by the ALCOR-Conventions. (These conventions are available from The Secretary of the ALCOR-Group, Inst. f. Appl. Math., Mainz University.) We report on the treatment of subscripted variables in some compilers of the ALCOR-Group (for the machines ERMETH (ETH Zürich), PERM (TH München), Siemens 2002, Telefunken TR 4). The basic ideas are outlined in (1).

II. THE STORAGE MAPPING FUNCTION

The mapping function for an array declared by

array $z[a_n : A_n, \ldots, a_2 : A_2, a_1 : A_1]$

is for all compilers of the ALCOR-Group

$$[1] \quad adr(z[y_n, \ldots, y_2, y_1]) = adr(z[a_n, \ldots, a_2, a_1]) - z^*[a_n, \ldots, a_2, a_1] + z^*[y_n, \ldots, y_2, y_1]$$

and it is

$$z^*[\beta_n, \ldots, \beta_2, \beta_1] = (\ldots((\beta_n \times B_{n-1} + \beta_{n-1}) \times B_{n-2} + \beta_{n-2}) \times \ldots \ldots) \times B_1 + \beta_1,$$

$$B_k = A_k - a_k + 1, \quad (k = 1, 2, \ldots, n - 1)$$

We call $z^*[\beta_n, \ldots, \beta_2, \beta_1]$ a "formal index vector" (the usefulness of this notation will be seen in later sections) and use for it also the abbreviation $h(\beta)$. Introducing the vectors a, y and the null vector, we obtain from [1]:

$$[2] \quad adr\,(z[y]) = adr\,(z[a]) - h\,(a) + h\,(y) = adr\,(z[o]) + h\,(y) = C + h\,(y)\,, \qquad (C = \text{const.}).$$

Of course, the following relations must hold

$$a_k \leqq y_k \leqq A_k\,, \qquad (k = 1, 2, \ldots, n).$$

Obviously, our mapping function is a generalization of storing a matrix "rowwise." From [1] it is easily seen

$$[3] \quad h\,(x + y) = h\,(x) + h\,(y)\,, \qquad h\,(\alpha y) = \alpha h(y)\,.$$

The coefficients B_k and C are computed only once by a sequence of instructions corresponding to the array declaraction.

III. DIRECT ADDRESS CALCULATION

If no further information about the expressions on subscript positions is available, the address of the subscripted variable must be computed completely according to [2] upon each entry into the statement or expression concerned. This requires n - 1

multiplications and n additions and certain rounding measures in general. The definition of h(y) shows the advantage of storing rowwise: the subscript positions enter the address calculation in the natural, appropriate order and unnecessary intermediate storing instructions are thus avoided automatically.

Obviously, this "index arithmetic" may far exceed the proper arithmetic of the statement or expression. Occurring in the innermost loops, this fact may cause a considerable increase in the running time of the generated program. The main problem is how to avoid multiplications. In the case in which all subscript expressions are polynomials in the running variable it is pointed out in (1) that the compiler has to generate a sequence of instructions which calculates the complete difference table for the resulting mapping function. The address-calculation then becomes recursive and only additions occur. Fortunately only the linear case is of practical importance.

IV. LINEAR DEPENDENCE ON THE RUNNING VARIABLE

We denote by i the running variable of a for statement. We assume that i is changed with each pass through the loop by a constant s ("loop step"). Furthermore we assume subscript positions

$$[4] \quad y_k = d_k + i \times c_k , \qquad (k = 1, 2, \ldots, n)$$

where c_k, d_k are constant while the for statement is executed. From [2] and [3] it follows

$$[5] \quad \text{adr}\,(z[y]) = C + h(d) + i \times h(c) .$$

Therefore adr (z[y]) is changed with each pass by the constant ("increment")

$$[6] \quad s \times h(c)$$

while the for statement is executed.

V. RECURSIVE ADDRESS CALCULATION

According to [5] and [6] the "initial address" and the "address increment" may be pre-calculated before entering the loop. Within the loop only one addition is necessary for each pass through the loop. This is independent from the dimension n of the array. In comparison with section III we see the reduction

of the computational amount within the loop at least with more than one-dimensional arrays. Besides, making identity checks (e.g., with respect to equal initial addresses, equal address increments) may save storage space. Other problems, such as tests of the possibility of making efficient use of index registers which are available, depend very much on the special features of the machine used.

VI. CONSEQUENCES FOR THE COMPILER

Trying recursive address calculation the question arises to what extent this should be done: only innermost loops, all loops or a compromise that lies between these extremes? It should not be overlooked that in general more instructions are necessary for a loop with recursive address calculation and therefore the ratio of the capacity of working storage to the operation speed plays an important role in such decisions. Also because of this fact the ALCOR-programmers have some influence on the way in which subscripted variables are treated by the compilers: if a certain for variable is declared *real* only direct address calculation is made in those parts of the loop which do not belong to another for statement lying within the for statement concerned. The compilers of the ALCOR-group differ with respect to the extent to which they can generate recursive address calculation. This comes among other reasons from the decision to have a one- or a multiple-pass compiler. On the other hand the question of the amount of the recursive address calculation wanted together with the number of index registers available has a certain influence on this decision.

VII. HANDLING OF SUBSCRIPTED VARIABLES IN MULTIPLE-PASS-COMPILERS

In the first pass a search is made for "admissible for statements" and additionally for "admissible innermost for statements." For admissible for statements the following conditions hold:

a) The for variable is an *integer* declared simple variable which does not occur on the left sides of assignment statements within the for statement.

b) The for clause consists of at least one *step until*-element and contains no *while*-element.

c) All steps are positive or negative integers.

d) All procedures (including functions) within the for statement must be standard procedures.

An "admissible innermost for statement" is an admissible for statement which contains no other for statement.

During the second pass the compiler searches for so-called "admissible index vectors" within admissible for statements. Admissible index vectors are (for brevity we use two-dimensional quantities only)

a) subscripted variables $z[y_2, y_1]$, also called "index vectors,"

b) formal index vectors $z^*[y_2, y_1]$, (definition in section II),

if they meet the following conditions:

a) The array z is not declared within the for statement.

b) The (formal) index vector allows a representation

$$[y'_2, y'_1] + i \times [y''_2, y''_1] ,$$

where i is the for variable of the for statement concerned. The quantities $y_j^{(\nu)}$ are expressions consisting exclusively of the characters

$$+ \; - \; \times \; (\;)$$

integers and for variables of outer admissible for statements (cf. section IV):

Admissible index vectors are treated in the following way:

a) A subscripted variable $z[y_2, y_1]$ is replaced by the operator content $(\bar{z})$, (Barred quantities are (address) variables introduced by the compiler. They differ from all other variables of the program.) Content $(\bar{z})$ means the value of the corresponding subscripted variable. The index vector $z[y_2, y_1]$ serves as a basis for the introduction of the following new statements which are transported into a subroutine (which is also introduced by the compiler) called "precalculation" (lying before the for statement or behind it depending on the direction in which the compiler works through the program: backwards or forward):

$$\overline{ze} := z^*[y'', y''] ; \qquad (s = \text{loop step})$$

$$[7] \quad \overline{zd} := s \times \overline{ze} ; \qquad (i = \text{running variable})$$

$$\bar{z} := \text{adr}\,(z[0, 0]) + z^*[y'_2, y'_1] + i \times \overline{ze} ;$$

In the same way the new statement

$$\bar{z} := \bar{z} + \overline{zd} ;$$

is transported into the rubroutine "addition." To each admissible for statement belong these two subroutines. Of course they may be empty. The contents of the subroutines is treated in the further search for admissible index vectors as belonging to the next outer for statement (if any).

b) An admissible formal index vector $z^*[y_2, y_1]$ is replaced by a new variable $\bar{f}$ and the new statements (cf. a))

$$\overline{fe} := z^{**}[y_2'', y_1''] ;$$

[8] $$\overline{fd} := s \times \overline{fe} ;$$

$$\bar{f} := z^{**}[y_2', y_1'] + i \times \overline{fe} ;$$

are transported to subroutine "precalculation" and

$$\bar{f} := \bar{f} + \overline{fd} ;$$

to subroutine "addition."

This case is only possible because of statements of type [7] in admissible for statements which contain at least another admissible for statement. If the introduction of $\bar{f}$ would cause a statement of the type

$$\overline{ze} := \bar{f} ;$$

(cf. first line of [7]), then $\bar{f}$ need not be introduced and $\overline{ze}$ will be used instead of it ($\overline{ze} := \overline{ze}$ is removed of course):

Identity checks within the subroutine save storage space. On the whole this mechanism causes precalculations and additions executed as far outward as possible and so far inward as necessary. Considerable simplifications come from special index vectors as $[0, \pm 1]$ or $[0, 0]$. In admissible innermost for statements index registers are used as far as possible.

Example:

for k := A *step* B *until* C, . . . *do begin*

. . .

for i := D *step* E *until* F, . . . *do begin*

. . .

$z[y_2, y_1]$

. . .

end ;

```
    . . .
end ;
```

Both for statements are assumed to be admissible for statements. The k-loop may not be enclosed by another admissible for statement.

After the second pass the program reads as follows:

```
go to k1 ;                                   (ALGOL-like notation)
procedure precalc k ;
        begin
            . . .
        fe1 := z**[Y''2, Y''1] ; fd1 := sk × fe1 ;
        ze  := z**[Y'2, Y'1] + k × fe1 ;
        fe2 := z**[X''2, X''1] ; fd2 := sk × fe2 ;
        f2  := z**[X'2, X'1] + k × fe2 ;
            . . .
        end ;
procedure add k ;
        begin
        k  := k + sk ;
            . . .
        ze := ze + fd1 ;
        f2 := f2 + fd2 ;
            . . .
        end ;
k 1 : k  := A ;
        sk := B ;
        precalc k ;
test k : if (k - C) × sign (sk) > 0 then go to next for list
            element k ;
        do k ;
        add k ;
```

```
        go to test k ;
next for list element k :
            . . .
        go to end k ;
procedure do k ;
        begin
            . . .
        go to i1 ;
        procedure precalc i ;
                begin
            . . .
        zd := si × ze ;
        z  := adr(z[0, 0]) + f2 + i × ze ;
            . . .
        end ;
procedure add i ;
        begin
        i := i + si ;
            . . .
        z := z + zd ;
            . . .
        end ;
i1 :    i  := D ;
        si := E ;
        precalc i ;
test i: if (i - F) × sign(si) > 0 then go to next for list
          element i ;
        do i ;
        add i ;
        go to test i ;
```

```
next for list element i :
        . . .
    go to end i ;
procedure do i ;
    begin
        . . .
        content (z̄)
        . . .
    end ;
end i :
        . . .
end procedure do k ;
  end k :
```

Of course it is possible to suppress the further treatment of formal index vectors according to [8]. In general the loss in speed will be negligible. Indeed, some of the ALCOR-compilers are working along these lines.

VIII. HANDLING OF SUBSCRIPTED VARIABLES IN ONE-PASS-COMPILERS

In this case a search for innermost loops is only possible by a "look ahead-technique." On the other hand this may be rather cumbersome if the ALGOL-program cannot be read in completely, e.g., for lack of storage. Today there is only one one-pass-compiler in the ALCOR-group which generates recursive address calculation and it makes no search for innermost loops nor does it test on assignments to the running variable within the for statement. This is feasible because of the ALCOR-convention which does not allow such assignments. All other conditions of section 7 with respect to admissible for statements remain valid. Admissible index vectors, however, are in this case only subscripted variables. The number of instructions in this one-pass-compiler for all work with respect to the recursive address calculation is very low: about 10% of the whole compiler.

A one-pass-compiler is sufficient if the machine has no index

registers or many index registers (hundred or so). Moreover it has the advantage of a very short translation time.

Some of the ALCOR-compilers make their tests for admissibility directly with the ALGOL-program, the others test by means of the generated instruction sequences.

IX. COMPARISON OF RUNNING TIMES

The PERM was used for a comparison of 3 programs, all performing a Gauss-Jordan-process (with a search for column-pivots) for solving linear equations. The algorithm is in all programs exactly the same. They differ in the following way:

Program I : Hand programmed library routine.
Program II : ALGOL program with *integer* declared for variables (recursive address calculation).
Program III : ALGOL program with *real* declared for variables (direct address calculation).

The innermost loops are of the form

for i := h + 1 *step* 1 *until* n *do* a[i - 1, k - 1] :=
a[h, k] × p[i] + a[i, k] ;

The running times were (matrices of degree 30)

I : II : III = 1 : 1.5 : 3.7 .

The generated recursive address calculation yields therefore a factor 2.5. (The bad ratio I : II comes from the fact that a special loop-instruction of the PERM cannot be used by the compiler. For longer loops the ratio tends to one.)

REFERENCES

1. K. Samelson and F. L. Bauer: Sequential Formula Translation. Comm. ACM 3 (1960) 76-83.

METHODE D'EDITION D'UN PROGRAMME EN LANGAGE SYMBOLIQUE

*T. A. Dolotta**

Laboratoire de Calcul, Université de Grenoble, France

I. INTRODUCTION

Le but de cette communication est d'exposer certaines idées et méthodes qui sont utiles dans le traitement des programmes en langage symbolique pendant le premier stade de leur compilation. Bien entendu, quelques-unes de ces idées sont déjà utilisées dans des compilateurs existants; ce que nous voulons faire ici, c'est mettre au point et formaliser toutes ces méthodes et obtenir un ensemble de principes qui se suffit à lui-même.

Le problème qui nous concerne est dû, d'un côté, à la flexibilité que l'utilisateur d'un langage symbolique désire y trouver, et, de l'autre côté, au désir de faciliter la compilation du point de vue du constructeur d'un compilateur. Ce conflit, d'ailleurs bien connu, est dû en grande partie, non pas à la syntaxe du calcul proprement dit, mais plutôt aux règles qui definissent la manière dont on écrit les symboles de base, la portée et la construction des identificateurs,.etc. Dans ce qui suit, nous utiliserons ALGOL (1),(2) comme exemple de langage symbolique. Ainsi, nous trouvons qu'ALGOL permet l'utilisation des identificateurs de longueur arbitraire, et l'utilisation d'un seul identificateur pour plusieurs variables dont les portées sont disjointes; en plus, les symboles de base ALGOL n'ont pas tous la même longueur, et la nature d'un identificateur est donnée par une déclaration séparée. Toutes ces qualités sont désirables pour l'utilisateur du langage. D'autre part, le constructeur d'un

---

*En conge d'études temporaire de Bell Telephone Laboratories, Inc., Murray Hill, N.J., USA, et boursier postdoctoral de l'O.T.A.N./National Science Foundation pour l'annee 1961-62, à Grenoble. Adresse actuelle: Computer Center, Princeton University, Princeton, N.J., USA.

compilateur trouve ces mêmes qualités très gênantes. En effet, il préfèrerait avoir tous les éléments syntaxiques (identificateurs et symboles de base) de la même longueur, avec chaque identificateur indiquant sa nature (type) par sa composition, etc.

Ce que nous proposons c'est de transformer un programme écrit en ALGOL (avant la compilation) en un autre langage, syntaxiquement très similaire à ALGOL, mais qui satisfait ces desiderata. Une telle traduction ou édition des textes ALGOL sera, bien sûr, plus ou moins nécessaire et utile, selon les calculatrices et les méthodes employées pour la compilation proprement dite. Son utilité dépendra des vitesses relatives d'entrée-sortie et de calcul, de la grandeur de la mémoire rapide, et des autres qualités de la machine. Notons ici, que le programme édité occupera moins de mémoires que le programme original.

Actuellement, nous sommes en train de construire des programmes d'édition (pour ALGOL) pour Gamma 60 Bull et IBM 7044.

II. METHODE D'EDITION

Nous expliquerons le processus d'édition en donnant la liste des tâches accomplies par le programme avec des commentaires appropriés.

1. Tâches Périphériques

Le programme accepte un programme ALGOL perforé sur des cartes. Les symboles de base alphabétiques sont délimités par des apostrophes (tirets sur la 7090 actuellement). Deux cartes de contrôle sont utilisées pour encadrer le programme. Toutes les cartes sont imprimées et écrites sur un ruban intermédiaire telles quelles. Les chaînes (strings) et les procédures en code machine sont passées sans aucune modification et placées à la même position dans le programme édité. Tous les commentaires sont éliminés (les commentaires dans les listes de paramètres sont remplacés par des virgules) ainsi que les espaces. Toutes ces tâches périphériques sont accomplies au fur et à mesure du traitement, dont la description suit.

2. Symboles de Base

Chaque symbole de base est remplacé par un mot machine. La composition exacte de ce mot est donnée en appendice. Ce mot

précise non seulement de quel symbole de base il s'agit, mais indique aussi l'ordre de précédence de ce symbole, si cet ordre est défini, et distingue chaque symbole de base de tous les autres symboles de base au moyen d'un numéro de symbole de base (SB). Ceci permet, par exemple, de distinguer facilement deux BEGIN différents. Notons ici que, puisqu'un symbole de base n'est identifié qu'une fois au cours de l'édition du texte ALGOL, il devient rentable de reconnaître plusieurs synonymes de chaque symbole de base, si ceci est désirable. Ainsi notre programme reconnaît les symboles de base du rapport ALGOL (1), ainsi que leur traduction française, aussi bien que certaines abréviations. Par exemple, notre programme traite d'une manière identique les trois symboles de base LABEL, ETIQUETTE, ETIQ. La première partie de la Figure 1 montre un programme ALGOL écrit avec des symboles de base français.

Certains symboles de base basculent en outre certains aiguillages du programme d'édition, de manière que les symboles qui les suivent soient traités d'une façon spéciale. Par exemple, les déclarateurs forcent le programme d'édition à traiter les identificateurs qui les suivent d'une manière spéciale.

3. Identificateurs

Les identificateurs sont traités par l'éditeur à l'aide d'un empilage ou stack. Ce dernier est divisé en plusieurs niveaux. Chaque fois que l'on trouve un BEGIN, on ouvre un niveau du stack; les identificateurs qui sont déclarés immediatement à la suite de ce BEGIN sont entrés dans le stack à ce niveau avec les mots machine qui vont les représenter dans le programme édité. L'appendice donne les détails exacts de la composition de ces mots. Leur partie S indique la déclaration correspondante, et les parties N_1 et N_2 les distinguent, à l'aide de numérotations appropriées, des autres identificateurs qui ont la même partie S.

Chaque fois qu'un identificateur est rencontré en dehors d'une déclaration, on le cherche dans le stack, en commençant par le niveau courant. Une fois trouvé dans le stack, l'identificateur est remplacé dans le programme édité par le mot machine qui lui correspond. Chaque fois qu'on rencontre un END, le niveau courant du stack est écrit sur un ruban magnétique (pour permettre l'impression ultérieure de la liste de tous les identificateurs avec leurs mots machine correspondants) et ce niveau est alors effacé.

Ce processus permet la détection de tous les identificateurs qui, soit, ne sont pas déclarés, soit, sont déclarés incorrectement. En plus, dans le programme édité, la question de la portée ne se pose pas, puisque chaque identificateur a une identité unique. Si deux identificateurs differents sont remplacés par le même mot machine, ceci indique que leurs portées sont disjointes, et qu'ils peuvent utiliser la même memoire pendant l'exploitation.

Ainsi, les deux instructions ALGOL suivantes occupent, dans le programme édité, le même nombre de mots machine (9):

1. A := −B + C × D; (10 caractères)
2. INDEX := SIN(BETA + GAMMA); (23 caractères)

Les étiquettes demandent un traitement spécial puisque leur déclaration ne se trouve pas en tête d'un bloc en général. Ceci nécessite un stack auxiliaire d'étiquettes non encore déclarées. Ce stack opère, du point de vue des niveaux, en synchronisme avec le stack principal. Les niveaux courants de ces deux stacks sont réunis chaque fois que l'on trouve un END, et c'est à ce moment que les étiquettes dans le bloc courant sont codées.

4. Nombres Purs

Les nombres purs sont convertis dans une représentation machine appropriée, et sont par la suite inclus dans une table de nombres purs. Les mots machine qui les représentent dans le programme édité donnent l'adresse dans cette table du nombre en question. Les détails de ce traitement sont indiqués dans l'appendice.

5. Détection d'Erreurs

Nous avons déjà indiqué que l'éditeur détecte toutes les erreurs de déclaration. De plus, les symboles de base non valables sont aussi détectés. Il est aussi possible, avec l'aide d'une matrice de précédence et succession, d'examiner chaque paire de symboles et de vérifier que cette suite est valable. Ceci permet de detecter une grande partie des fautes d'écriture, telles que l'omission des point-virgules, des signes de multiplication, etc. Enfin, l'éditeur fait un décomptage de tous les crochets (BEGIN, END, (,), [,], ‘, ’) et verifie qu'il n'en manque pas.

La détection d'une erreur entraîne l'impression d'un indicateur approprié en regard de la ligne du programme original où l'erreur a été trouvée.

Notons enfin deux autres avantages de la codification telle qu'elle est faite par notre éditeur. Le programme édité n'a plus besoin des déclarations des variables simples, et celles-ci peuvent être éliminées au même titre que les commentaires. En plus, l'encombrement en mémoire rapide du programme édité représente seulement 30 à 60% de l'encombrement du programme original. La réduction est plus marquée pour une machine à 24 positions binaires par mot (Gamma 60) que pour une machine à 36 positions (7044 ou 7090).

III. DISCUSSION

Une méthode d'édition ou de codification telle que nous l'avons décrite ci-dessus permet de concilier les exigences souvent contradictoires imposées d'une part par l'utilisateur d'un langage symbolique, et d'autre part, par le constructeur d'un compilateur pour ce langage, et par la machine utilisée. Elle nous permet de garder une grande flexibilité dans le langage symbolique et, en même temps, de présenter au compilateur des programmes dans une langue très compacte, non-redondante, et complètement normalisée. La plupart des erreurs les plus courantes sont détectées avant la compilation, et le compilateur n'a pas besoin de s'occuper des questions de portée, des déclarations de variables simples, des sauts en avant, etc. Tout identificateur indique, par sa composition, son type et sa spécification.

Nous avons implicitement indiqué qu'un passage préliminaire doit être réservé au processus d'édition. Ceci est préférable lorsque la compilation se fait en plusieurs balayages. Néanmoins, il est possible de faire l'édition et la compilation avec un seul balayage. Dans ce cas, l'éditeur lit le programme original, délimite et code une unité syntaxique à la fois, et la transmet au compilateur. Celui-ci, à son tour, demande l'unité syntaxique suivante lorsqu'il en a besoin, et l'éditeur la fournit à ce moment-là. Ce genre de processus est utilisé déjà dans certains compilateurs existants, mais la distinction n'y est pas faite d'une manière très nette entre l'édition et la compilation proprement dite. La nécessité d'une édition (soit préliminaire, soit parallèle à la compilation) dépend très étroitement des caractéristiques de la machine utilisée, telles que les vitesses d'entrée-sortie, la dimension de la mémoire, le jeu d'instructions machine, etc. Mais en tous cas, une distinction conceptuelle très nette entre les fonctions d'édition et celles de la compilation proprement

dite est, à notre avis, toujours très utile et aide beaucoup durant la conception initiale d'un compilateur.

Bien entendu, notre méthode n'est pas restreinte à ALGOL; en effet, les idées exposées ci-dessus ne dépendent pas du langage source et sont applicables à une gamme étendue de machines digitales.

L'auteur tient à remercier tous les membres du groupe ALGOL du Laboratoire de Calcul de l'Université de Grenoble qui ont aidé au travail dont les résultats sont décrits ci-dessus, et en particulier Messieurs L. BOLLIET, M. BERTHAUD, Y. SIRET et J.-C. BOUSSARD.

BIBLIOGRAPHIE

1. Naur, P. (Ed.), et al. Report on the algorithmic language ALGOL 60, Comm. ACM 3 (1960), 299-314, ou Numer. Math. 2 (1960), 106-136.
2. Genuys, M. F., et al. Raport sur le language algorithmique ALGOL 60, [Traduction française de (1)], Chiffres 3 (1960) 1-44.
3. FORTRAN Assembly Program (FAP) for the IBM 709/7090, IBM publication J 28-6098, White Plains, N.Y. (1960), 13-15.

APPENDICE

Nous donnons ici les détails précis de la codification pour une machine dont le mot comporte 36 positions binaires (IBM 7044) et aussi pour une machine à 24 positions binaires (Bull Gamma 60).

Dans les 2 cas, le mot machine est divisé en 3 parties, nommées S, N_1, et N_2. Si les positions binaires sont numérotées (à partir de 0) de gauche à droite, alors le découpage du mot binaire se fait comme suit:

	Positions binaires	
	7044	Gamma 60
S	0 et 3 à 14	0 à 8
N_1	18 à 26	9 à 17
N_2	27 à 35	18 à 23

Les positions 1, 2, 15, 16 et 17 ne sont pas utilisées dans le 7044.

La partie S sert à spécifier de quel genre d'unité syntaxique il s'agit (par exemple, une variable simple du type REAL, ou le symbole de base +), alors que les parties N_1 et N_2 servent à la

distinguer, au moyen de numérotations appropriées, de toutes les autres unités syntaxiques qui comportent la même partie S.

Si la position 0 du mot machine contient un 1, il s'agit d'un symbole de base. Autrement, le mot représente un identificateur, un paramètre formel, ou un nombre pur. La Table I donne le contenu du reste de la partie S pour tous les symboles de base. Certaines régularités sont apparentes dans ces codes. Par exemple, pour le 7044, la position 3 indique s'il s'agit d'un opérateur ou non, les positions 4, 5 et 6 font la distinction entre les opérateurs de relation, les opérateurs logiques, etc., d'une part, et entre les déclarateurs, les spécificateurs, les crochets, etc., d'autre part. Les positions 7, 8, 9 et 10 sont soit égales à 0 pour les symboles de base pour lesquels un ordre de précédence n'est pas défini, soit indiquent cet ordre s'il existe. Les codes pour le Gamma 60 donnent des renseignements similaires, mais d'une façon un peu moins explicite, car l'information est comprimée dans 8 positions au lieu de 12.

Tous les symboles de base d'un programme complet sont numérotés globalement et d'une façon séquentielle. *Ce numéro de symbole de base (SB)* se place (cadré à droite) dans les parties N_1 et N_2 du mot machine correspondant.

Pour tous les identificateurs et tous les nombres purs, la partie S donne la déclaration et, s'il y a lieu, la spécification correspondante. La Table II indique les détails de la composition de la partie S dans ces cas. Rappelons qu'ici la première position de la partie S (c'est à dire position 0) est égale à 0, pour faire la distinction entre les symboles de base et les identificateurs. A titre d'exemple, les positions 3 à 14 de la partie S (pour le 7044) d'un mot qui remplace un identificateur déclaré comme un REAL ARRAY, contiendraient, en octal, 0022, alors qu'un paramètre formel appelé par valeur et spécifié comme BOOLEAN, contiendrait, dans les mêmes positions, le nombre octal 6004.

L'utilisation des parties N_1 et N_2 varie suivant les différents cas qui se présentent:

1. *Les identificateurs de procédure* sont numérotés d'une façon globale et séquentielle pour un programme complet. *Ce numéro de procédure (IP)* est cadré à droite dans la partie N_1, et la partie N_2 est égale à zero.

2. *Les paramètres formels* d'une procédure sont numérotés suivant leur ordre d'apparition dans l'entête de procédure. *Ce numéro de paramètre formel (PF)* est cadré à droite dans la

TABLE I

Spécification des symboles de base

Type	Symbole de base	Spécification octale 7044 (Pos. 3-14)	Spécification octale GAMMA 60 (Pos. 1-8)	Code mnémonique 7044
op. logique	≡	0420	220	OL 1 0
	⊃	0440	222	OL 2 0
	∨	0460	224	OL 3 0
	∧	0500	230	OL 4 0
	¬	0520	236	OL 5 0
op. de relation	<	1141	244	OR 6 1
	≤	1142	246	OR 6 2
	=	1146	240	OR 6 6
	≥	1144	243	OR 6 4
	>	1150	241	OR 6 8
	≠	1151	247	OR 6 9
op. arithm.	+	2161	264	OA 7 1
	−	2162	266	OA 7 2
	×	2201	270	OA 8 1
	/	2202	272	OA 8 2
	÷	2204	273	OA 8 4
	↑	2220	277	OA 9 0
op. sequentiel	GOTO	0001	211	OS 0 1
	IF	0002	202	OS 0 2
	THEN	0004	204	OS 0 4
	ELSE	0005	205	OS 0 5
	FOR	0010	210	OS 0 8
	DO	0011	216	OS 0 9
valeur logique	TRUE	4001	361	VL 0 1
	FALSE	4000	360	VL 0 0
déclarateur	PROCEDURE	5011	347	DE 0 9
	OWN	5010	344	DE 0 8
	BOOLEAN	5001	343	DE 0 1
	REAL	5004	341	DE 0 4
	INTEGER	5002	342	DE 0 2
	ARRAY	5007	345	DE 0 7
	SWITCH	5006	346	DE 0 6
spécificat.	STRING	5401	351	SP 0 1
	LABEL	5402	352	SP 0 2
	VALUE	5404	353	SP 0 4

TABLE 1 (continued)

Type	Symbole de base	Spécification octale 7044 (Pos. 3-14)	Spécification octale GAMMA 60 (Pos. 1-8)	Code mnémonique 7044
séparateur	,	6006	324	SE 0 6
	.	6010	325	SE 0 8
	10	6011	337	SE 0 9
	:	6005	326	SE 0 5
	;	6007	327	SE 0 7
	:=	6004	320	SE 0 4
	STEP	6001	334	SE 0 1
	UNTIL	6002	335	SE 0 2
	WHILE	6003	336	SE 0 3
	COMMENT	ne se présente pas dans le programme édité		
	espace			
crochet	(	6402	303	CR 0 2
	)	6403	314	CR 0 3
	[	6404	305	CR 0 4
	]	6405	312	CR 0 5
	‘	6406	306	CR 0 6
	’	6407	311	CR 0 7
	BEGIN	6400	300	CR 0 0
	END	6401	317	CR 0 1

partie N_2 du mot machine, et la partie N_1 contient le IP correspondant.

3. *Tous les autres identificateurs* sont numérotés d'une façon globale. Pour permettre l'économie de mémoires pendant l'exploitation, le même numéro peut être donné à plusieurs identificateurs, pourvu que leurs portées soient disjointes. Ces numéros sont cadrés à droite dans les parties N_1 et N_2.

4. *Les nombres purs* peuvent être traités de deux manières distinctes. On peut soit les garder dans le programme édité (à la manière de facteurs effectifs, ou "literals", de FAP (3)) aux endroits où on les trouve dans le programme ALGOL original, mais en les faisant précéder d'un mot repère, soit les ranger, après une conversion appropriée, dans une table spéciale. Dans ce deuxième cas, on place dans les parties N_1 et N_2 du mot machine, qui remplace un nombre pur, *un numéro de nombre pur (NP)*. Ce numéro est l'adresse relative, dans la table des nombres purs, du nombre en question.

La table des nombres purs, si elle existe, contient des nombres déjà convertis en représentation machine appropriée.

TABLE II

Spécification des identificateurs

Genre d'identificateur	7044		GAMMA 60	
	Positions mises à 1	Positions forcées à 0	Positions mises à 1	Positions forcées à 0
FORMAL PARAMETER	3	5,9	1	5
VALUE[1]	4	5,6,7,8,9,14	2	3,4,5,9
DECIMAL NUMBER[2]	5	6,7,8,9,12,13,14	2	1,3,4,5,8,9
LABEL[3]	6	9,10,11,12,13,14	3	4,5,6,7,8,9
SWITCH	7	9,10,11,12,13,14	3,4	5,6,7,8,9
STRING[4]	8	9,10,11,12,13,14	4	3,5,6,7,8,9
OWN[5]	9	14	5	9
REAL	10	11,12	6,7	
INTEGER	11	10,12	6	7
BOOLEAN	12	10,11	7	6
ARRAY	13	14	8	9
PROCEDURE[6]	14	9,13	9	5,8

[1]Si la position réservée à VALUE est égale à 1, la position précédente (FORMAL PARAMETER) doit aussi contenir 1.

[2]Un nombre décimal est soit REAL soit INTEGER. Par la suite, pour le 7044, une seule des positions 10 ou 11 est égale à 1, et pour le GAMMA 60 la position 6 est égale à 1.

[3]Y inclus les étiquettes (labels) purement numériques.

[4]Dans le cas de spécification de STRING, la position de FORMAL PARAMETER doit être égale à 1.

[5]Dans le cas d'un identificateur du type OWN, les positions correspondantes aux REAL, INTEGER, our BOOLEAN ne peuvent toutes être égales à 0.

[6]Notons que pour le GAMMA 60, la position correspondante à PROCEDURE déborde la partie S du mot machine, et occupe la première position de la partie N_1.

Cette table peut être construite de trois manières différentes:

1. Chaque nombre y est rangé au moment où on le trouve dans le programme original.

2. Les nombres sont triés, et on range un nouveau nombre seulement s'il s'agit d'un nombre qui n'est pas encore contenu dans la table.

3. On introduit a priori quelques nombres tels que 0, 1, 2, et on range tous les autres par ordre d'arrivée.

Programme ALGOL (avec cartes de contrôle) avant la codification :

```
ALGOL          PROCEDURE FACTORIELLE
-ENTIER- -PROCEDURE- FACT(N)¤¤   -ENTIER- N ¤¤
    -COMMENTAIRE-  CETTE PROCEDURE CALCULE LA FACTORIELLE DE N ¤¤
  -DEBUT- -ENTIER- A ¤¤
      A ¤= -SI- N=0 -ALORS- 1 -SINON- N*FACT(N-1) ¤¤
            FACT ¤= A  -FIN- ¤¤
FALGOL          FIN DE LA FACTORIELLE
```

N.B. : ¤¤ représente ;
¤ = représente :=

Programme ALGOL après la codification :

Code mnémonique : Programme ALGOL codé :

```
*DE 0 2        0.  1   10C1C1C0000000100C0C0C0C0C0C0C0C0C0C00001          0
*DE 0 9        0.  2   10010100000010010000000000C0000000C00010          1
 IE . .E P     1.  0   00000000000100100000000000010000000000             2
*CR 0 2        0.  3   10011010000C0010000000000000C0000000011          3
 FN . .E .     1.  1   00010000000100000C0C0000000100000000001          4
*CR 0 3        0.  4   10011010000000110C0000000000C0000000100          5
*SE 0 7        0.  5   10011000000011100000000C0C0C0000000101          6
*DE 0 2        0.  6   10010100000001C0C0C0C0C0C0C0C0C0C0C00110          7
 FN . .E .     1.  1   00010000000100000C0C0C0000001000000001          8
*SE 0 7        0.  7   10011000000011100000000000C0000000111          9
*CR 0 0        0.  8   10011010000000000000000C0C0C0000001000          10
*DE 0 2        0.  9   10010100000001000C0C0000000000000001001          11
 IE . .E .     0.  1   00000000000100C0000000C0C0C0C0000000001          12
*SE 0 7        0. 10   10011000000011100000000C0C0C0000001010          13
 IE . .E .     0.  1   00000000000100000C0C0C0C0C0C0C0000C0001          14
*SE 0 4        0. 11   10011000000010C0000C0000000C0000C01011          15
*OS 0 2        0. 12   10000C0000000010000C0000000C0C0000001100          16
 FN . .E .     1.  1   00010000000100000000000000C01000000001          17
*OR 6 6        0. 13   10000100110011000C0000000C0000001101          18
 NP . .E .     0.  1   00000C1000001000C0000000000000000001          19
*OS 0 4        0. 14   10000000000010000000000000C0000001110          20
 NP . .E .     0.  2   00000100000100C0C0C0C0C0C0C0C0000000010          21
*CS 0 5        0. 15   10000000000010100000000000C0000001111          22
 FN . .E .     1.  1   00010000000100000000000000010000000001          23
*CA 8 1        0. 16   10C0100100000010000C000C0C0000000000010000          24
 IE . .E P     1.  0   00C0000000010010C000000C0010000000000          25
*CR 0 2        0. 17   10011010000001000000000000000000010001          26
 FN . .E .     1.  1   00010000000100000000000000001000000001          27
*CA 7 2        0. 18   10001000111001000C0C0000C0C0C0000010010          28
 NP . .E .     0.  3   00000100000100000C0000000000000000011          29
*CR 0 3        0. 19   10011010000001100000000000C0C0C0100111          30
*SE 0 7        0. 20   10011000000011110C0C0000C0C0C0000010100          31
 IE . .E P     1.  0   00000000000100100C0C0C0000C0100000000          32
*SE 0 4        0. 21   10011000000010000C0000000000C0000010101          33
 IE . .E .     0.  1   00000000000100000000000000000000000001          34
*CR 0 1        0. 22   10011010000000100000000000C0000010110          35
*SE 0 7 /      0. 23   10011000000011100000000000000000010111          36
```

Fig. 1. Exemple de codification sur IBM 7090

Le choix entre ces trois méthodes dépend surtout de la machine utilisée.

Nous précisons qu'actuellement, notre méthode ne peut pas encore manipuler correctement des étiquettes purement numériques qui se trouvent dans des listes de paramètres effectifs.

Finalement, notons qu'un code mnémonique a été développé pour aider à la mise au point du compilateur et du programme d'édition. Ce code prend comme données un programme entièrement édité, et le traduit, mot par mot, en une forme plus facile à interpréter visuellement. La Table I donne ce code mnémonique pour les symboles de base (pour le 7044).

La Figure 1 donne le résultat de notre processus d'édition. Les listages de la Figure 1 ont été obtenus à l'aide d'une partie du programme d'édition qui marche sur le IBM 7090 ou sur le IBM 7044. On y voit le programme original en ALGOL, ainsi que le programme édité (sur mots de 36 positions) avec le code mnemonique de chaque mot paraissant dans la colonne gauche. Les symboles de base sont précédés par un astérisque, et la traduction mnémonique montre les trois parties S, N_1 et N_2. Les symboles de base paraissent ici dans leur traduction française.

EFFICIENT COMPILATION OF PROGRAMS WRITTEN IN A MIXED PROGRAMMING LANGUAGE

Sydney P. Levine

School of Industrial Management
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA

A mixed programming language is defined as a programming language with an expandable set of commands, a fixed number of which are the symbolic synonyms of the hardware commands, and the remainder of which, excluding control commands to the compiler itself, are pseudo-commands which are replaced during compilation by sets of actual hardware commands.

Mixed programming languages, as defined above, include all programming languages which may be said to be direct descendants of those early programming languages which provided the programmer with some clerical conveniences, but which demanded that his program be an isomorphic translation of the program as actually executed by the computer.

The definition specifically excludes so-called higher level languages, such as FORTRAN and COBOL, by including the symbolic synonyms of the hardware commands in the programming language. Mixed programming languages have evolved comparatively slowly, while the first higher level language to gain general acceptance, FORTRAN, appeared more or less in its present form—a fait accompli.

A search of the literature reveals two somewhat contradictory facts. First, if one examines the lists of programs available through the ubiquitous "users' groups," one is struck by the number of programs written in the lower level languages. Second, if one examines the technical computing journals, one is struck by the large number of articles devoted to compilation in higher level languages, and the scarcity of articles dealing with similar topics with respect to lower level languages. As a result, evidence can be cited for the loss of interest in lower

level languages, or equally well, for the lack of acceptance of higher level languages.

The truth, which finally seems to be gaining wide acceptance, is that both types of language are useful, and that each is superior to the other in some number of ways.

In an effort to incorporate some of the features of higher level languages into lower level languages, several attempts have been made to expand these languages so that they would include statement types similar to those found to be particularly useful in the higher level languages (1)(2). These statements in the lower level languages normally preserve the format of the machine language synonyms which are the basis of the programming language; that is, each particular statement requires an operation code (or more strictly, a pseudo-operation code), a number of operands, and, if needed, a label as a reference point. Not only is the statement formally identical to a machine language synonym, but (until recently) the processing of the statement by the particular compiler was a logical extension of the way in which machine language synonyms were processed. Let me illustrate. Once the operation code of a machine language synonym is identified by the compiler, the machine language into which the particular statement is translated is completely determined except for the particular "name" values which are derived from the operands of the statement. The operands themselves (except in rare cases) play no active role in this process.

The same mechanism has been used to process higher level statements in basically lower level languages. The only difference is that there is another level of non-machine language coding intervening between the statement as written by the programmer and the machine language as executed by the computer. Once the pseudo-operation code is identified by the compiler, the machine language *synonyms* into which the statement will be translated, and ultimately the machine language itself, is completely determined except for the "name" values derived from the operands of the original statement. Once again, the operands play no active part in the process. (Extension to the case of pseudo-operations which generate other pseudo-operations is immediate.)

In their attempts to expand the power of so-called lower level languages, designers of these languages have in fact allowed only for expansion in one direction; that is, new operation codes may be added to the list of those operations which are indigenous

to the computer. The statement types thus added may be thought of as machine language synonyms which are actually executed by programmed simulation.

The statement types which have been added to lower level languages in the manner described above have been much less powerful and useful than the statement types of the higher level languages which they have attempted to emulate. The reason is not difficult to discover. As mentioned previously, the operands of either machine language synonyms or pseudo-operations in mixed programming languages (with exceptions to be subsequently discussed) play no part in determining which machine language instructions will be generated from the original programmer written statement. This is in direct contradiction to the mode of operation of higher level language compilers. Consider, for example, a statement in the FORTRAN language:

A = B + C.

Consider at the same time, a statement of the form:

I = J + K.

Although the statements are formally identical, the machine language instructions which are generated in order to accomplish the addition will be different in each case, and the difference will be directly attributable to the fact that in the first case all of the variables were identified as floating point numbers, and in the second case, all of the variables were identified as integers.

The fact that during higher level language compilation the operands within a statement play an active part in determining just what will be generated from the original programmer written statement and that during compilation of programs written in lower level languages this is not true, is the principle reason for the power difference between superficially similar statement types in the two language categories. Consider once again the FORTRAN arithmetic statements just cited. In FORTRAN, both of the examples are instances of a single statement type, the arithmetic statement; in lower level language, a different pseudo-operation would be needed to handle each of these instances, a floating add, and an integer add.

It should be said that the designers of higher level languages have restricted their universe of operand types in order to facilitate the active operand type of compilation that we have been discussing. In most higher level languages, one is generally re-

stricted to the manipulation of either integers or floating point numbers. This restriction, while not binding in many problems of interest, is often the main reason for using a lower level language in preference to a higher level language.

The problem faced by the designer of the expanded lower level language (or as I have chosen to call it, the mixed programming language) may be stated as follows. Those statements which have been added to lower level languages in an effort to emulate the more desirable features of the higher level languages have proven to be less powerful than hoped, because, unlike the statements in the higher level languages, the operands do not play an active role in the compilation process. Moreover, the designers of mixed programming languages cannot solve this problem by restricting the types of acceptable operands as has been done in the case of higher level languages, because this would destroy one of the most desirable characteristics of the lower level languages; namely the ability to deal with variables of virtually any form.

The recognition of this problem is not new, and in at least one instance it has been met squarely, and I think solved(3). In essence, the solution is as follows. Due to the nature of lower level languages, each operand, whether it appears in a machine language synonym or in a pseudo-operation, must be defined somewhere in the program. In the case of machine language synonyms this definition can be thought of as occurring always in one place, and often in two. The implicit definition, which always occurs, is in the statement itself. Usually by position, and occasionally by morphemic convention as well, the operand is identified. The second definition, which is sometimes not necessary, is explicit. In this second definition, the operand name appears as the label of another statement which explicitly describes an area of storage that is to be reserved, a constant that is to be stored, or another operation. This second definition is usually not available to the compiler at the time the original statement, with its implied operand definition, is being processed. For a machine language synonym, this is of no consequence. For a pseudo-operation, however, this fact restricts all statements added to the language to be generic extensions of the actual operation codes.

In higher level languages, both of these definitions are combined in the appearance of the operand in the original statement. This is possible because of the naming restrictions imposed by the language. Consider once more the FORTRAN arithmetic

statements cited earlier. In both instances, the operands would have been identified as addend, augend, and sum, by their positions within the statement, but the operands themselves carry information, because of the naming conventions, as to their specific nature, fixed or floating point.

Such a statement in a lower level language would carry the addend, augend, sum information with it, but the specific information as to the nature of the factors would be found only in the explicit definitions of the operands, which would be found in other statements in other sections of the program.

If the information contained in the second, explicit, definition could be made available to the compiler at the time the statement calling for the particular operand was being processed, then statements comparable in power to those of higher level languages could be included in the lower level languages.

The solution to this problem is quite ingenious. As each statement of the original program is read by the compiler, a separate record is made for each label and for each operand. In the record made for each label, the characteristics of the piece of information which is named by the label are noted; that is, whether or not the labelled statement is a machine language synonym, or a pseudo-operation, or a fixed point number of so many digits, and so forth. After the entire program has been processed in this manner, the resultant file of records can be sorted so that the record for a particular label will be followed by all of the operand records that reference that label. In this disjointed form, the entire program can be read and characteristics transferred from the label records into the operand records.

After the program is sorted back into its original order, each operand carries with it the specific characteristics of the piece of information which it references.

After the pseudo-operations have been expanded to include the characteristics of the operands, the particular machine language coding which ultimately results from the inclusion of the statement will be determined by a generator program which will have available to it a complete and explicit description of the factors involved in the computation. This is the characteristic which has in the past distinguished higher level compilation from lower level compilation. In addition, because there are no naming conventions necessary for the purposes of data description, the types

of data which a particular statement type will accept is limited only by the ingenuity of the generator writer.

It would appear then that the designer of the mixed programming language has at hand the ability to produce a programming language which rivals the higher level languages in the power of its individual statements, while combining the flexibility inherent in lower level languages. This is not without its cost, however, and the cost, as would be expected, is in increased compilation time.

While to some extent this is to be expected, I believe that the condition is being aggravated because the designers of mixed programming language compilers have failed to take advantage of some of the unique properties of programs written in mixed programming languages. As things stand now, the compilation process in both mixed programming languages and higher level languages can be described, grossly, in the same way. The compilation may be divided into two sections. In the first section, each statement written by the programmer is expanded into its machine language synonym equivalents. In the second section of the compiler, the program, which is now completely in terms of machine language synonyms, is assembled by some version of the venerable two-pass assembly program.

In principle, this sequence of operations is logical, and, in the case of higher level languages, quite necessary. However, in the case of mixed programming languages, I believe that the two sections can be, to a great extent combined. The reason for this is rooted in the fact that, unlike a program written in a higher level language, a program written in a mixed programming language has many statements (the machine language synonyms and explicit definitional statements) whose exact nature with regard to the space they will occupy in the executed program is known as soon as the particular statement is recognized by the compiler. These locally exact segments of program can be partially assembled as soon as they are encountered. Consider for example, a sequence of machine language synonyms that is bounded before and after by pseudo-operations. These statements are read in sequence by the compiler. Although we do not know the location which will be occupied in the executable program by the first instruction generated from the first pseudo-operation, we can name it, say, A. The ultimate location of the next statement, the first of a sequence of machine language synonyms, is also unknown, but we can name it B. The next statement, also a machine

synonym, is similarly unknown, but we do know its location *relative to the statement that preceded it*, and so we can name it B+1. All of the following machine language synonyms can be similarly given B numbers. The final pseudo-operation can also be given a B number because we know its first location relative to the other B named statements. However, since the number of statements generated from this second pseudo-operation is unknown, the statement following this second pseudo-operation will have to be given a new name, presumably a C number. In this way, the entire program can be relabelled in terms of these A, B, C, . . . numbers with their associated incremental suffixes. The number of these internally generated names is determined solely by the number of times during the course of scanning the program that it is impossible to associate the actual location of a statement, relatively, with the one that immediately preceded it.

Obviously, this renaming of the labels in the original program is useless unless this information can be made available to the operand records. From the preceding discussion, it is obvious that we have a mechanism for doing just that. At the same time that data characteristics are being transferred from label records to operand records, information about the renaming of the symbols used in the program can also be transferred. After this has been done, and the program has been sorted into original order, the program can be thought of as having been completely rewritten in terms of the new, internally assigned symbols. This revision of the original program has several advantages. First, a partial assembly has taken place which has eliminated all labels except those associated with statements whose length in the executable program is indeterminate at the time that the internally generated symbols were assigned. Second, because these labels are generated during the course of the compilation and are not used by the programmer who wrote the program being compiled (indeed he need never know that they exist), they can be formed in a manner most useful for computer manipulation without any regard for whether or not they are convenient for human interpretation. This is important because one of the limiting factors in the assembly phase of most compilers is the number of symbols (usually meaning programmer written symbols) that can be accommodated in high speed storage during actual assembly, and symbols that are most convenient for humans are often most awkward for computer manipulation(4).

The results of this partial assembly and symbol compression are seen in the two pass assembly that is the last stage of the compilation. The two passes may be described quite simply. In the first pass, actual locations are assigned to symbolic labels, and in the second pass, this actual information is passed on to the operands. It is this second pass of the two pass assembly that is the most time consuming, and where the restriction on the number of symbols that can be accommodated in storage becomes binding.

The assembly process described above is actually done in one of two ways; either with an explicit symbol table, or with an implicit symbol table. In the explicit symbol table techniques, the symbols and the actual locations to which they have been assigned are collected during the first pass, and a table is formed. During the second pass, as each operand is recognized, it is searched for in the table, and when found, the actual information is transferred to the operand. The implicit symbol table technique is the same device that we used previously to transfer data characteristics from label records to operand records.

The explicit symbol table techniques have the drawback that they are limited by the number of symbols that can be accommodated in high speed storage, while the implicit symbol table technique, while not limited in that manner, requires two complete sorts of the entire program.

The method that I would propose is an explicit symbol table technique that I believe avoids the limitations of traditional devices of that type.

During the first pass of the two pass assembly, the internally generated symbols and the actuals assigned to them are collected. Because of the partial assembly done earlier, there will probably be fewer of these symbols than of the original programmer written symbols. Because these symbols were assigned sequentially, when they are collected they will be in ascending order and need not be sorted for a subsequent search.

It should be remembered that when we assigned these internal symbols, we were free to use any convenient form, as long as the symbols could be assigned in an ascending sequence. Since that is the case, it is perfectly legal to use as our symbols the addresses of the locations of our symbol table area. By using this device, it is not necessary to store in the table both the symbol and its actual assignment. Let me illustrate. Suppose we know that we have available in the compiler, words 1240 to 1999

for use as a symbol table. Then we assign our internal symbols to be the numbers 1240, 1241, and so forth. Then, suppose that during the first pass of the assembly program we assign symbol 1240 to be actual location 784, and symbol 1241 to be actual location 802. We store the number 784 in location 1240, and the number 802 in location 1241. Then, during the second pass of the assembly, when the symbol 1240 is recognized as an operand, it is not necessary to search the symbol table in order to locate the actual that has been assigned to this address, because we know that the number stored in location 1240, the number 784, is that actual. And similarly for the symbol 1241.

By using this device, namely the use of the addresses of the actual locations of the symbol table as internal symbols, we have eliminated the need for any searching during the second phase of the assembly, and we have increased the capacity of our symbol table by eliminating the need for storing the original symbol and its actual assignment in the symbol table.

The coupling of the two techniques, that is, the partial assembly and the symbol table compression, should speed up compilation considerably and make the process relatively insensitive to the amount of core storage available to the compiler (beyond a not inconsiderable lower limit).

The insensitivity of the compilation to the amount of core storage that is available brings up an economic question. The principle difference between so-called large scale computers and medium size computers is the amount of core storage available with each. In most cases the input-output equipment, specifically the tape units, is identical.

The techniques that I have described above reduce the problem of compilation in a mixed programming language, in large part, to one of sorting and sequential data processing. Would it not be economically desirable to compile programs that are to be run on large scale computers on medium size computers that are very nearly as efficient data processors as are the large scale machines?

For example, consider the following information:

	IBM 7090	IBM 7074
Average cost per month(5)	$64,000	$29,300
Add time	4.4 micr.	10 micr.
Access time	2.2 micr.	4 micr.

I have chosen these two computers because their comparative

characteristics were easily obtainable. I am certain that similar comparisons could be made between any two computers of similar relative sizes.

With the disparity in the two monthly costs, and the fact that, computed from the manufacturer's literature(6)(7), the sorting times on the two machines are almost identical, does it not seem highly probable that compilation on the IBM 7074 for the IBM 7090 would be highly profitable?

REFERENCES

1. Greenwald, I. D. "A Technique for Handling Macro Instructions", Communications of the ACM, Vol. 2, No. 11, November 1959.
2. McIlroy, M. D. "Macro Instruction Extension of Compiler Languages", Communications of the ACM, Vol. 3, No. 4, April 1960.
3. IBM 7070 Autocoder, IBM 7070 Data Processing Systems Bulletin, Form J28-6032.
4. Williams, F. A. "Handling Identifiers as Internal Symbols in Language Processors" Communications of the ACM, Vol. 2, No. 6, June 1959.
5. Statland, N. "Computer Characteristics Revisited", Datamation, Vol. 7, No. 11, November 1961.
6. IBM 7090 Generalized Sorting Program, Form J28-6043-1.
7. IBM 7070/7074 Generalized Sorting Program, Form C28-6111.

PANEL DISCUSSION
Philosophies for Efficient Processor Construction

Moderator :	W. L. van der Poel	(Netherlands)
Members :	A. Caracciolo	(Italy)
	E. W. Dijkstra	(Netherlands)
	J. M. Foster	(U. K.)
	P. Naur	(Denmark)
	J. Poyen	(France)
	K. Samelson	(Germany)
	C. Strachey	(U.K.)
	J. Weizenbaum	(U.S.)
	M. V. Wilkes	(U.K.)

VAN DER POEL (Netherlands)—I first wish to make a few remarks and visualize what will be discussed. I think we have to restrict ourselves just to a few points; namely, in the first place: efficient translators. That is, how to make translators which are doing the translation job efficiently. Secondly: how do we make translators which make object programs which are working as efficiently as possible. And I think we must mainly restrict the discussion to these two points.

Well, this is going to be a hot thing, of course. I am called Moderator here! I hope you people will serve as fuel, and now I have to start the chain reaction. I know that several of the members have something to say.

NAUR (Denmark)—I am a little bit surprised at what my Chairman said, that I have to stick to the subject of speed of object program and speed of translation. Because I see in discussing philosophies for efficient processor construction, there are many more interesting aspects. Obviously, besides the aspects of the two speeds, there are the storage requirements of the object program. But there are also, it seems to me, other equally significant possible philosophies, for example the speed of writing the processor. This is certainly interesting, and something some other people may not think of, the efficiency by which the processor construction will teach the implementor, for example, on ways of handling the mechanism of the language in a given machine, or the way the processor might teach the implementor, what a good machine has to be like. And, quite generally, there are processors such that the whole work might inspire the people who are involved with good ideas. I just mention this and I would be very unhappy if all these aspects were completely forgotten.

SAMELSON (Germany)—Our topic is "philosophies for efficient compiler construction"; now, to raise a really controversial issue I feel obliged to state here—although I hate doing so—that the final judge in matters of efficiency is money. It may sound a little bit harsh, but what we have to look for is the system which will give the people who work on the machine the most economical means for solving their problems. In this connection we can forget about the research institutes where compilers are being developed and where economy is not a very important consideration. What really counts is the large group of commercial users of machines, and the manufacturers. The compilers are essentially distributed by the manufacturers. Here I think that economic

considerations will have to guide us completely and I feel that all the speed problems that have just been mentioned will have to be considered from the economical point of view.

VAN DER POEL (Netherlands)—I do not think that we have to bother too much about which subject goes in exactly. I said "mainly" of course, and Mr. Naur is completely right when he says the efficiency is also dependent on the speed with which you can teach. This is granted, of course.

DIJKSTRAI (Netherlands)—You said that discussions would center mainly on efficiency of the translators and efficiency of the running system. I fear that you turn out to be right and that the main point of this Conference will give attention to these problems and that this will be done in an atmosphere where everybody apparently agrees that these problems are the major problems. I am confirmed in this by the statement made by Samelson. He said that money is the only scale of measurement. O.K. But I feel that the extreme importance given to considerations of space and time is mainly due to the fact that these are the most trivial aspects of efficiency, because they can be measured with a quantity scale. When you said that what is important is the most economic way of solving problems, and you put a stress on economic, I should like to put a stress on "solving" the problems and I should like to ask your attention—with my very weak voice of course—as to how we know that we have solved the problem. It may well turn out that you find that by introducing I don't know how many kinds of techniques for improving efficiency you end up by having a very economic way of doing something very unreliable. My greatest objection depends on the fact that improvement of efficiency makes it very hard to convince oneself of the correctness of a translator. I am therefore afraid that one might turn out to find a very economic way of doing nothing reliable.

VAN DER POEL (Netherlands)—May I just make a few remarks to this last remark of Mr. Dijkstra. I think that I do not agree with the difficulty of getting efficient Algol translators. Have I misunderstood you?

DIJKSTRA (Netherlands)—One should not bother himself too much with optimizing. This is just a simple point: optimizing—at least as far as I can understand it—is taking advantage of a special situation. If you do this for one special situation, you

may check by hand, perhaps, or prove, or convince yourself that this is correct. As soon as you have two special situations where you optimize, you have to check that the solution is correct and that the two manners of optimization do not interfere with one another. I am afraid that if you do this with ten ways, then the difficulty of clarifying explodes beyond our limited powers.

VAN DER POEL (Netherlands)—In that respect I think you are right. Perhaps some one can pick up the discussion later on. First I want to give the opportunity to panel members to speak up.

STRACHEY (U.K.)—I would like to try to bring the discussion down to a method discussion and what we mean by it, and to talk a little bit on what we might do in order to get something done.

One of the problems I have been concerned with is the problem of trying to build compilers for machines which have a very different structure from the ordinary sorts of machines for which most of today's languages have been designed. I hope that some time in the future we might be able to get away from heavily machine ordinated languages such as Algol. The problem of such languages is that they insist on giving a great deal more information than is mathematically necessary and this information, far from making it easier for the compiler program, makes it a great deal more difficult, and for some machines makes it actually impossible. Take for example a simple operation like matrix multiplication: if you write an Algol program for it, you have to specify precisely the order in which all the multiplications have to be made. Now, from a mathematical point of view, it makes no difference in what order you do these multiplications, so long as you do them all and you add them to the right things. When you come and try to translate this algorithm for a computer which has a different storage plan, say a magnetic tape, or a drum, or something like this, you have to resequence and rearrange it to suit your machine. Now, this is an operation which cannot be done if too much has been put into the source language. Therefore I would like to put forward the plea that source languages contain the minimum of essential information and no extra information. It is a very difficult job and I do not think that we know how to specify it yet.

VAN DER POEL (Netherlands)—I must remark to Mr. Strachey that he is a little bit outside the scope of this discussion. We have already discussed it in other panels and I fear that we

must start here with specific source languages—it does not matter whether it is Algol or another language.

WEIZENBAUM (U.S.)—First I want to say that apparently I am the only American in this panel and I am glad I am not the one who mentioned *money!* Secondly I'd like to express a little disagreement with the remark of Mr. Strachey. It raises the issue of free processing which was referred to previously by Huskey, for example, and it may be that what is needed is not so much a language which leaves this kind of information out, but rather a technique for detecting such irrelevant information and then having detected it, substituting the suitable algorithms.

I think the general point of view is the following and this is the one I want to make. We talk about optimization in almost any field. There is always a temptation to do the easy thing which is local optimization. The hard thing—and generally speaking the most effective thing—is to do "strategic optimization." So, I think that, as a philosophy for efficient compilers, it may be a good philosophical point to attempt to look at some of the problems in the large, and make strategic decisions which should be subjected to optimization, and, before proceeding, to optimize all the details.

VAN DER POEL (Netherlands)—May I just make a remark on this. Optimization is called the best possible solution and here we can also bring into the discussion not only the best possible solutions, but efficient solutions.

BEMER (U.S.)—Before the confusion gets more general in my own mind, I should like to admit which language problems exist here in English. The title of the panel is "philosophies for efficient processor construction." I should like to ask the Moderator what this means in English: the efficient construction of processors, the construction of efficient processors or the construction of processors which are efficient for the total. And I feel that once the Moderator has restricted the subject to one of these three I can follow the discussion much better.

DUNCAN (U.K.)—First of all I apologize for misunderstanding the scope of the previous talk and therefore having to repeat this question. It would appear that if I want to implement Algol 60 on a small slow computer quickly and get an efficient program I must either (1) have a slow compiler (2) have slow object programs (3) not a compiler of Algol 60 but of something less

powerful. In other words, I cannot have all three ways. Now, I am not at all sure that we are kept and restricted in the language as it has been thoroughly investigated. I have a feeling that although we may get far from compilers and faster object programs, there may be a significant class of problems for which, because of the restricted language, the source program may need to be more cumbersome and complicated than it would have been had the full powers of Algol 60 been available. I should like to hear any comments on this aspect of efficiency and while I am here I'd like to quote what Gorn has just said to me, that maybe the best solution is not to try to get the best solution.

DIJKSTRA (Netherlands)—You stated the possibility that by introducing a more restricted language the thing becomes so cumbersome that overall efficiency gets lost. I should like to put the thing still more strongly, you are not only hindered by restrictions that prohibit you to do things, it is even so that you gain by possibilities that are not actually used in the program at all. One of the great features of our compiler is that it happens to turn out that it is very easy to have a good recursive function in it. I am very fond of them. They are hardly used by customers. Nevertheless, it is very important that they are in. The reason is that they give us possibilities that make the tool inspiring. So I am perfectly willing to incorporate a further thing like recursive procedures which are never used, dependent on their efficiency to make better programs.

VAN DER POEL (Netherlands)—I am too much biased myself so that I shall not go into discussion with Mr. Dijkstra on this point. But I assume that there are several people who are not in full agreement with this point of view.

STRACHEY (U.K.)—I think the question of simplifying or reducing a language in order to make the object program more efficient is extremely important. I disagree fundamentally with Dijkstra, about the necessity of having everything as general as possible in all possible occasions as I think that this is a purely theoretical approach which is very suitable in establishments but which is quite out of place for people who want to solve problems actually, particularly when we have small and cheap computers. The cost of making every possible procedure in Algol 60 is very considerable. I think that for a large class of problems a more restricted language is suitable and I should think it might be

possible to design such sets of modifications or dialects of more general languages which leave out the more general features and allow them in for those who want them. I would therefore strongly recommend that call by names in Algol procedures should be rather drafted than modified, and side effects of functions should be prohibited except when stated. And merely for things like these, one should modify a language when that would very considerably improve the object program.

VAN DER POEL (Netherlands)—Well, you can already see that two directions are developing: one, that you can better write restricted language to make fast programs; the other says "I'm making it very general, we can make it just as fast." I think the final word should be given to practice alone, and not by theoretical considerations, but as I said before I am much too biased, and I cannot enter too much into the discussion myself. But let us not restrict the discussion to this specific point. We want specific points for what is required for the efficiency of compilers.

STRACHEY (U.K.)—May I just make the point that I did not want to exclude general functions, I simply wanted to be able to use more restricted ones as well.

GARWICK (Norway)—For myself I completely disagree with Strachey and I completely agree with Dijkstra. He wants to have recursive procedures, but I don't think his reasons are good. People do not use them, and why? They have not been implemented and you can't use what you don't have. There is no doubt that there are some classes of problems which can be expressed very well in a recursive form but you can't expect people to use program facilities that programs don't have. You certainly gain quite a lot by supplying them with tools that they did not have before. So I am strongly in favour of recursive procedures.

INGERMAN (U.S.)—I'd like to make two points. First, I think that the recursive procedure, as an example, certainly should be implemented, if only to show that it can be implemented, which of course now is in fact true, but that there should be no question that a person, a programmer who is using the recursive procedure, does so at his own risk with respect to the efficiency in his own object program. The second point I'd like to make, essentially to Mr. Strachey, is that his point of wanting

to supply only the necessary information and nothing else is an appropriate sort of thing if, and only if, there is a convenient way, a reasonable way, or even a workable way, to define what is necessary. Programs are written by people and are used by people and they are implemented by machines but I do not think the machine is interested in the way the problem is presented to it. I would prefer to think that machines are not sentimental and I would suggest that having a language in which only the necessary information was stated would tend to be confusing rather than helpful, from the point of view of stating a problem.

MORIGUTI (Japan)—We Japanese have mostly small computers. When we discussed how to implement Algol on these machines we came to a point which I want to communicate on this occasion.

First, as Mr. Strachey said it, some sort of restrictions are desirable. I will not stop longer on this point at this time. Second, I am opposed to what Mr. Strachey said; we would like to have more information put into the language in order to help the compiler to do the work more efficiently without sacrificing the efficiency of the object program. Taking this point of view, two major examples came out. One is what Mr. Seegmüller talked about this afternoon, and that is to identify what are admissible indices, and this task can be done by programmers much more easily than by the compiler, at least in the case of smaller computers. Therefore we incorporated some means for putting the information into the language. We put 'comment' and then we state admissible or something, and then we go on. The actual device is not important here. Another example, the so-called optimization on repeated expressions. For instance "a plus b" is repeated in several places in a single statement; we can optimize the object program a little bit by storing this "a plus b" in some place. But this repeated expression can be identified by the programmer more easily than the compiler, and so we would like to put some indications about this fact into the source program. And one candidate I recommend to you is the operator 'plus' to indicate the repeated expression in the 'for' statement. This will not conflict with the present Algol 60 report. We give up some freedom to utilize this information for the purpose of getting better compilers.

WEIZENBAUM (U.S.)—I will first of all call attention to the fact that we are engaged here in what might be called a product plan-

ning effort, and in a large effort of this kind it is not reasonable to suppose that only one product is going to come out of it. We are going to have many kinds of compilers, and so on. I think that is obvious. So it is not a question of doctoring this philosophy, or that or the other; there is a place for Jovial, there is a place for very small compilers, and so on. I think it may be just up to the user to decide what sort of transportation he wants to use when he decides to go from one place to another, many forms are available; so the user may be very well in a position to decide which abbreviated version of the given compiler's system he may wish to use. I think the next step, then, in this direction is that we will soon see that many of these decision making tasks, that is deciding which means of transportation to use, are themselves capable of computable solutions and will provide the free processing, as hoped by this discussion.

DIJKSTRA (Netherlands)—I should like to draw your attention to a couple of aspects which have not entered the discussion at all, and I center them around the recursive procedure and whether they should be in or not, because it is such a well known example. This is still not a reason to put a funny thing like that in. This is not even the reason that it is unfair to the implementor to deprive him of the full challenge.

My next remark is again connected with some remarks made by Strachey. Mr. Strachey and Mr. Wilkes together have suggested in a recent publication to improve the efficiency without reducing the flexibility of the language, in a way for instance that normal procedures could not be used recursively, and they introduced an additional declarator which could state the recursive procedure if it was to be used recursively. This seems a very wise decision. You are saving both: you obtain a greater efficiency on one hand and you don't lose any flexibility on the other hand. Nevertheless, to check whether a procedure is to be used recursively or not is not a trivial matter and I dislike to see this burden placed on the shoulder of the human user when the machine can do it. I am perfectly willing to defend the attitude of programmers who always write recursive procedures just to be sure that it works correctly and not to be bothered.

Then a small remark. At every moment small computers turn up. I have the feeling that it is not the size of the computers which is decisive but a series of other qualities or maybe lack of qualities.

SAMELSON (Germany)—I feel I must say my initial remark is still unchallenged as it is in agreement with what Dijkstra said: when talking of solving problems we never speak of "possibly" solving problems. What has been stressed by other people too is for me the main thing: it is the economic handling. Now what does economic handling in the solving of problems really mean? There are several points involved: the efficiency of the generated program, the efficiency of the translator, the language itself. I think Naur has given all these and we have not tried to look at them all. I feel we really should try to find out what are these aspects, these main important aspects. I can give my own opinion that the efficiency of the running program is very high, but, on the other hand, when talking about problem oriented language I admit other considerations, otherwise we would stick to machine codes, which will certainly give the most efficient running program that we can get. So the consideration of the language used by the programmer has a bearing too. In my opinion the efficiency and the speed of the translator and the translation process itself are of secondary importance. But we have heard that for small computers it might be desirable to have a language which gives more access to the facilities present in the computer than some problem oriented languages presently allow. This is an immediate valid reason to try to modify the language for the user's benefit in certain respects. I think in that respect I agree with both Mr. Duncan and Mr. Moriguti. We should try to talk on these subjects.

NAUR (Denmark)—As I see it, clearly the fundamental question is "should man slave for the machine or should the machine slave for man?" To me this is a fundamental issue. Personally I am quite positive that the machine should slave for me. Of course, in practice, in a particular installation, conditions may vary. In a place you have very limited machine capacity and a lot of people around who want to use it. Well, you have to save machine time by all means, even if you waste these people's time. But the moment these conditions turn around, I would hate to see that advantage was not taken of the fact. This is one point.

For the second point I want to make, I want to challenge particularly the statement regarding the cost of recursion. This is highly dependent on certain particular circumstances and I can assure you that in certain very small machines you have to do it

recursively because the machine capacity is so limited, since this is a very efficient storage saving device. This holds to the particular project in which I am now engaged. It is a very small machine, it has 1000 words of core store, it has a backing store on drum. We had to save by any means we could devise the use of core storage. Consequently, everything had to be pushed into the stack and all areas which are not active have to be deleted and be overwritten by program. It is very simple.

The third point I want to make: I think it is very exciting to notice this widespread and—it seems—fundamental disbelief in the long range value of generality. I was struck particularly by Professor Samelson taking this very short sighted attitude of the immediate economical consideration. If we want examples of the power of generality, there are plenty of them all over the place. Just take something as fundamental as our number system, the Arabic number system which was put into the place of the Roman numerals, and this was a replacement of a poor specialized notation with a marvelously general one, with marvelous effects. We can take another example: how was the whole field of automatic computers initiated? I know of one man who was surely interested primarily in the generality of principles, Von Neuman. Just think what this field would have been without a man of his vision of generality.

VAN DER POEL (Netherlands)—Thank you very much. I think I have now to moderate a little bit because I see that the reaction is becoming almost over critical.

STRACHEY (U.K.)—I would like to suggest, in answer to the last remark, that generality is a very good thing so long as it is not carried too far. We normally agree, when we outsight our particular profession, that the base we will use is 10. Now I would hate to think that when I was suggesting that it was necessary or valuable for efficiency reasons that we should be able to use a limited set of a language, I would hate to think that this implied that I meant that the general language was not to be available. I think that it is extremely important that generality should be available; for example that recursive functions should be available. I use them myself extensively and I think that they are very useful and much more powerful than any other form of programming. But for very small mathematical jobs they are just a waste of time. I think that it is very important that we should be able

to do them but I agree with Dijkstra that when the man is in front of all these functions it is better if he can get someone to help him. There is just one more point I want to make about the relative values of generality and that is this: if you have a machine in which all your arithmetic is done by means of subroutines, so that the balance between the generality necessary to do the red tape of procedures and your arithmetic is such that one multiplication costs the same time as all this red tape to do the procedure, then it is evidently a very good bet to have your procedure recursive. If, on the other hand, you have a machine where the cost of doing a recursive procedure is equivalent to 60 multiplications then you begin to wonder whether it is worth while calling procedures recursively. This is a very important point which is awfully overlooked.

CARACCIOLO (Italy)—I think that when speaking of efficiency of processors we are looking for a certain kind of universal efficiency. I don't see why one should not distinguish efficiency on different occasions. There are classes of problems and applications where the speed of the object program is essential. There are other problems, other cases, other situations in which the speed of the object program is not so essential and, instead, the generality of a system might be more appreciated.

Perhaps another point I should raise is how to increase efficiency, instead of just speaking of efficiency or processors. I think that in "philosophies for efficient processor construction" we also ought to take care of the problem of how to increase efficiency in general with respect to what we have today, and to what we are doing today.

POYEN (France)—The question I would like to be discussed is the following: given a language, given a problem, given a machine, what kind of compiler can you give me to solve this problem, and if I ask to save time in compiling, what time will I lose in running the object program. And vice versa, if I spend more time in compiling what will I win in the object program, that is what is the balance between compiling and object program. We all know that if we do many things in a compiler we save time in the object program but we waste time in the compilation.

VAN DER POEL (Netherlands)—I think it is very useful to bring this into discussion. Of course it is not quite general because the translation comes in only once and the run perhaps comes

in many more times, though it is not quite fair to make the comparison.

SEEGMÜLLER (Germany)—I want to support the point of view which was stated by Mr. Strachey. We have in Munich a big installation; we have about 100 permanent users. Our customers are all on the engineer level. We have no possibility and no time to debate all errors which come from the recursive functions and other devices, like side effects of functions, and so on. They are called neat devices. In my opinion they are very dangerous and much money is lost because of these neat devices. In this room here there are people from all over the world who are interested in efficient computation and much money is spent for aiming at this. With regard to Algol 60, all the people would be prepared to use it if there were not these annoying features. They are all willing to have it, if we could come to an agreement, to cancel such features. They would say, we know what we are using, and they would all use Algol at once. And the question is—to state it once more—that we want to work with this language, really to work and not to play with it, and I hope that we don't become a kind of Algol play-boys.

VAN DER POEL (Netherlands)—Of course this is a little bit apart from the subject in the discussion on whether we shall have recursiveness or not. I see that there is already a division in pro's and con's, but I do not think it has anything to do with efficient processors, as Dijkstra has said already, and Naur has corroborated this very much. I myself can say it too, that recursiveness incorporated in a compiler does not lead to inefficient programs. So, I must separate it from the main issue of the subject of this talk.

BERNSTEIN (U.S.)—I think somebody is missing a very important point here. I do not really believe that you can talk about efficiency in the abstract. The only way you get an honest measure of efficiency is to look at the installation in which you try somehow to optimize or maximize something. In the universities you have one aspect; if it is engineering, you have some others.

NAGAO (Japan)—You have just said that machines should serve man, and that is very correct. But if in general the object program is very inefficient and time consuming, we cannot perform

the computation. Can we then say that machines are slaves to man? I do not think so. And I think that three very important points should be considered: first the freedom to write any program freely: second, compiling the program and compiler construction itself should be made easy; and third, the computation speed of the general object program is very important. These three points contradict each other, therefore I think there should be some compromise to that. Compromises are always necessary for certain things and I think there should be some restrictions on Algol or else we cannot use Algol practically. We have to say also that we have made compilers which cannot accept freely the Algol 60 language, if there is a large amount of data that exceeds the memory space, and in fact we cannot accept arrays of infinite dimensions. So, there should be some further compromise possible, I think.

MATTHEWMAN (U.K.)—I should like to take issue with our Moderator who says, essentially, that if we do not want to have side effects, then we do not need to use them. I believe we are talking here of possibilities of efficient compiling. The compiler has to take account of the possibility that side effects might be occurring. It has no essential way of finding out whether this is happening or not. And therefore one is led to produce inefficient programs just because the side effects might be occurring, whether they are or not.

WILKES (U.K.)—Now the question is red hot. We seem to be falling into two camps. Those who believe in efficiency and those who don't! I would like to declare myself that I am all for efficiency. Efficiency in all ways, compiling speed, efficiency of the compiling program and the other efficiencies that have been mentioned. I think that we shall move in the direction of greater efficiency. I think that the movement of the last two years, which has been one in which people have been interested in language, for its own sake and ways of expression, without very much regard to its practical application, I think that in the long run that may be salutary, but I think it is a passing phase, and I think that very soon we shall enter into a phase in which programming systems are tested by their efficiency, and in particular by the efficiency of the object programs that they produce. Now, I was struck by Mr. Seegmüller's remarks and by the applause that they elicited. If it is true, and I think it may be, that these fea-

tures lead to the inefficiency of the object program in Algol 60, if it is true that if it had not been for these things Algol 60 would now be generally accepted—I think that this may be true—this seems to me a great tragedy. This is a serious matter. It seems to me a pity that there is not a movement to establish a good language with the features of other languages that could be generally accepted. I think the time is ripe for one to be generally accepted, and if anyone is interested in this point of view perhaps they'd like to drop me a postcard.

INGERMAN (U.S.)—I'd like to observe that there seems to be not only an ambiguity in the precise meaning of the title of this panel, but there seems to be also a very strong ambiguity in the precise meaning of the word "efficiency." I have some notes with myself, and I have six kinds of efficiency here that have been mentioned. Some of them only briefly. The first three are the efficiency of time with respect to running a program, with respect to compiling a program and with respect to programming a problem. The second three are the efficiencies in the sense of minimization costs, presumably again in respect to running, compiling or programming a problem.

The next problem I'd like to mention is that only in passing has anything been said concerning the environment in which these compilers exist, and I personally feel that the efficiency of a processor certainly can't be measured abstractly and in fact can be measured only with respect to the environment in which it occurs. For example, the gentlemen on my left—I am sure—would consider themselves as part of an environment in which an Algol compiler containing recursive procedures could well exist. There are others who would prefer to be not a part of that environment and they should certainly be entitled to absent themselves from it. Perhaps the approach is to say that when one is looking for an efficient processor, one should rather have an efficient family of processors and should be able to process one's program with whichever one of these processors maximizes the particular type of efficiency (and I mentioned six, there are undoubtedly others) that one happens to be interested in in this particular moment.

BEMER (U.S.)—I certainly concur with Wilkes. On hearing the panel here I noticed the complete absence of some elements which I consider extremely important to efficiency processors, and I would like to enumerate those briefly.

In the first place, a processor, as we have defined it, transforms, translates whatever you will from one language to another. It is a well known fact that if you know a language is going to be translated beforehand you can pass along additional information which can make the translation process easy. If you are speaking in French and you know that there is something which is going to be very difficult to be translated into English, you make a special dictionary note of this. Well, in this respect, I certainly support Mr. Strachey when he says that additional information given to the processor can have a great deal of effect upon the efficiency because the processor does not have to guess. A second area which strikes me is the question whether we process or translate one or more times. I know cases in the so-called higher languages where a single program is translated as many as sixty times before it is actually run. I can document this. Therefore it seems to me that a part of the character of an efficient processor shall be to save some information from time to time as is feasible. In other words, you might well document the object statements, the source statements, and also whichever statements correspond to them, so if you go in and actually make your diagnostic in source languages, change certain statements, you need to go through the list of statements that are produced as objects of your object instructions which are affected by the source statements. I think this is a very important fact for efficient construction.

Another thing that has been left out here and I feel strongly about is whether a processor shall compile or interpret, or a combination of both. Many things can be left to run time to be executed dynamically, some things can be done statically during the translation. I have not heard so far any mention of how these functions should be allocated.

I would say that processors ought to be built on exception basis. And here of course I support the proposal of Strachey about such things as giving an actual modifier for recursive procedures, and, by the way, I do not know why recursive procedures are so much a part of this discussion. There are some which are not recursive. Now, if you happen to have an Algol processor which treats all procedures, also recursive ones, then you can ignore this particular word in a source language, but processors should be definitely made on exception basis. Take the regular case of subscripts which are very simply implementable through index registers. This makes a far more effi-

cient object program if you tell processors that this is indeed the simplest case and to treat it so.

Now the last suggestion I wish to offer here. There has been no mention made about the distinction between procedure oriented languages, of which Algol is one, and the problem oriented languages, of which there are so many examples, such as the table languages. Now the question shall be: how is information allocated? Because we want a procedure language, do we write the characteristics that bear in any procedure start, or do we supplement them out as we do in commercial languages and give the characteristics in data, and say "this is a variable which has three whole numbers and four real numbers and the range is so and so." You may argue, and I think that some of you will probably do so, that this does not make much difference. If you look down to the title of this symposium, it says "in data processing." Now I submit to you that the talk here has not been defined and has been set mostly on mathematical processing and almost nothing on alphabetic information, symbol manipulation and such, which are also important processes. So I admit these points in the discussion of efficient processors, but I think that what has been discussed in the panel so far has been mostly on the minor problems and I would like to see also these problems taken up here.

VAN DER POEL (Netherlands)—We have to thank Mr. Bemer very much for having introduced this subject, and we are very grateful that he indeed mentioned the fact that there are other important fields of application of processor constructions, and not only the numerical field. Indeed this is very true.

WEIZENBAUM (U.S.)—I'd like to say that lots of the points that have been mentioned, particularly Mr. Bemer's, would have undoubtedly been mentioned by the panel, had the panel had as much time as some of the other speakers have taken.

HUSKEY (U.S.)—Bemer stole some of my fire. I think the whole point is this: if you want Algol widely accepted, then it should be considerably more used in computing payrolls. If you rub shoulders with people who sell computers, it is enlightening to see how they refer to their customers. One of the favorite terms is "unwashed masses." I am not sure that in dealing with the audience I would use the same term, but for a reason like this, just because the majority of people who use computers are not inter-

ested in the things we are interested in but they are interested in production, pay rolls, accounting, and so on, for these reasons people who sell computers for example don't talk loudly about algebra, and if they don't talk loudly about algebra they never talk about recursiveness.

And finally then I would like to support Mr. Wilkes. Really I think that you cannot expect a wide acceptance of Algol as a general language. And, if this is what you really want, then you want something else for a language; and I think that what has happened to Algol in the United States, for example, is a real support of this point.

NAUR (Denmark)—As regards the question raised by Wilkes, as far as I can understand, it is a suggestion of a subset of Algol. Perhaps I am mistaken in this, but in any case there exists a proposal for defining recommended subsets of Algol, all of which do not have this horrid recursion and other horrid features. Personally I am interested in noting that Wilkes has never officially made any comment on this although he has had his chance like everybody else who sees the Algol Bulletin.

WILKES (U.K.)—I will not want a subset only. There are features in which Algol is deficient. But I would wish to have, for example, the handling of multilength numbers and so forth. I don't think subsets would satisfy me altogether.

GORN (U.S.)—I'd like to follow up what Bemer said and remark that it is quite possible to define two types of procedure statements in Algol, one recursive and one not. And when one is using Algol, one should indicate whether he wants the recursive procedure. However let me point out that there are perfectly good scientific programs, rather than mere scientific computation programs, which would be very inefficient if they were not run recursively. I just sketched one out before and it would be a standard program which an engineer might use to compute an antenna pattern if he recursively uses integration inside itself and sines and co-sines inside themselves. It will also be inefficient not to have available recursive procedure, of course, for data processing usages as well. And let us not forget that general symbol manipulation in data processing includes compiler writings, and when one wants to produce compilers very fast and very cheap one needs recursive procedures.

BASTIAN (U.S.)—First of all I would like to point out that the sine of the sine of x is not recursive. The second point is this: I have not heard mentioned the use of packing of data, which if used in a compiler could be used very efficiently to handle problems which are completely beyond the scope of compilers presently available. Recently I had the opportunity to write a program which used a Boolean matrix which had 255 000 entries, a simple matter to handle by simply packing 37 elements to a word. I am sure something that all of us use daily, but which is completely unavailable in compilers, those that I have seen at least.

VAN DER POEL (Netherlands)—The time is up and I want to bring this discussion to an end but not before making a remark myself on something which Naur has said. He said that he always wants the machine to do the things that human beings would not like to do. I must take a little bit wider view: just because we human beings are able to do things which a machine can do, we are human beings. And I think it is very important to keep training and not quite let the machine do everything which a machine could do, but sometimes do the things ourselves.

And with this last remark I should like to thank all the panelists and thank all the speakers who have taken the floor.

DESIGN OF PROBLEM ORIENTED LANGUAGES—ALGOL AND COMPETITITORS

THE BASIC PHILOSOPHY, CONCEPTS, AND FEATURES OF ALGOL

Peter Naur

Regnecentralen, Copenhagen, Denmark

Perhaps it is not quite out of place to begin these remarks on ALGOL by making it clear that any opinions expressed are my own and do not constitute the agreed view of any group, formal or informal.

My purpose in presenting these views is to promote a discussion of the ideas underlying a language like ALGOL 60. More specifically, to bring into the light those ideas incorporated in ALGOL 60 which in one way or another may be described as basic, with a view to future language developments.

In doing this I want first to present some reflexions on what I consider must be the basic philosophy of the ALGOL effort. Then I will apply these ideas to a few of the more basic concepts and mechanisms of the language.

In my view the basic point of ALGOL is that it is designed to be a common language to be used for actual communication, or in other words that it is designed to be understood and used by groups in many different countries and centers and by people with a great variety in purpose and background. Thus the final test of the success of the effort lies in the answer to the question: Is the language used in this manner?

If this be granted it is then important to realize that the discussion of the language is not exclusively, or even predominantly, a question of scientific or technical points, but has important political and psychological aspects as well. In other words the characteristics of this language must necessarily be a compromise. We can have no absolute rules to guide us in the choice of its features. Instead this becomes a question of our present experience and also to a large extent a matter of taste. Thus it is a matter of taste how much stress should be put on making the language easy to learn, how rich in special mechanisms it should

be, to how high a degree it should follow established usage in other languages or fields (mathematics, logic, etc.), to how high a degree implementation problems should be allowed to enter the discussion, etc. The compromise to be established can appeal to none of these as an absolute standard. All we can hope is to make the language as a whole acceptable.

This lack of absolute standards of course extends to the very idea of the language being acceptable. Clearly there can be no question of absolute acceptance. There will always be certain people whose acceptance or lack of acceptance must be disregarded. Among these the most conspicuous group consists of people who simply do not accept the idea of a common language. These people seem to be quite common. They will tell us that they cannot accept ALGOL because they don't like such and such a feature and often they then walk off and produce their own variation of the language, a dialect. It is important to recognize these people when one encounters them because trying to please them is bound to be futile. They simply don't accept the basic thesis of the work: that the property of being common is more important than any specific feature of the language.

However, in this sea of compromises there is one steady rock where we are entitled to demand the absolute: the description of the common language must be complete, that is, it must define unambiguously what constitutes a legal text and what the meaning of every legal text is supposed to be. This is the point where criticism may rightly be raised and where admittedly a lot of the objections raised against ALGOL have been justified. The fact is that of course nobody before had had much experience in producing a complete definition of a somewhat elaborate language for electronic computers in terms which do not refer to any particular computer. It is quite clear now that we need far more powerful tools for accomplishing this task than the ones used in the ALGOL 60 report. This is overwhelmingly evident for the description of the semantics, but even the syntactic description, which is in a rather better condition owing to the use of the notation of Backus, must be improved in the future. As a matter of fact a number of syntactical rules of ALGOL 60 are not and could not be reflected in the metasyntactic formulae. Also the use of the Backus notation is not sufficient to avoid syntactic ambiguities. In fact, one of the well-known ambiguities in ALGOL 60 arises because the syntactic rules as they

stand make it possible to decompose certain constructions in two different ways.

This problem of description adds to the factors to be considered in the design of the language as a whole. Indeed, in order to facilitate the description it is desirable to build the language from a few very general concepts. If the number of independent constructions in the language is permitted to grow the problem of avoiding contradictions very rapidly grows alarmingly. It is interesting to note that difficulties of this kind are responsible for a very significant part of those ambiguities in the ALGOL 60 report which are presently being discussed. And this is the case in spite of the fact that the redundancy in this report is generally rather low, so low, in fact, that the dryness of the presentation probably has caused a considerable delay in the appreciation of the contents for many readers.

Thus to conclude this brief discussion of the basic philosophy, these are my main theses:

1. Since the ultimate goal is acceptance we can insist on no absolute standards with respect to the actual features of the language.
2. Where absolute standards may be demanded, on the other hand, is in the completeness of the description.
3. The practical difficulties of preparing a complete description increase considerably with the number of independent constructions within the language. This is an additional factor to take into account in designing the language.

I would now like to illustrate these points with a few remarks about some of the features now included in ALGOL 60, viz. the procedure concept and the for statement.

As is now widely recognized the powerful and flexible procedure concept is one of the most important features of the language. Essentially it is a general substitution scheme which permits the user to add additional mechanisms to the language at will.

The for statement, on the other hand, is a rather special construction designed to facilitate certain repetitive computations which in some applications are very frequent.

Now it is a fact that in spite of this difference of generality and conceptual difficulty the amount of work and discussion which has gone into the definition and description of these two mechanisms has been about the same. And the state to-day with

respect to the general satisfaction with them seems to be that the general and rather more difficult procedure concept is in a much better condition than the limited and simple for statement.

To me this experience indicates clearly that we cannot be too cautious in our demands for special mechanisms in our common language. Therefore, let us instead try to extend our general facilities, such as the procedure mechanism, so that it will satisfy all our individual needs.

In order to illustrate this approach let us just try to see what extensions of the present procedure concept would be required in order to enable the user to introduce his own for statement at will.

Considering first the case of one single step-until-element we get a good part of the effect by introducing a procedure defined by the following declaration:

```
procedure  FOR (controlled variable, initial, step, final,
           statement); value initial; real controlled variable,
           step, final; procedure statement;
begin controlled variable := initial;
test: if (controlled variable − final) × step ≤ 0 then
           begin statement;
                 controlled variable := controlled variable +
                                        step⌊;
                 go to test
           end;
end;
```

It is quite clear that this procedure FOR essentially expresses the semantics of the step-until-element. However, it is still not as convenient to use as the for statement of ALGOL 60 for the following three reasons:

1. The controlled statement has to be the body of a declared procedure.

2. The form of the activation is limited to the standard form of the procedure call.

3. More general kinds of for statements, including mixtures of elements cannot be handled by a single declaration.

Each of these defects points to a particular kind of generalization of the ALGOL 60 procedure concept. The first one indicates the need for admitting complete statements, and not only expressions, among the possible actual parameters. This would be an entirely logical development. Indeed, since a general expression

placed as an actual parameter logically is equivalent to the declaration of a type procedure, a more general statement placed in this position ought to be equivalent to the declaration of a procedure without type. This might be added to ALGOL without any trouble.

The second defect points to the need for some way of adding arbitrary new syntactic constructions, with meanings defined through procedure declarations, to the language. This is rather less straightforward, particularly since this would require a clarification of the question of the unique analysis of strings admitted within a given syntax. More research is needed here.

The third point indicates the need for a notation which would enable the procedure declaration to be called by procedure statements having an arbitrary number of actual parameters. The notation of ALGOL 60 is clearly inadequate for this need and I feel it would be difficult to amend it in a satisfactory manner. On the other hand if we decided to disregard the notation used for establishing the formal-actual correspondence in ALGOL 60 it would be easy to solve this problem.

On the basis of cases such as the one just described I believe it is correct to claim that considerable gains in the simplicity of the definition of the language, with no corresponding loss in the convenience of use, might be achieved by suitable generalizations of the procedure concept. Thus the moral of this story is to make the language general, not rich.

In conclusion I would like to point out that I have taken the identifier ALGOL appearing in the title of this talk to denote an ideal notion, namely the Supreme Common Language. I have compared some of my current thoughts about this ideal with the existing approximation called ALGOL 60. Now I wish to make it quite clear that the fact that I have found some discrepancies between the ideal and the reality to me is not a sufficient reason to reject ALGOL 60 at present. ALGOL 60 is only just on the way to being reasonably well established and impatience at this stage might confuse the development completely. So I urge you to be patient with the imperfections of ALGOL 60. In the meantime we have to continue thinking, and if we think hard enough (and only then!) we may, in 5 or 10 years from now, be in a position to produce a language which is really much better.

THE DESCRIPTION OF COMPUTING PROCESSES. SOME OBSERVATIONS ON AUTOMATIC PROGRAMMING AND ALGOL 60

M. Woodger

National Physical Laboratory, Teddington, England

SUMMARY

This paper is the result of an attempt to sort out the basic ideas involved in the description of a computing process as a program for a computer, expressed in a formal symbolic language such as Algol 60. The emphasis is on the information conveyed by the program constituents, i.e., its semantics, rather than the particular form used, i.e., its syntax.

A preliminary section discusses the fact that a process description is in practice always incomplete, and relative to some assumed level of detail of analysis, in the sense that it indicates a sequence of subprocesses (such as arithmetic operations) to be carried out in a particular order, but stops short of the description of how each subprocess effect is achieved. The essential features of the use of names for reference to stored information are reviewed.

Next some characteristic features of Algol 60 are mentioned, and detailed suggestions are made as to how they may be generalised and unified so as to reduce the number of anomalies in that language and consequently to reduce the number of separate ideas which have to be assimilated in order to understand it.

The principle adopted in this generalisation is to observe what facilities have already to be provided for the correct handling of Algol 60 as source language, and to allow the free use of these facilities. One important feature is the use of lists of quantities as values of variables, assignable as a whole, as the result of evaluating expressions of type **list** (i.e. vector) or **list list**

(i.e. including matrix). A second important feature is the use of names (identifiers) in general as permissible values of variables of type **name**. The avoidance of generating program at run time, as in Algol 60, is maintained—an expression or statement may not be produced as value of an expression and assigned to a variable, although it may be associated with a name by appearing in a declaration.

1. GENERAL REMARKS

1.1. The physical realisation of a computing process as a program for a digital computer depends essentially upon two things. In the first place, the data to be processed is expressed in finite digital form. In the second place, the process itself is expressed as the overall effect of a finite number of discrete steps to be carried out in a specified sequence.

A complete description of such a process would detail each of these aspects down to the last digit, specifying how each digit of the result of the process was to be derived from each digit of the data.

In practice, such a description is never given; instead, familiarity is assumed with the details of a class of common constituent subprocesses, such as arithmetic operations on particular floating-point representations of real numbers using the conventions of the particular computer employed, and the process to be described is analysed only down to the level of a sequence of such subprocesses.

If these subprocesses comprise the basic machine order code the analysis is complete as regards the particular computer in use. The discrete steps of the process correspond to the execution of individual machine instructions, and the finite digital form of the data at this level of analysis is governed by the word length and fixed conventions regarding number representation. The programmer is here not concerned at all with the way in which the individual machine instructions achieve their specific effects, or the order in which the digits of their results are produced, but only with the effects themselves.

It is clear that the description of a process may either be timeless, in the sense that it simply states a result and in no way how the result is achieved, or it may in greater or less detail specify a temporal sequence of subprocesses which will produce this result, these subprocesses being themselves de-

scribed only as to their results and not as to how these are achieved. Thus a subroutine for calculating the cosine of a floating-point number may be described 'timelessly' in terms of its overall result in its title, and processwise in terms of the basic machine code in the detailed program or flow diagram.

The statement of a data processing problem is a description of a desired process-result, a functional relationship, and its solution from the programmer's point of view is an expression of this result as the outcome of a particular sequence of available programmed subprocesses.

1.2. A third fundamental feature of processes carried out by digital computer is of course the use of stores for containing representations of the process description itself as well as of the data being processed. An isolated use of a numerical constant in a calculation might only require it to be presented once at the input, and not stored. Likewise an instruction to perform a particular operation just once could be accepted once from the input and not stored. Storage must be used when an item of information (a datum or an instruction) is to be referred to (i.e. used) more times than it appears at the input, including the special case where it is generated internally by the process itself.

We are using the word 'instruction' in the sense of a representation of a sub-process description in the form of a stored pattern of digits which will cause that process to take place when this pattern is 'obeyed' by the machine, that is to say, when the 'control' of the machine is directed to it. This presupposes mechanisms of selection of such patterns in store, and recognition of their extent, in addition to interpretation of each pattern as it is obeyed. The specification of the path followed by control when obeying the instructions is part of the process description itself, but is usually only made explicit where there is to be a deviation from simply following the linear arrangement of instructions in store.

1.3. By the *current state* of the machine at any instant *between* subprocesses during the execution of a program is meant the totality of stored information at that instant, together with the position of the instruction about to be obeyed at that instant (the 'position of control'). This information determines the subsequent events, so that the process could be interrupted and subsequently

completed on another occasion if only the current state were preserved.

1.4. Information *used* by an instruction (its 'data') may either

(a) appear *explicitly* in the program text with it (its position and extent governed by notational conventions), or

(b) be available *implicitly* in store at a position determined by convention [such as the accumulator, the 'store of position of control' or control counter, or the top cell of the 'stack' where this method of store management is used (1)], or

(c) be *detached* information, available in store at a position determined by an identifier or *name* appearing in the instruction. (The identifier in this case is itself explicit information).

Information *produced* by the execution of an instruction (its 'results') may likewuse appear either implicitly in store at a position determined by convention (the accumulator, the control counter in the case of control jumps, or the top cell of the stack) or detached at a position determined by an identifier appearing in the instruction. It is this last detached information that is considered to be the *subject matter* of the process, since it alone can be altered in store by such an 'assignment' of a value to a name. In this sense the subject matter of an Algol 60 program comprises only single integers, real numbers, and logical values.

1.5. The use of names as parts of instructions for referring to detached information in store has the following important consequences.

(a) Conventions must be assumed to connect the name with the store named. In particular, since storage is precious, conventions are required to sever this connection when no longer needed and thus free both storage and name for other use.

(b) An instruction cannot be obeyed until all detached information which it uses occupies the named stores.

(c) The stored information is independent of the instructions which use it (as data), in the sense that it may be altered without altering these instructions themselves.

(d) An instruction using detached information expresses a generalised *function* of that information, in that its effect depends directly upon that information.

2. ALGOL 60

2.1. Among the essential features of Algol 60 (2) are included the following.

(1) A completely nested (bracketed) structure of the program text, used to define definite scopes of *all* identifiers employed for reference to detached information.

(2) Complete freedom to elaborate and to compound individual program structures to an arbitrary degree.

(3) The provision for defining (with a few restrictions) a *procedure* from any statement or block and any fixed list of identifiers as 'parameters', choosing for each one that it shall be 'called by name' or 'by value', and invoking such a procedure by name (as statement or as expression) with corresponding lists of 'actual parameters'.

(4) No reference whatever to storage of information.

(5) Subject-matter (i.e. information which can be generated and named) restricted to single numbers (real or integer) and logical values. (The provision for handling arrays does not permit the generation of an array by an assignment statement, so they are not subject-matter in the true sense, but rather ordered sets of numbers or logical values associated with a rule of selection using index values.)

(6) As a particular consequence of (5), no operations upon parts of the program itself are expressible, so that all association of statements, expressions or procedures with their names may be done in a fixed manner in a preliminary translation phase before the calculation proper begins.

In Section 3 I shall try to show some ways in which these features might be extended and unified, with a view to both strengthening the means of expression provided by the language and reducing the number of separate ideas which have to be assimilated in order to understand it.

2.2. In Algol 60 a declaration in a block head introduces an identifier and defines its scope as being that block (including the bodies of procedure declarations in its head, but excluding subblocks in whose head the same identifier is redeclared). It also provides all the detached information which will be referred to by instructions within the block which use that identifier and which is unalterable in the block (i.e. all but the actual values of variables).

For type declarations this information is the type of a simple variable. For array declarations it includes also the selection process whereby a list of index values is used to select one variable in the array, and a list of pairs of arithmetic expressions (i.e. programs) used once on entry to the block to specify the size and shape of the array. (So-called 'dynamic array declarations'). For switch declarations it includes the selection process whereby a single index value is used to select a designational expression in the switch list, and includes (in the form of programs) these designational expressions themselves. Finally for procedure declarations the detached information provided is a process whereby a list of actual parameters is used to complete the description of a piece of program to be executed.

2.3. The inherent limitations on process description which were outlined in Section 1 are recognised in Algol 60. A process can be described in terms of a chosen set of subprocesses defined by procedure declarations, and its description apart from these declarations is independent of how these subprocesses are in fact achieved. But as regards the form of the data to be processed the language is on the one hand limited in detail to individual real numbers, integers and logical values, and in the other direction it allows no synthesis of these into compound forms which might be assigned as values of variables. Reference to the exponent or mantissa of a floating point number is excluded, as are also matrix operations (other than as defined by explicit procedure declarations).

3. OUTLINE OF THE SUGGESTED SEMANTICS.

3.1. The *subject matter*, i.e. information which may both be generated within the program by the evaluation of an expression and also then *stored* as the value assigned to a name, may be of one of the following *types*. (This name is then a *variable* and this is *its type*.)

A **real**
or **integer** number, for use in calculation.
A **boolean** value (i.e. logical value, **true** or **false**), for choice between alternatives.
A **symbol**, for text manipulation.
An **index** value (i.e. positive integer), for selecting a component of a list.

A **name**, for indirect reference, including control jumps.
A **list** of values of any of these types, for associating these together as components of one entity.

The intention is that knowledge of the type of a variable shall suffice to determine the manner of use of its value in the program, without examining that value itself.

In the case of a variable N of type **name**, i.e. one whose value V is a name, no further information is needed if this value V is to be simply assigned to another variable or inserted in a list. But if V is to be evaluated (or otherwise used, if not itself a variable) it is necessary to indicate in the type of N what kind of name V is, that is to say the *specification* of the name V. Following the classification of the uses of names given in section 3.2 we include in the type **name** the following subtypes.

real name (i.e. name of a **real**)
integer name
boolean name
symbol name
index name
label (i.e. statement name)
expression name
procedure name
list name, including **real list name** (i.e. name of list of **reals**)
etc.

In the case of a variable N of type **list**, whose value V is a list of subject-matter items with a common type, the use of this list requires a knowledge of how to select its components, and a knowledge of its extent in store if it is to be assigned to another variable or joined to another list to make a longer one. The *length* of the list, i.e. the number of its components, is part of the type of N if assumed constant, but part of the value if to be altered. According to the common type of the components, we include in the type **list** the following subtypes.

real list (i.e. list of **reals**)
integer list
boolean list
symbol list (i.e. string)
index list

name list, including **real name list**, etc.
list list, including **real list list** (i.e. list of lists of **reals**), etc.

Appearing alone, these subtypes imply lists of alterable length. Fixed length m is indicated by following the type indication by (m), and in the case of type **list list** with a fixed number m of components each of which is a list of fixed length, n, this is indicated by **list list** (m,n)

3.2. A *name* (identifier) is either
(1) a *variable*,
or (2) a name of a particular *statement* in the text of the program, indicated by the name being attached to it as a *label* instead of appearing in a declaration in a block head,
or (3) a name of a particular *expression* exhibited in the declaration for the name,
or (4) a name of a particular *procedure* exhibited in the declaration for the name.

3.3. An *expression* is a piece of program which produces a result (its value) when executed (evaluated). The *type* of the expression is the type of this result.

Evaluation of a constant is interpreted to mean producing the constant itself as value. Evaluation of a variable yields the value last assigned to it.

Evaluation of a list means evaluation of its components in turn, resulting in a list of their values.

Since we admit values of type **name**, any expression of this type could be further evaluated, so we adopt the convention that evaluation stops as soon as the value can be used in the context in which the expression appears. If further evaluation is nevertheless intended it is indicated by a special operator. (Script *v*).

3.4. An *assignment* statement takes the form

N := E

where N and E are expressions or lists of expressions.

This means "Evaluate N, then evaluate E, then store the value of E in the store indicated (named) by the value of N". If the value of N (but not that of E) is a list, then the value of E is assigned to

each component. When N and E are lists of equal length assignments of corresponding components occur. The types of values assigned must be compatible with the types of the corresponding names.

In accordance with the convention stated in the previous section, the evaluation of N is suppressed when it is simply a variable of the same type as E, since it then already names a store for the value of E.

3.5. A *declaration* is a piece of program in a block head specifying the use of a name in that block.

3.6. A declaration for a *variable* reserves space for its value and specifies its type, and may be combined with an assignment of an initial value to the variable. If N is the name and T the type the declaration takes the form

T N

or TN := E

where E is an expression whose value is to be assigned initially. E is to be evaluated using declarations valid at the moment of entry to the block, *not* other declarations in the same block head.

3.7. An *expression declaration*, i.e. a declaration for a name of a particular expression, exhibits the expression and its type (i.e. the type of values produced by executing it). If N is the name, T the type and E the expression, it takes the form

T N : E

Occurrence of this name in an expression invokes the evaluation of the expression E. Thus one has in effect the facility of substitution of an expression for a name (as when calling procedure parameters by name).

In the special case when E is simply a variable, N is effectively equivalent to E.

In the special case when E is simply a constant of type T, N is effectively equivalent to this constant.

E may in particular be a list of expressions, in which case T must be a subtype of list.

3.8. A *procedure declaration* exhibits the name, the procedure (a statement or block, which may (but need not) produce a result as value), the type of the result it produces (if any), and a list of names and their specifications (the *formal parameters)* which may occur in the procedure. It also indicates which of the formal parameters are to be *called by name* and which *by value*. If the procedure yields a result, this is produced as in Algol 60 by assignment to the procedure name. This may occur one component at a time if the result is a list.

3.9. A *procedure statement* consists of the procedure name (or a primary expression of type **procedure name**) followed by a list of expressions (the *actual parameters).* It is an instruction to execute the procedure *with declarations for the formal parameters attached to its head,* making them local to it, and converting it into a block if it is not already one. For the parameters called by name these declarations are expression declarations using the corresponding actual parameter expressions, but to be evaluated using declarations which apply at the place where the procedure statement occurs, while for the parameters called by value they are variable declarations with initial assignment of value, using the corresponding actual parameter expressions evaluated in the same way.

3.10. A *statement,* i.e. a unit of program with no designated 'result', may be:

an assignment statement,
a procedure statement,
a go to statement,
a dummy statement,
a conditional statement, } as in Algol 60
a sequence statement,
or a compound statement or block as in Algol 60.

3.11. The *sequence statement* is a generalisation of the 'for statement'. It contains as part a statement S. It may contain a list of formal parameters, which may occur in S, and then indicates which are to be called by name and which by value. It may contain a *sequencing clause,* i.e. a piece of program which produces a list of values of corresponding actual parameters (called by value) each time it is invoked, or a *for list,* i.e. an explicit list of such

lists of actual parameters (expressions, called either by name or by value). Finally it may contain a *while clause*, of the form "**while** *B*", where B is a boolean expression.

The sequence statement is an instruction to evaluate B, do nothing further (but proceed to the next statement) if this value is **false**, otherwise evaluate the sequencing clause, execute S as if it were a procedure called with the actual parameter values produced by the sequencing clause, and repeat this cycle of operations by evaluating B again. In the absence of a *while* clause the cycle begins with the sequencing clause, which then has the additional effect of terminating the execution when a stated list of values is produced. In the absence of a sequencing clause S is merely repeated until B takes the value **false**, and in this case there are no parameters. If there is a *for* list in place of a sequencing clause its components are used in turn as actual parameters, and the execution is terminated after use of the last component, if not previously by the *while* clause which may also be present.

A sequence statement is thus analogous to a procedure call, with the procedure attached to it rather than invoked by name.

3.12. As in Algol 60, *expressions* are built up from 'primary' expressions using suitable operation signs. The primary expressions of any type T include representations of constant values of type T (such as $-2.03_{10}+5$ for type **real**, or **false** for type **boolean**), variables of type T, function designators of type T (interpreted following 3.8 and 3.9 above), and general expressions of type T enclosed in parentheses. In addition to these primaries there are others which result from operations on expressions of types different from T. Thus as in Algol 60, *relations* of equality or inequality between arithmetic expressions are primaries of type **boolean**, having value either **true** or **false**.

3.13. As in Algol 60, a boolean expression B and two espressions E and F of the same type T can be combined to form a *conditional expression*

if B **then** E **else** F

which is also of type T.

3.14. Any actual list of primary expressions of type T, written down with commas between them, constitutes an expression of type *T*list whose value is the list of the values of these expressions taken in that order. The comma can be regarded as a list-forming operator.

3.15. If U is any operation sign which operates on two operands of type T, producing a result of type R, then U may be written between primary expressions of type *T*list and will yield the result of type *R*list which is obtained by operating on corresponding components of the two lists of type T (resulting from evaluating these primary expressions). Evidently this requires that the two lists be of equal length. We also permit U to be written between an expression of type T and one of type *T*list yielding the result of type *R*list which is obtained by operating on successive components of the list with the same operand of type T.

If U operates on a single operand of type T, producing a result of type R, then U may operate on a list of operands of type T and will produce a list of results of type R, i.e. a result of type *R*list.

These facilities correspond to the suggestions of R. W. Hockney in the particular case of matrix operations (3).

3.16. If E is any expression of type **name**, then

$$v\,(\mathrm{E})$$

is an expression whose value is the value of the value of E. The script v is an operator indicating one further step of evaluation (cf. 3.3 above).

3.17. Indices, Names and Lists.

An index is used as an address of a component within a list, the first component having address 1.

The ability to assign a value to a particular component of a list without disturbing the other components implies the ability to name that component. This name is *derived* from the name of the list and an index value, and is in fact the only case of a name which is not explicitly present as an identifier in the program. We thus regard the expression

$$\mathrm{L}\,[\mathrm{I}]$$

as an expression of type **name** which names the i^{th} component of l, where i is the value of the expression I of type **index** and l is the value of the expression L of type **list**. It is important that L is *not* here to be evaluated completely. The process intended is the evaluation of I to get i, then evaluation of L to get the name of a list l, and finally the formation of a name of the i^{th} component of l. If L [I] is on the left of an assignment statement the evaluation stops here. If it forms part of a larger expression the selected component of l may itself be used to continue evaluating that expression, so that one further step of evaluation is involved.

Multidimensional arrays are represented as lists of lists and correspond to the case where the above component of l is itself a list. If j is the value of an expression J of type **index**, and L[I] is of type **list name**, then

$$L\,[I]\,[J]$$

is an expression of type **name** whose value is a name of the j^{th} component of the i^{th} component of the list of lists named by the value of L. The expressions I and J are to be evaluated in that order, and the resulting list of indices used with the name of the list of lists to form a name of the selected component. The same process is expressed by

$$L\,[I, J]$$

where an explicit list of expressions of type index is to be evaluated.

Since names may be stored as values, assigned to variables of type **name**, it is possible to separate the selection process from the use of the resulting name. This can be used to advantage in some matrix processes.

3.18. Constants

Constants of type **real**, **integer** or **boolean** are expressed as in Algol 60. A constant of type **index** is simply an unsigned integer other than zero. A constant of type **symbol** is represented as any basic symbol (including the 'space' symbol ␣ if necessary) enclosed in string quotes ( ). An identifier which is used as a label is a constant of type **label**, a subtype of **name**. Other identifiers used in the program are constants of other subtypes of **name**, according to their declarations. A constant of type *T* **list** is simply an explicit list of constants of type T, written with separating commas and enclosing parentheses.

3.19. Strings.

Strings of basic symbols are represented as lists of constant values of type **symbol**, and the convention is used that a list such as

('c', 'a', 's', 'e', '␣', 'X', '>', '−', '4')

can be abbreviated to

'case X>−4'.

In general, the occurrence of more than one symbol between string quotes indicates a list of this type. The type name "**symbol list**" could be abbreviated to "**string**".

3.20. Labels

As in Algol 60, an identifier attached as label to a particular statement in the program specifies the use of that identifier, within the block to which that statement belongs, as a name of that statement. This is like a declaration which is not in a block head.

A label expression is what in Algol 60 is called a designational expression. It is of type **name**, subtype **label**, since its value is an actual label, i.e. an identifier used as such in a block enclosing the piece of program where the expression appears. (This value may be assigned to a variable of type **label**).

A switch is a name of a list of label expressions, and is defined by an expression declaration. It may be used with an index as a label expression, as a switch designator in Algol 60, but it may also be used alone as an expression of type **label list**.

The general form of a 'go to' statement is

go to L

where L is any label expression, which may in particular be a constant (an actual label), or a variable of type *label* (which is then to be evaluated).

4. DISCUSSION

4.1. The above suggestions are admittedly incomplete. They stop short of a detailed syntax, and a full set of operations useful in widely occurring contexts. The reason for this is that whereas it is hoped that the proposals given are sufficiently general and fundamental to be strong candidates for adoption in a general-purpose programming language, it appears that elaborations in a variety of likely directions would be governed more by the

kind of application intended, and an arbitrary choice among the possibilities would have to be made. Thus if list processing *per se* were the main application, then an arbitrarily extended set of subtypes of **list** might be required, probably allowing list components of differing types. If processing of strings of binary digits in general is to be a chief consideration, then these may be regarded either as of type **symbol list** or of type **boolean list** (depending on the interpretation) and a suitable set of operators has to be defined. It is evidently undesirable that a translator be equipped to deal with all such specialised fields of application simultaneously, unless it happens that a common simple set of operations will suffice, on account of the size of the program.

A detailed syntax is best left until the semantic constituents of the language are chosen, to avoid using clumsy constructions for commonly occurring operations.

4.2. The above treatment has dealt with the most natural form of synthesis of compound forms of data from their constituents, namely as lists, but has not provided for analysis of values of type **real**, **integer** or **index** into their constituents. Operations on the constituent digits of integers are not generally required, but it frequently happens that a **real** or floating-point number has to be treated as composed of two numbers, namely its exponent e and mantissa m, with e an integer and $|m| \leq 1$. In that the mantissa is represented to a fixed number N of digits (binary or decimal), it may be treated as an integer of limited size N digits, and the associated exponent then becomes e−N. We could therefore treat **reals** as of type **integer list** (2), and interpret the usual arithmetic operations as producing 'normalised' lists of this form, i.e. lists L such that the integer L[2] is either zero or lies between 2^{N-1} and 2^N in magnitude (base 10 if decimal representation is used). The justification for not doing this can only be that a great many numerical processes do not require such analysis.

4.3. With the introduction of the simple notion of a list whose components may themselves be lists, and an index used as ordinal number for counting the components, the concept of 'array' is no longer required, and such aspects of Algol 60 as lower bounds of indices and dynamic array declarations seem arbitrary and unnecessary.

Whereas no convenient upper bound can be chosen for the

magnitude of integers in general use, the magnitude of indices whose explicit purpose is for counting in stored lists can readily be bounded in practice due to the finite extent of the store. A half-word commonly suffices to store an index value.

4.4. Lists have been distinguished as of fixed or variable length. The way of managing variable length lists in practice will depend upon the main applications intended, but could be via a 'free storage list' as in the LISP system (4), where the list structures are effectively chains of indirect addressing references terminated by recognisable 'atoms'. For applications involving more restricted variations it might be adequate to combine a fixed maximum storage reservation at declaration time with automatic extension by a fixed amount on subsequent overstepping of the currently allocated space. Such a system preserves as far as possible the principle of storing successive list components in successive storage locations or blocks of locations.

It is worth remarking that integers are already stored as lists of digits, and treated as of fixed length (one word), so that multiple precision working can be described either as operations upon lists of digits exceeding the standard length, or alternatively as rather more artificial operations upon pairs (or longer lists) of integers.

4.5. The description of the effect of procedure calls has been simplified through the use of expression declarations (which are useful abbreviatory facilities in their own right) and by allowing declarations for variables to be combined with initial assignment of values, calculated at the moment of entry to the block or procedure body. A description in terms of a substitution operation which does not in fact take place is thereby avoided.

Expression declarations have been suggested at various times by different people, under the name 'function declaration'(5). The present proposal effectively includes as special cases the 'equivalence declaration' and the 'constant declaration'(6).

4.6. The 'for statement' of Algol 60 embodied the essential features of the present sequence statement (section 3.11) but in a restricted form. The 'step-until' element represented a sequencing clause which produced a list of values to be assigned to the controlled variable each time it was invoked (i.e. at each test for termination), but these had to be in arithmetic progres-

sion. The arithmetic expression elements in a 'for list' are retained, but the 'while element' is separated from the annexed arithmetic expression of Algol 60 in order that it may be combined, as required, with the other elements. This is often needed in practice.

The controlled variable of the 'for statement' of Algol 60 has been replaced by a list of formal parameters, thus recognizing the bound nature of the variable and extending it to a flexibility equivalent to that of the information supplied at calls of a procedure. The 'for list' of Algol 60 which involved expressions evaluated at each cycle corresponds here to a for list with actual parameter expressions called by name; call by value is also possible, in which case a list of values is produced at commencement of the sequence statement. The requirement for simultaneous sequencing of several variables has been expressed before.

ACKNOWLEDGMENTS

I wish to express my thanks to Dr. E. W. Dijkstra of the Mathematical Centre, Amsterdam, for his helpful comments which clarified my understanding of process description.

The work described above has been carried out as part of the research programme of the National Physical Laboratory, and this paper is published by permission of the Director of the Laboratory.

REFERENCES

1. E. W. Dijkstra, "Recursive Programming", Numerische Mathematik, **2**, 5 (October 1960).
2. P. Naur (Ed.), "Report on the algorithmic language ALGOL 60", Num. Math., **2**, 106-136 (1960).
3. R. W. Hockney, "A.B.S. 12 ALGOL", Computer Journal (February 1962).
4. J. McCarthy, "The LISP Programming System", Quarterly Progress Report No. 53, Research Laboratory of Electronics, Massachusetts Institute of Technology, (April 15, 1959).
5. A. van Wijngaarden, E. W. Dijkstra, Algol Bulletin 7.34.2, (October 1959).
6. Algol Bulletin 4.10 and 4.7, (August 1959).

GENERALIZED ALGOL

A. van Wijngaarden

Mathematisch Centrum, Amsterdam, Netherlands

The title "Generalized ALGOL" of this paper needs an explanation. The word ALGOL is used because of the fact that many of the concepts of the language to be described can be found, partially at least, in ALGOL. On the other hand the generalization goes to such an extent that the connection with ALGOL can only be appreciated by those who know ALGOL quite well.

The main idea in constructing a general language, I think, is that the language should not be burdened by syntactical rules which define meaningful texts. On the contrary, the definition of the language should be the description of an automatism, a set of axioms, a machine or whatever one likes to call it that reads and interprets a text or program, any text for that matter, i.e. produces during the reading another text, called the value of the text so far read. This value is a text which changes continuously during the process of reading and intermediate stages are just as important to know as the final value. Indeed, this final value may be empty.

In order that such a language be powerful and elegant it should not contain many concepts and it should not be defined with many words. On the contrary by saying less one can say more, at least say more general things. Each definition in the language may restrict the set of meaningful texts. Without any definitions, however, one can only be absolutely silent in full generality. Of course, some compromise must be made in practice. This compromise has been made in ALGOL in a certain way. There are other ways, however, by which a better defined and more general language can be obtained using fewer concepts. In this short paper not a complete technical description will be given, but only some features will be described.

Let us first discuss the way in which such a syntax-free language might be described. Logically, the best way is to give the precise axioms or the precise description of the machine. However, such a definition would give little insight perhaps in the way in which one has to write a program in order to obtain a

wanted result. Also in the definition of the language there might be a distinction between fundamental concepts and useful but logically unnecessary conventions. Hence, we rather see the language as a machine M0 which is fed with the program at one end and produces the value at the other end. The rules of the language, i.e. a rough description of the working of M0 is printed on the lid of the machine so that the user can easily see how to use the language. This description is quite simple, quite elegant in a way and will suffice in many cases taking into account that the user will often take for granted that in the language certain expressions like $x + y \times z$ will stand for what he himself understands to be the meaning of $x + y + z$. If, however, the user does not trust his intuition or does not understand what the short description on the lid implies in a particular case, he can open the machine to inspect the precise working. To his surprise, he finds that there are actually two machines inside, named P1 and M1. The working of the machines is explained in much more detail on the lids of the machines. The machine P1 is a so-called preprocessor, which chews the offered text and produces another text in a more basic language which is evaluated by the processor, i.e. machine M1. In the text offered to P1 use is made of those conventions mentioned above, which are easy for the programmer but logically unnecessary. P1 recognises this use and translates the text into one in which those conventions are not used. This can be done before the evaluation of the text by M1 is performed and the description of M0 by splitting these two tasks is considerably simplified. Since the description of the action of P1 and M1 on the lids is much more basic it is also less easy to grasp its implications but it will settle many uncertainties left by the description of M0. Of course, this description of the action of P1 and M1 being in some language or another goes again only some way. In order to know what happens in cases which are still felt to be dubious one has to inspect the interior only to find that inside one finds again two machines, a preprocessor which translates a text into one written in a language with fewer concepts and a processor which processes this translated text. The functioning of these machines is described on the lids. It is again more primitive and it is harder to understand what it leads to but many more uncertainties are settled. Proceeding in this way one hits eventually machines which cannot be opened. Their working cannot be better explained than by the wording on the lid. If one does not understand it, that is a

pity but one cannot go further than that. It uses the most primitive notions that one presupposes to be understood without further explanation. If we now describe a language defined by such a process, we start by describing it in very general terms and refer to a more detailed description to a forthcoming publication elsewhere.

Let us first define the concept of a name. There are basic symbols in the language, just as many as one likes. Some of them are peculiar, they are called ordinators. To them belong first the brackets like ( ) [ ] ‘ ’ and moreover others like, **if then else** and maybe others. Then there are other distinguishable symbols like letters which have no inherent meaning but serve to build identifiers which may be used to denote variables. Also there are symbols which are used for very specific purposes, viz. for themselves like digits, logical values and operators. Which ones exist in the language is left to the programmer who is free to or rather has to define his own language in terms of some basic concepts which constitute the basic language. If, for instance, the programmer wants to use the symbols + and – he is free to do so but, of course, he has to define what he means. A sequence of letters and digits starting with a letter is an identifier, the simplest notation of a variable, i.e. a single entity or a linearly ordered set. The elements of such a set can be denoted by the variable followed by the number of the element in the brackets [ ]. Since the elements in themselves may again be such ordered sets one might have for example a, a[3], a[3][2]. If one does not like this notation but wants to denote the last example by a[3, 2] then this is just a matter of convention language in which we are not interested here. One has only to instruct the preprocessor to replace a[3, 2] by a[3][2] and then one can use that notation. Of course, in actually establishing a language internationally like ALGOL 60, one might wish to agree upon standard notations, but anyhow the definition of the basic machine should not be burdened by such unnecessarily complicated concepts as multiple indices in the bracket [ ].

There are also entities called constants, viz. sequences of symbols the meaning of which is defined by the sequences themselves. Constants are for instance digits, numbers, operators and strings. Open strings are sequences of strings and symbols other than ‘ and ’. A string is an open string enclosed in the bracket ‘ ’.

Again there are entities called metavariables, viz. sequences

of letters possibly followed by digits all enclosed in the meta-bracket < >, which denote sequences of none or more basic symbols.

Also there are entities called metaconstants, viz. all basic symbols except the comma, strings, metaoperators as **value**, **in** and so on.

All these entities are examples of primaries. Simple names are formed by concatenation of primaries, e.g.

x + y[2]
x := y[2]
go to L
new x
if a **then** x **else** y
3 + 4 = 7
a **in** <*letter*>

A simple name is a name. Also a name followed by a comma followed by a simple name is a name. A name enclosed in the bracket ( ) is a primary, another denotation of a set, the elements of which are the simple names which constitute the name and which are numbered from 1 onwards. Hence (x := y, **go to** L)[2] stands for **go to** L.

The fundamental concept value of a name is now introduced. This is again a name. The value of the program is determined dynamically by the machine when reading the program. In order to find out how this value is to be found the machine examines regularly a sequence V consisting of truths separated by commas, i.e. a name which is however precisely the value of the program as found so far and recorded by the machine! The examination proceeds as follows. Suppose the machine wants to determine the value of a certain name, **value** <*name* 1> say in an obvious notation. It examines the simple names, the truths, which constitute V until it finds one which is applicable, i.e. which conveys information about <*name* 1>, in order, to start with the last one. If it finds such a truth, it applies it. Generally, the problem is not solved then since in the value so obtained the operator **value** occurs again, perhaps even more than once which fact induces new evaluations until a name is obtained in which the operator **value** does no longer occur. As an example, suppose that **value** [x+y] had to be determined. By examining V the machine might find the truth

value {<*sum* 1> + <*term* 1>} = **value** {**value** <*sum* 1>
+ **value** <*term* 1>}.

In order to know whether this is applicable it would consult V to find out whether x **in** <*sum*> and y **in** <*term*> hold. It finds

x **in** <letter>,

hence it consults V in order to find out whether <*letter*> **in** <*sum*>. It finds

<*letter*> **in** <*identifier*>,

hence it consults V in order to find out whether <*identifier*> **in** <*sum*>. Suppose it finds (we assume in this example that definitions analogous to those given in ALGOL are found in V)

<*identifier*> **in** <*simple variable*>,
and again
<*simple variable*> **in** <*variable*>
and again
<*variable*> **in** <*primary*>
and again
<*primary*> **in** <*factor*>
and again
<*factor*> **in** <*term*>
and again
<*term*> **in** <*sum*>,

then it has verified that x **in** <*sum*> and the machine starts to investigate whether or not y **in** <*term*>. If it finds

y **in** <*letter*>

then we know from the history of x that it will deduce indeed that y **in** <*term*>, and hence that our truth is applicable if x is substituted for <*sum* 1> and y for <*term* 1>. Hence it has to determine **value** x by consulting V, where it finds, let us say

x = z,

which it applies by a built-in mechanism in stating

value x = **value** z

and now looking for the value of z. Suppose it finds in V

z = 3

then it knows

value x = **value** 3

and it proceeds to find the value of 3. Nothing is found which is applicable until on the very bottom of V it finds

value $<$*name* 1$>$ = $<$*name* 1$>$

It verifies that 3 is a name and hence it finds

value x = 3

Since the operator **value** no longer appears on the right hand side the evaluation of x is ended, and now **value** y has to be found. Suppose in some way or another it finds eventually

value y = 4

then it knows that

value {x+y} = **value** {3+4}

and it starts to determine the value of 3 + 4. Suppose it finds in V before it finds

value {$<$*sum* 1$>$ + $<$*term* 1$>$} = **value** {**value** $<$*sum* 1$>$ + **value** $<$*term* 1$>$}

which we know already to exist in V, the truth

3 + 4 = 7

then the same mechanism yields

value {x+y} = **value** {7}

which is in the same way as above leads to

value {x+y} = 7.

One observes that the fact that V is consulted in a prescribed order prevents the occurrence of any contradictions. For instance, the truth that was found at the bottom of V, viz.

value $<$*name* 1$>$ = $<$*name* 1$>$

is not in contradiction with any of the other truths since it can only be applied when nothing else is applicable. It necessitates however that the truths are in the correct order. If, for instance 3 + 4 = 7 would be lower down in V than

value {$<$*sum* 1$>$ + $<$*term* 1$>$} = **value** {**value** $<$*sum* 1$>$ + **value** $<$*term* 1$>$}

then the process mentioned above would never end! If on the other hand no information concerning y could be found in V then the machine would have found

value {x+y} = 3 + y.

One sees that the action of the machine is determined by its built-in mechanism and further by the truths it finds in V. Some truths are in V to start with. This list starts as follows:

value < *name* 1> = < *name* 1>,
< *sequence of none or more symbols not containing ' or '*> **in**
  < *proper string*>,
< *proper string*> **in** < *open string*>,
'< *open string*>' **in** < *open string*>,
< *open string*> < *open string*> **in** < *open string*>,
value {'< *open string* 1>'} = < *open string* 1>,
value {'< *open string* 1>', < *sequence of symbols* 1>}
  = {< *open string* 1>,
  value < *sequence of symbols* 1>},

The main point is that the value of a string is the stripped string and that the value of a sequence of simple names separated by commas is the corresponding sequence of the values of these simple names separated by commas. However, one does not know yet what a simple name is and therefore this last rule has been specialised to the case that one has a string followed by a comma, and followed again by some sequence of symbols. This is sufficient for our purpose since now the computer can read the program and add its value to the text already existing in V. If the program starts with all the additional rules that one wants to have in the language, i.e. also for instance the rules governing arithmetic, all separated by commas and all enclosed in the bracket ' ' followed by a comma and followed again by other information, then the first result will be that the set of language rules will be added to the few that existed already in V. In that way the language is fully available for the following part of the information, which is presumably more the *ad hoc* part of the program, but of course may also contain new language rules. In this way, moreover, most of the task of the precise definition of the language is left to the programmer. Of course, some suggestions can be made, how rules can be chosen in such a way that an elegant language is obtained. A piece could run as follows:

value {< *simple name* 1>, < *sequence of symbols* 1>}
  = {**value** < *simple name* 1>, **value** < *sequence of symbols* 1>},

if {<*name* 1> = <*name* 2>} **in** V **then value** <*name* 1>
= **value** <*name* 2>,
value {<*variable* 1> := <*expression* 1>} = {<*variable* 1>
= **value** <*expression* 1>},
if {<*variable* 1> = <*variable* 2>} **in** V **then value** {<*variable* 1>
:= <*expression* 1>} = **value** {<*variable* 2>
:= <*expression* 1>}

The third and fourth simple name in this segment define the main part of what is called in ALGOL the meaning of the assignment statement, the procedure declaration without parameters, the procedure statement without parameters and the formal actual substitution.

Let us first consider a simple name like s := x + y. Its value is, if we assume that nowhere in V a truth of the form s = t appears, according to the segment of V above s = **value** (x+y) which may give rise to s = 7 in V. If one wants to express what in ALGOL would be expressed by

real procedure s; s := x + y

then s := 'x + y' does the job. Indeed this gives rise to s = x + y in V. The procedure concept, at least without parameters, is therefore no longer needed. A name replacement like in the substitution of actual parameters for formal parameters in ALGOL, is simply done by s := 't'. This gives rise to s = t in V. Suppose this actually appears in V. Then the value of s := x + y will be, according to the last rule **value** {t := x + y}, which will give rise to t = 7 in V. One sees that the substitution is not actually performed but that just a note is left in V which will yield the desired result. Also in the case that the value of the actual name is not defined but required the scheme works. For instance the value of x := s would be x = **value** s which gives x = **value** t according to the second rule of the segment.

Before investigating how the parameters ot a procedure are dealt with the concept of locality will be introduced. In ALGOL a declaration serves three purposes: it introduces an identifer which is local to a block, it restricts the use of that identifier to a particular class of entities, e.g. **real** a, **array** A[1:n], or again it can completely define the meaning of an identifier as in procedure declarations and switch declarations. We have seen already that this last function is superfluous, but the concept of locality is useful. We shall not deal here with the concept own. The concept block in ALGOL as a sequence of statements, preceded by

declarations and embraced in the bracket **begin end** is too special for our purpose since even the concept statement does not exist here. Hence we shall define that inside the bracket ( ) local identifiers can be introduced by the simple name **new** < *identifier*> with the following meaning. The value of an opening parenthesis (is in V $\{(<$ *integer*>,$\}$. The integer is determined by consulting V. If no simple name of the same form is found, it is 1, else it is one more than the integer found in that simple name.
The value of **new** < *identifier* 1> is **new** < *identifier* 1> ⍟ < *integer* 1> where < *integer* 1> is the integer found following (by consulting V. Here ⍟ is a letter, which is chosen as one which is not likely to be used by the programmer normally. Since it is a letter, however, the sequence < *identifier* 1> ⍟ < *integer* 1> is again an identifier. The evaluation of a name is now redefined to the extent that the evaluation of a variable, the identifier of which, < *identifier* 1>, does not end in ⍟ < *integer*>, causes first the identifier to be extended with such an ending. The extended identifier is found by consulting V in the simple name of the form **new** < *identifier* 1> ⍟ < *integer* 1>. The value of the closing parenthesis) is defined as follows. V is consulted until (< *integer* 1> is found. The simple names in V are now scanned in the advancing order. If in a simple name < *identifier*> ⍟ < *integer* 1> is found not inside a string then that simple name together with a separating comma is deleted. If this is not the case but if the simple name is a relation like x ⍟ 14 = 3 then a copy is inserted together with a separating comma directly after the last comma which precedes (< *integer* 1> in V after which it is itself deleted together with a separating comma. After this process (< *integer* 1> in V is replaced by (and the closing parenthesis) is added to V. If in this way V would end with ( ) then these symbols are deleted.
This seemingly long definition of the value of the pair of parentheses has quite a lot of useful consequences. Let us first consider the concept function designator as it occurs in ALGOL. It assumes the existence of a procedure declaration in the body of which there occurs an assignment to the procedure identifier. A simple example is for instance given by the ALGOL declaration

```
real procedure P; begin real s; s := if x > 0 then y else z;
   P := s × (s+1); i := i+1 end
```

where the procedure P uses the nonlocal variables x, y, z and i and has a side effect on i.
This would run in our new language

```
new P , . . . , P := '(new s, s := if x > 0 then y else z,
  s × (s+1), i := i+1)',
        . . . , u := 3 × P.
```

where . . . , may either stand for empty or for a sequence of simple names separated by commas and opening parentheses. One may very well change the meaning of P by another assignment, since there is no logical distinction between the assignment above and for instance P := 3.14. "Assignment to the procedure identifier" is not necessary since the value of P is automatically delivered by the process described, enclosed in parentheses. The arithmetic in 3 × P does away with the parentheses. However, this value should probably occur only once between the parentheses since otherwise a set would be delivered like e.g. (3, 4, 2). In itself such a set may of course quite well be the value of P, but it is not the same as the last value (2). The following example has no counterpart in ALGOL.

```
P := '(10, n := n+1, if n > 20 then P := 25)'
```

This has the effect that the value of P in an expression will be 10 until n has surpassed 20, from where onwards P has the value 25, but no counting and testing will be done anymore!

At last we shall show that the parameters in a procedure can just be dealt with by the following two simple preprocessing rules.

Replace

< *identifier* 1> (< *name* 1>) := '(< *name* 2>)'

by

< *identifier* 1> := '(< *name* 3>)'

where < *name* 3> is found by replacing in < *name* 2> each simple name which is identical with (< *name* 1>) [< *integer* 1>] by ⍵ < *identifier* 1> [< *integer* 1>].

Replace in any other occurrence

< *identifier* 1> (< *name* 1>)

by

(**new** ℘ < *identifier* 1 >, ℘ < *identifier* 1 > := (< *name* 1 >), < *identifier* 1 >).

This process is perhaps best illustrated by an example.

P(u, v, 2) := '(< *text containing the identifiers* u, v, w >)'

will be replaced by

P := '(< *text containing the identifiers* ℘ P[1], ℘ P[2], ℘ P[3] *instead of* u, v, w >)'.

The "function designator" or "procedure statement"

P(a, 'b', c[i, 'j'])

will be replaced by

(**new** ℘ P, ℘ P := (a, 'b', c[i]['j']), P).

Effectively, those actual parameters which are not in quotes are called by value, those which are in quotes are called by name, whereas even mixed cases can occur as is shown in this example. This way of dealing with the call by name value-concept, viz. taking the decision on the call side rather than on the declaration side like in ALGOL has great advantages. Often it is only clear on the call side whether or not it is necessary or advantageous to call by name but also one declaration will now do for many different uses.

As a matter of fact, in the presentation given here, some things have not been defined, although explicitly used under the assumption that either the reader would be willing to accept that a proper definition could be given and would give intuitively the desired interpretation or that he would not recognise the difficulty. It may seem clear, however, that an elegant, flexible, and powerful language can be defined with a great rigour using the techniques described above.

A FAMILY OF SYMBOLIC INPUT LANGUAGES AND AN ALGOL COMPILER

Sigeiti Moriguti

University of Tokyo, Japan

PART I. THE SIP FAMILY

1. Introduction

Japanese manufacturers of electronic computers joined forces in developing a family of coordinated input languages for their computers. While each computer naturally requires its own input language, they tried to keep the fundamental structure the same, so that both the general education in programming and the conversion from one computer to another would be easy and smooth. The family is called the SIP family. SIP stands for "Symbolic Input Program".

The activity started in 1959 under the sponsorship of Japan Electronic Industry Development Association (JEIDA) and is still going on. The writer has been responsible for setting the course and guiding the working group on the task. Special mention is in order here of the name of Prof. Hidetosi Takahasi, who contributed essentially in designing the framework of the language in the present form.

2. General outlook of the language

Figure 1 shows a program written in a SIP language.

This program will produce a table of arcsin x for $x = -0.95$ step 0.05 until 0.95. "SIP100." is the heading which indicates the beginning of a new program and serves as the identification of the language used. (SIP100 is a common educational SIP language for a certain type of machines.) "L0(200)" instructs the SIP system to store the following information into storage location 200 and thereafter. The block starting with PP10) is the square root subroutine; the next block starting with PP11) is

```
      SIP100.
      L0(200)

PP10)JZ  /PP1.
      T  /A.
      XA /C.
 PP2)T  /R.
      XA /A.
      LMD/R.
      DIV/~.
      B  /R.
      T  /W.
      LMD/W.
      XMA/C+1.
      JP /PP3.
      A  /R.
      J  /PP2.
 PP3)A  /R.
 PP1)J  /I1.
  C) +99999 99999.
     +.5.
  A) WS.
  R) WS.
  W) WS.
  CANCEL/A/C/R/
  W/PP1-3.

PP11)TI1/I.
      T  /W.
      LMD/W.
      XA /A+7.
      I1S/6.
 PP1)T  /W.
      XMA/W.
      A  /A+I1.
      I1N/PP1.
      LI1/I.
      J  /I1.
   W) WS.
   I) WS.
   A) +15707 96305.
      - 2145 98802.
      +  889 78987.
      -  501 74305.
      +  308 91881.
      -  170 88126.
      +   66 70090.
      -   12 62491.
    CANCEL/A/I/
    W/PP1.

PP12)XA  /X.
      JP  /PP1.
      OS  /-.
      XAO/X.
      J   /PP2.
 PP1)OS  /SPC.
 PP2)SL  /1.
      OS  /0.
      OS  /PRD.
      ON  /2.
      OS  /SPC.
      OS  /SPC.
      XA  /X.
      JP  /PP3.
      OS  /-.
      J   /PP4.
 PP3)OS  /SPC.
 PP4)XA  /W.
      ON  /1.
      OS  /PRD.
      ON  /5.
      OS  /SPC.
      ON  /4.
      OS  /CRLF.
      J   /I1.
   CANCEL/PP1-4.

PP20)A   /X+2
      T  /X.
PP21)XBO/X.
      JZ  /PP22.
      A   /C.
      SL  /1.
      I1J /PP10.
      T   /W.
      XAO/X.
      SL  /1.
      I1J /PP11.
      T   /W+1.
      LMD/W+1.
      XA  /C+1.
      MB  /W.
      R   /~.
      T   /W.
PP23)I1J /PP12.
      XA  /X.
      B   /X+1.
      JM  /PP20.
      HJ  /PP0.
PP22)XT  /W.
      J   /PP23.
   C) +.1.
      +15707 96327.
   X) -.095.
      +.095.
      +.1.
   W) WS.
      WS.

   START(PP21)
```

FIG. 1. A program written in SIP100

the subroutine for computing a polynomial of 7th degree; the third block starting with PP12) is the print subroutine. The last block starting with PP20) is the main routine. "START(PP21)" instructs the system to finish the assembly and start the program at the point designated as PP21.

3. Control instructions

Figure 2 shows a set of control instructions, which are common to almost all SIP languages.

These control instructions do not represent orders to be stored into the memory, but rather instructions to the SIP system, effective only in the input phase.

CANCEL instruction is used when one wants to cancel the definition of local symbolic addresses and/or local program points. Each undefined symbolic address (or program point) is replaced by its value as soon as the definition is given. After the definition is given, the replacement occurs as soon as it is read in. The values thus replaced are not affected by the later cancellation of the definition.

HALT. is useful when more than one input tapes are used. The machine stops right after this control instruction is read, and one can replace the input tape with another before pressing the RESTART button.

Besides those control instructions in Fig. 2, some SIP's have CHECK. and STŌP/CHECK. The former instructs the system to set up a linkage there to the dynamic check subroutine which will print out the content of the accumulator at the run time. The latter nullifies the effect of CHECK. There can be more than one CHECK.'s in a program, and all these are nullified by a single STŌP/CHECK.

If the machine has a pair of orders in each storage location, then its SIP usually has a control instruction LEFT., which forces the following order to be the left-hand order, putting if necessary a non-effective order NE/~. into the otherwise vacant space. Likewise, there is a control instruction EVEN. with the

SIP100. etc.	Heading of a new program.
L0(. . .)	Store instruction, designating the initial location L0.
START(. . .)	Start instruction, designating the starting point.
WS.	Reservation of a working space.
PP1) etc.	Definition of PP(program point) No. 1 etc.
A) etc.	Definition of symbolic address A etc.
A(. . .) etc.	Another form of definition of symbolic address A etc.
CANCEL/A/PP1. etc.	Cancellation of definition of symbolic addresses and/or program points.
HALT.	Instructs the system to stop and wait.

FIG. 2. Control instructions of SIP.

machine which combines an even address with the next odd address to form a double-length storage location.

4. Operation part

An order consists of an operation part (a function part) and an address part. The two parts are separated by a /(slant). The entire order ends with a period.

The operation part is usually a string of alphabets. Very rarely, some numerals come in. The length of the string is usually not greater than three characters.

Figures 3 and 4 show the orders of SIP100 and their meaning. X is always read "clear". A means "add", B "sub(– tract)". The last Ō means "absolute". MA means "multiply and add". The leading L is always "load", T "store", J "jump", I "index". It is very rare that we do not know how to read the first letter before we see the rest of the operation part. This is a feature of SIP family which accounts for easy learning.*

Most of the operation codes in Figs. 3 and 4 are common to many members of the SIP family. The same function is always represented by the same code. Essentially different functions are represented by different codes. A typical example of this statement is the "divide" order. SIP100 has "DIV/~." which means dividing the content of the accumulator by the content of the MD register. This is a compromise between the two machines on both of which the original SIP100 was designed to run. SIP101 for NEAC-2203 (a Nippon Electric machine) has the divide order "DBY/n." which means dividing the content of the accumulator *by* the content of storage location n; the quotient goes into MQ register. SIP102 for HITAC-301 (a Hitachi machine) has "XAD/n.", "AD/n.", "XBD/n." and "BD/n." as the (composite) divide orders. Here, either XA or A or XB or B is carried out first (with n as the pertinent address) and then the result is divided by the content of MD register; the quotient goes into the upper accumulator.

Another example is the conditional jump order related to the index registers. SIP-T for TŌSBAC-3121 (a Toshiba machine under construction) has the conditional jump orders "IiE/Ij/n.",

*Sekine (Keiō University) has communicated personally to the writer that, according to his experiment with students, the learning curve of the SIP operation part goes right up from the beginning whereas for a three-letter code it rises very slowly at first.

Order	Operation Part stands for:	Meaning
XA /n.	clear add	(n) → Acc
A /n.	add	(Acc) + (n) → Acc
XB /n.	clear sub	−(n) → Acc
B /n.	sub	(Acc) − (n) → Acc
XA$\overline{O}$/n.	clear add absolute	| (n) | → Acc
A$\overline{O}$ /n.	add absolute	(Acc) + | (n) | → Acc
XB$\overline{O}$/n.	clear sub absolute	−| (n) | → Acc
B$\overline{O}$ /n.	sub absolute	(Acc) − | (n) | → Acc
XMA/n.	clear mult add	(MD) × (n) → Acc
MA /n.	mult add	(Acc) + (MD) × (n) → Acc
XMB/n.	clear mult sub	−(MD) × (n) → Acc
MB /n.	mult sub	(Acc) − (MD) × (n) → Acc
DIV /∼.	divide	(Acc)*/(MD) → Acc [remainder → REM]
LMD/n.	load MD	(n) → MD
T /n.	store	(Acc) → n
XT /n.	clear store	"0" → Acc ; "0" → n
R /∼.	round	(Acc)* rounded to 10 decimal places
SL /n.	shift left	(Acc)* × 10^{n} → Acc
SR /n.	shift right	(Acc)* × 10^{-n} → Acc
HJ /n.	halt jump	halt; restart at n
J /n.	jump	go to n
JP /n.	jump plus	if (Acc) ≥ 0 then go to n
JM /n.	jump minus	if (Acc) < 0 then go to n
JZ /n.	jump zero	if (Acc) = 0 then go to n
I1S /n.	index 1 set	"n" → IR1
I1J /n.	index 1 jump	(SCC) → IR1 ; go to n
I1Z /n.	index 1 zero	if (IR1) = 0 then go to n; (IR1) − 1 → IR1
I1N /n.	index 1 non-zero	if (IR1) ≠ 0 then begin (IR1) − 1 → IR1; go to n end
TI1 /n.	store index 1	(IR1) → n
LI1 /n.	load index 1	(n) → IR1

FIG. 3. Orders of SIP100.

"IiN/Ij/n.", IiH/Ij/n.", "IiL/Ij/n.". The first one causes the jump if (IRi) = (IRj). Likewise with the others (≠, >, < replacing = in the above statement).

Floating point arithmetic operations are represented by FA, FB, etc. with F attached in front. With some machines, these operations are carried out by the subroutines, in which case the SIP system generates the linkage rather than the machine order.

Order	Operation part stands for:	Meaning
SEL/□.	select	select input/output equipment □.
XRN/n.	clear read numeric	n digits on tape → Acc
$\overline{O}$N /n.	out numeric	n digits in Acc → output
$\overline{O}$S /□.	out special	special mark □ → output
XRH/n.	clear read character	n characters on tape → Acc
$\overline{O}$H /n.	out character	n characters in Acc → output
XEA/n.	clear extract add	(MD)* ⊗ (n) → Acc
EA /n.	extract add	(Acc) + (MD)* ⊗ (n) → Acc

In the table, (. . .) means "the content of . . ."; Acc = accumulator; MD = multiplicand-divisor register; (Acc)* = long accumulator (double length); IR1 = index register No. 1; SCC = sequence control counter; (MD)* ⊗ means the extraction of those digits which have 1's in the corresponding positions in the MD register; ~ means empty; □ means something special.

FIG. 4. Orders of SIP100 (continued).

This kind of coordination minimizes the loss and confusion accompanying the switch from one computer to another. In particular, those who program for more than one machine at the same time profit the most from this coordination.

5. Address part

The address part can be the absolute address such as 200, 1358, etc. Symbolic address A, B, X, etc. are also allowed. Jump orders usually have program points in their address part (e.g. PP2, PP23). Such expressions as A + 7, PP12 + 3 are also allowed. Modification by the content of index register 1 is represented by +I1 attached to the address part.

The address part of the select order (SEL) can be one of the following 6: KB (keyboard), MTR (mechanical tape reader), PTR (photoelectric tape reader), TYP (typewriter), PERF (perforator), B$\overline{O}$TH (both typewriter and perforator).

THE out-special order ($\overline{O}$S) can have in its address part either a digit, or an alphabet, or a special character such as + − ( ) etc., or PRD (period), C$\overline{O}$M (comma), SLT (slant), SPC (space), CRLF (carriage return and line feed).

For those machines which can address the accumulator, the

Name	Machine (maker)	status
SIP100	common to NEAC-2203 and HITAC-301	extensively used for education since 1959
SIP101	NEAC-2203 (Nippon Electric Co.)	in use since 1959
SIP102	HITAC-301 (Hitachi, Ltd.)	in use since 1959
HISIP101B	HIPAC-101B (Hitachi, Ltd.)	extensively used since 1960
ŌKISIP	ŌKITAC 5090 (Oki Electric Ind. Co.)	just completed
HISIP103	HIPAC 103 (Hitachi, Ltd.)	just completed
SIP-T	TŌSBAC-3121 (Tokyo Shibaura Electric Co.)	in preparation
SIP300	common to MELCŌM and MADIC	just completed
SIP301	MELCŌM 1101 (Mitsubishi Electric Mfg. Co.)	in preparation
SIP302	MADIC II (Matsushita Comm. Ind. Co.)	in preparation
---	FACŌM 222 (Fuji Comm. App. Mfg. Co.)	being considered
SIP1000?	common to all Japanese computers (educational language)	being considered

FIG. 5. Members of the SIP family.

address part can be AC (= accumulator). SIP102 for HITAC301 allows MD as the address part of XA, A, XB, and B.

In HISIP 101B for HIPAC 101B (a Hitachi parametron computer), A//n. means adding the content of short location n, whereas A/n.—with even n—means adding the content of long location (n, n+1). Likewise T//n. means storing the first 21 bits of the accumulator into location n. whereas T/n. means storing the 42 bits in the upper accumulator into location (n, n+1).

6. State of the art

The present situation with the SIP family is summarized in Fig. 5.

All major computer makers in Japan are represented in the table. SIP100 has been publicized by JEIDA and several university professors. Its use in the first course in programming has been most successful. SIP300 is a newly developed common educational language for the two new computers MELCOM and MADIC. SIP100 was too much oriented towards the particular machines available at the time of its development and consequently it was necessary to develop a new "common" language for those new machines. SIP1000(?) is under serious consideration as a more widely applicable common educational language.

Other members are the languages specifically devised for the particular machines. These are more "natural" than the educational languages, and, in principle, each order has one-to-one correspondence with the "machine word".

The use of a SIP language as a practical programming language is not yet universal. There is some reluctance among "professional" programmers because of the theoretical inefficiency in preparation of program tapes ("Numerical codes are easier to punch"). Also it is true that in some cases existing SIP system (the assembler) does have certain drawbacks which displease the expert programmers. These can be and should be amended, and the writer believes the merits will outweigh the demerits in the long run, provided the coordinated efforts towards the common goal are continued at the present pace. The experience with HISIP 101B at the Institute of Japanese Union of Scientists and Engineers has been quite encouraging. The Computation Center of the University of Tokyo has decided the policy that it will use OKISIP and no other language as the input language to its OKITAC's. (ALGOL will also be used after the processor is completed.) The experience to be gained in this Center will be very helpful in the further development of the SIP family.

The system program occupies about 1000 storage locations in the memory of capacity 2000—4000. One reason for the relatively large size of the system program is the extensive check made on the incoming program. Almost all syntactical errors are found out during the input process. This slows down the speed of input considerably, but it pays off in terms of freedom from minor errors which might otherwise require expensive debugging. If the storage requirement is heavy, one can overwrite the system program by the intermediate results, defining the symbolic addresses for the working space in such a way as

name	author(s)	machine	type	status
JUSE ALGŌL	Moriguti	HIPAC 101	ALGŌL	debugging
~	Nagao	KDC-1	"	"
NALGŌ	Takenaka	NEAC-2203	"	"
~	Inoue	FACŌM 202	"	coding
~	Waseda Univ.	TŌSBAC 3121	"	"
~	Imai, Kodaira	MADIC-II A	"	flow-charting
~	Shimizu	ŌKITAC 5090	"	"
TALGŌL-I	Miura	USSC 90 with tape	"	"
2203 NARC	Furuyama	NEAC-2203	FŌRTRAN	in use
HARP 103	Hitachi Cent. Lab.	HIPAC 103	"	debugging
MUSE	Mitsubishi El. Co.	MELCŌM 1101F	"	"
EASE	Tamura, Kidera, Maseki	NEAC-2203	"	"
~	Fuji Comm. Co.	FACŌM 222	"	coding
M1 AUTŌCŌDE 1	Ikeno	MUSASINŌ-1	~	in use
HARP 101	Simauti, Kondō	HIPAC 101	~	debugging
~	Simauti, Urabe	HIPAC 101	~	coding

FIG. 6. ALGŌL/FŌRTRAN type compilers of Japan

A(3500), where the absolute address 3500 is in the area occupied by the system program.

PART II. THE JUSE ALGŌL

7. Interests in ALGŌL/FŌRTRAN

In recent years in Japan, a great pressure is felt for compilers of higher intelligence level. Figure 6 shows several examples of the endeavor to meet the pressure.

Japan Electronic Industry Development Association has organized Software Committee, under which there is a Subcommittee on ALGŌL/FŌRTRAN. Information Processing Society of Japan has a Research Committee on ALGŌL. There are also several other groups seriously interested in ALGŌL and/or FŌRTRAN.

8. JUSE ALGŌL

The writer is engaged in the development of an ALGŌL compiler for HIPAC 101, a binary parametron computer with a drum memory of 4096 words (1 word consisting of 21 bits). JUSE stands for the Japanese Union of Scientists and Engineers, which owns a HIPAC 101.

The restrictions and a little extension to ALGŌL 60 are as follows [the section numbers refer to those of the ALGŌL 60 report (1)]:

2.1. LETTERS

<letter>::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

2.2.2. LOGICAL VALUES

No logical values will be allowed

2.3. DELIMITERS

- <arithmetic operator>::= + | − | * | / | div | ↑
- <logical operator>::= or | & ((≡, ⊃, ¬ not allowed))
- *while*, *Boolean* not used
- Boldface letters are replaced by middle case (small) letters.
- Spaces will be delimiters between basic symbols and/or identifiers.

2.4. IDENTIFIERS

- If more than 6 characters are used, the first 6 characters must be different for different identifiers.

2.6. STRINGS
. left string quote ' is replaced by''.
3.2.4. Standard functions
. abs, sign, etc. are expressed as ABS, SIGN, etc.
3.3. ARITHMETIC EXPRESSIONS
. The expression A ↑ B is allowed only when B is a non-negative integer.
. The two operands of +, −, * must be of the same type.
. The two operands of / must be of type real.
The two operands of div must be of type integer.
. No implicit conversion is allowed.
3.4. BOOLEAN EXPRESSIONS
. Variables cannot be of type *Boolean.*
3.5. DESIGNATIONAL EXPRESSIONS
. Unsigned integers may not be used as labels. ((It is suggested that 217, e.g., be replaced by L217.))
4.2. ASSIGNMENT STATEMENTS
. No Boolean assignment statements will be allowed.
4.3.5. GO TO an undefined switch designator
. A *go to* statement is undefined if the designational expression is a switch designator whose value has not been defined.
4.6. *for* STATEMENTS
. The "step" value must be of the form "<primary>" or "-<primary>" where the <primary> must have positive value while the program is running.
. The "*for* list" may consist either of a single "step-until" element or of a sequence of constants separated by commas.
4.7. PROCEDURE STATEMENTS
. No procedure may be called before it is declared.
. Recursive calls on procedures are not permitted.
. If a procedure P is an actual parameter to some procedure Q, all of the parameters of P must be called by *name.*
5. *Declarations*
5.2. ARRAY DECLARATIONS
. The upper and lower bound expressions must be integer "constants". Here, the term "constant" is used in an extended sense:---
A constant is either a number or a simple variable

declared *constant*. *constant* declaration is of the form

constant < simple variable list>

where < simple variable list>: : = < simple variable> | < simple variable>, < simple variable list> Those simple variables must be the last simple variables type-declared, and the constant declaration must immediately follow the type declaration. A simple variable declared "constant" must be assigned a value once and only once. This assignment statement must be such that the value of the expression on the right-hand side is fixed at the time of compilation. Operations on "constants" are carried out at the time of compilation as far as possible; those operations are not coded in the object program, but rather the result is entered in the "constant table" and its address is referred to when necessary.

5.3. SWITCH DECLARATIONS

. The elements of a switch list may only be labels.

5.4. PROCEDURE DECLARATIONS

. Procedures may not be of type *Boolean*.
. No formal parameter may be of type *Boolean*.
. If a procedure is to define a value, it must not store quantities into nonlocal variables. In this sense, no "side effects" are allowed for such a procedure.
. No parameter specified as an array may be called by value.
. Procedure declarations must follow all type, switch, and array declarations of the block in which they occur.
. The procedure identifier may only occur inside the procedure body as the left part of an assignment statement.
. The following procedures can be used without declaration:

READREAL (< real variables>)
READINTEGER (< integer variable>)
PRINTREAL (< real arithmetic expression>)
PRINTINTEGER (< integer arithmetic expression>)
PRINTSTRING ("< open string>')
CRLF --- carriage return and line feed
RANDOM --- real procedure generating a random number between 0 and 1.

Besides these, the standard functions

SQRT (E), ABS (E), SIN (E), COS (E), ARCTAN (E), LN (E), EXP (E), SIGN (E),

and the transfer functions

FLŌAT(E), FIX(E), ENTIER(E)
can also be used without declaration.

The philosophy and the way of presentation are very much like the SMALGŌL (2). The only major differences are:

(a) No implicit conversion is allowed. Any conversion must be stated explicitly by using one of the transfer functions FLŌAT, FIX, and ENTIER.

(b) The "*for* list" can be a sequence of constants separated by commas.

(c) If a procedure P is an actual parameter to some procedure Q, all of the parameters of P must be called by *name* (rather than by *value* as in SMALGŌL).

(d) The meaning of a "constant" has been somewhat extended as stated in 5.2. above. This extension corresponds to the well-known notion of a *preset parameter*. It avoids the dynamic allocation of the storage locations for arrays, and still retains the generality of library routines.

9. Particulars of the compiler

The principle of the JUSE ALGŌL compiler is that of sequential translation (3). A great influence came from (4).

Figure 7 shows the main part of the matrix which tells what to do when a new symbol γ is placed against the state symbol σ_S at the "top" of Σ-stack (state-symbol stack).

In the table, $\gamma \rightarrow \Sigma_+$ means that the new symbol γ is to be stored into Σ-stack by pushing down all the existing symbols there; $\gamma \rightarrow \Sigma_0$ means that γ is to be stored at the top of Σ-stack, replacing the state symbol there; Σ_- means the popping-up of the Σ-stack. Id stands for identifier. ω_{arith}, ω_{rel}, and $\omega_{\log}$ the generic symbols for arithmetic, relational, and logical operators, respectively. I is a special state symbol indicating that an identifier (or an equivalent) has been encountered. E is another special state symbol that an expression is expected. E is useful in distinguishing unary operators + and – from their binary counterparts. rep. means to repeat, i.e. to examine the same symbol γ against the new state symbol uncovered by the operation Σ-. next means to read in and examine the next symbol γ. π means that a piece of object program is to be generated. α means that the location of the identifier γ is to be stored into the A-stack (the operand stack).

All the blank spaces are understood to indicate the error.

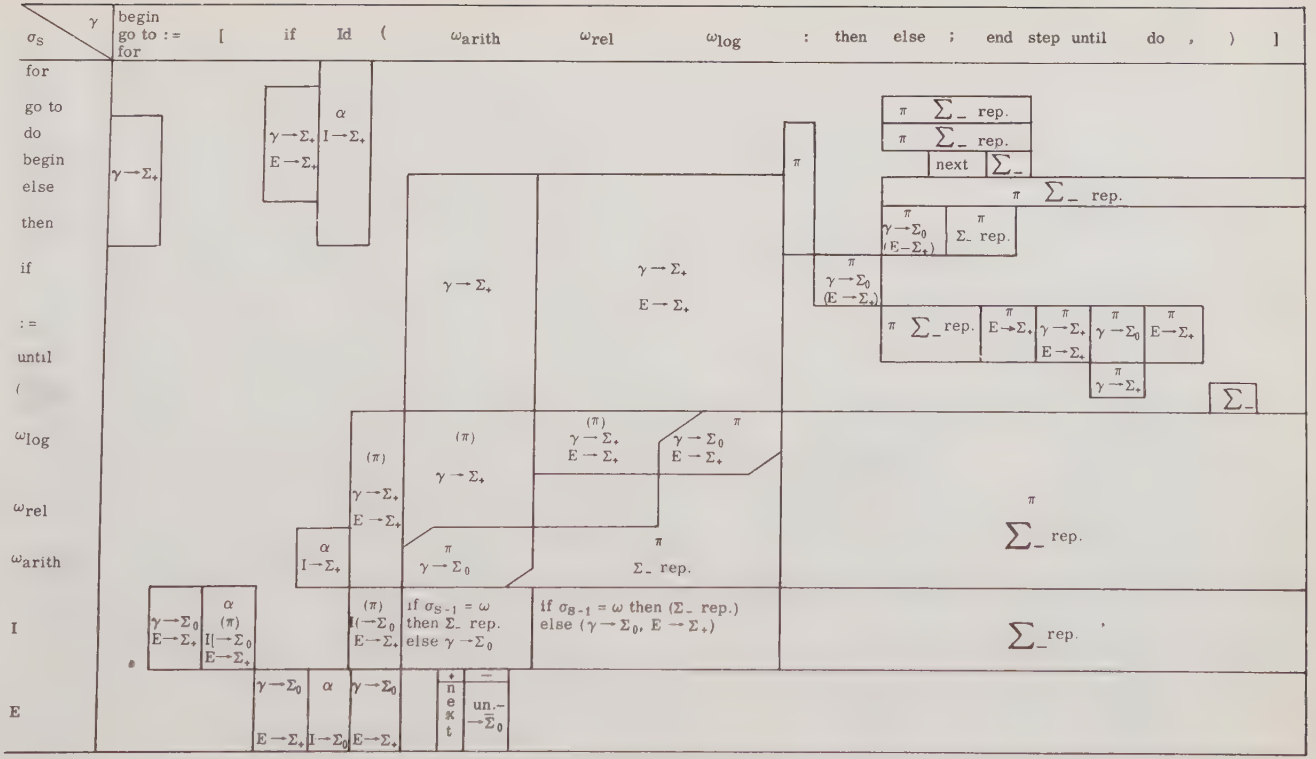


FIG. 7. The matrix for the JUSE ALGOL compiler.

Algorithms Nos. 8—72 (Comm. ACM, vol. 3, no. 5 ~ vol. 4, no. 11)

string	0
label	0
while	1
own	2
¬	2
switch	5
∨	7
true	9
false	9
Boolean	10
÷	12
value	47
array	85
else	87
procedure	94
comment	115
integer	130
real	159
go to	170
if	232
then	232
step	236
until	236
for	236
do	236
begin	294
end	294

FIG. 8. Frequency distribution of basic symbols.

Identification of the incoming symbol γ is carried out by a programmed table-look-up operation. It is somewhat expedited by arranging the symbols in the table in the order of Figure 8, which shows the frequency distribution of basic symbols (plus some extras) obtained from Algorithms 8—72 of Comm. ACM.

During the compilation, two identifier tables are maintained dynamically:—one for simple (not *own*) variables, the other for all other identifiers. Both have the block structure as prescribed in 4.1.3. of (1). Actual form and the ways of manipulation of the V-table (simple variable table) is illustrated in Fig. 9. Each identifier in this table occupies 2 storage locations. The heading of each block consists of two entries:— (i) the address of the first identifier in the preceding block, (ii) twice the number of identifiers in the preceding block. The v-counter maintains the

address of the first identifier in the current block and twice the number of identifiers already stored in the current block. The operations illustrated on Fig. 9 will automatically insure the desired block structure of the table. The table in some sense grows and contracts as the compilation proceeds and contracts to the initial heading (0, 0) at the end of the whole program.

Another table—C-table (complex identifier table)—contains all the other identifiers than those in V-table. The form and ways of manipulation are almost the same as those of V-table. The only differences are:—(a) An identifier in C-table occupies a pair of storage locations, and the related information is stored in the adjacent pair of locations, thus 4 locations in total are consumed per identifier. Accordingly, the block heading consists of a pair of locations just like the heading of V-table and

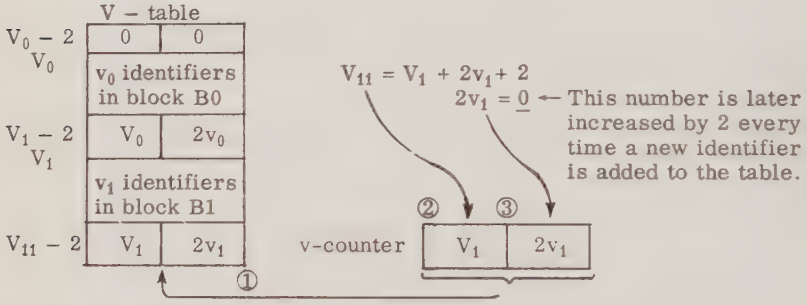


FIG. 9. Block structure of the simple variable table.

FIG. 9.1. Operations at BB (block begin).

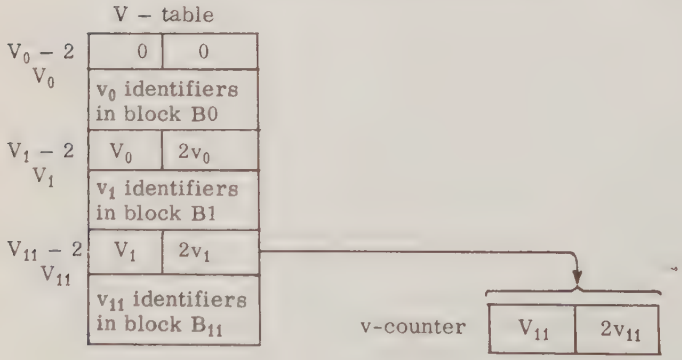


FIG. 9.2. Operation at BE (block end).

another pair of locations containing all zeros. (b) The manipulation at the end of a block is somewhat more complex. That is, besides storing the content of the current block heading into the c-counter (corresponding to the v-counter in Fig. 9.2), each undefined label in the current block must be checked with the labels in the preceding block. If there exists an identical label already defined, then that definition should be carried over. Otherwise, the undefined label must be planted anew into the extension of the preceding block (which has now become the current block by virtue of the BE operation as described in Fig. 9.2).

The compiler occupies about 3000 storage locations of 21 bits each. The object program is punched out on paper tape in SIP language (see Part I).

10. Acknowledgment

The writer is indebted to many friends for the ideas put into the works reported here. In particular, discussions with Prof. Hidetosi Takahasi, Messrs. Tomesaburo Shimizu and Kenzō Inoue have been invaluable. Thanks are also due to Mr. Kenichi Koyanagi of Japanese Union of Scientists and Engineers for the sponsorship of the work and to Miss Eiko Miyazaki for her help in debugging.

REFERENCES

1. P. Naur (ed.), "Report on the algorithmic language ALGŌL 60", Comm. ACM, vol. 3 (1960), pp. 299-314.
2. G. A. Bachelor et al. (ed.), "SMALGŌL - 61", Comm. ACM, vol. 4 (1961), pp. 499-502.
3. K. Samelson and F. L. Bauer, "Sequential formula translation", Comm. ACM, vol. 3 (1960), pp. 76-83.
4. A. A. Grau, "Recursive processes and ALGŌL translation", Comm. ACM, vol. 4 (1961), pp. 10-15.

PALGO: AN ALGORITHMIC LANGUAGE AND ITS TRANSLATOR FOR OLIVETTI ELEA 6001

M. Pacelli, D. Gavioli, G. Palermo, and U. Picciafuoco

Olivetti, Milano, Italy

INTRODUCTION

The PALGO project was started with the aim of defining an Algol 60(1) subset to be used as a programming language for medium size computers of the type Olivetti Elea 6001.

It was decided that the process ought to point toward the following goals:

1) to leave out those structural elements of the Algol language which would necessitate a very large compiler program.
2) To provide all data spaces and group of operations of the Algol 60.

Beside the above two major points, additional features were introduced in order to eliminate some ambiguities of Algol 60 as well as to increase the flexibility of the language itself.

These features concern both the procedure statements, for which Church's λ-notation was introduced, and the FOR statements, which were allowed to control more than one variable in the same loop.

As a result of this philosophy PALGO language should not be considered a subset of Algol 60.

For the terminology used in this paper, reference must be made to the Algol 60 Report.

1. THE PALGO LANGUAGE

Integer, real and Boolean variables are allowed. Subscripted variables are of real type and subscripted expressions can be only linear forms of integer variables and constants.

This constraint allows the compiler to use index registers in treating subscripted variables under control of FOR statements.

Simple arithmetic and Boolean expressions are accepted and the latter may use all Algol 60 operators and relational expressions.

Statements and declarations are the fundamental elements of

this language which is simplified in its structure as compared with the Algol 60, by the following conditions:

1) Program and procedures consist of only one block.
2) Compound statements are accepted only if controlled by FOR statements.

1.1 Declarations

Type declarations are the same as in Algol 60, list elements of SWITCH declarations are labels, and ARRAY declarations have the following form:

ARRAY $A_1 (a_1), \cdots, A_k (a_r) : (m_1, M_1; \cdots ; m_r, M_r)$

Where:

A_i (i = 1, 2,...,k) are Array identifiers with the same number of subscripts.

a_1 (i = 1, 2,... k) are integers and represent the maximum number of A_1 elements that are present simultaneously in the main memory.
a_1 may be suppressed if all the A_i elements are simultaneously in the main memory.

mj ,Mj (j = 1, 2,...r) are lower and upper bounds of the subscripts and must be only integer numbers.

Optimization in storage allocation is realised with the help of the declaration statement:

EQUIVALENCE "list 1" "list 2".

Written in "list 1" are the variables whose values no longer interest at the time of calling out any procedure. All the other variables that are not defined OWN are in List 2. In each list it is possible to establish a hierarchical level by the use of the parenthesis symbol for each variable. For example:

EQUIVALENCE "A,B(C(D,E)(F,G))(M,P)""X(Y(Z)(W))"

Where:

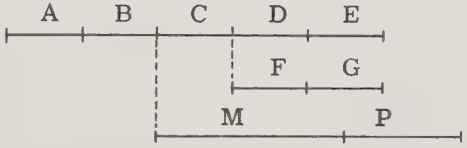
A,B	have	level	0
C	has	level	1
D,E F,G	have	level	2
M,P	have	level	1
X	has	level	0

Y	has	level	1
Z W	have	level	2

The process for storage allocation is:

a) Consecutive fields are assigned to variables with the same level
b) Fields assigned to consecutive groups of variables with the same level, have the same origin.

In the above example we have the following storage allocation for the list 1:



in which parallel fields should be considered overlapped.

Procedure statements have the following form:

PROCEDURE NNNN (input-list: output-list)

in which

NNNN is the procedure identifier

input-list is the list of the formal input parameters
output-list is the list of the formal output parameters.

The formal parameters can indicate real and Boolean variables, array and procedure identifiers. In the last case the formal identifiers must be followed by the symbols "(" ")" between which the programmer may insert any string of symbols not containing other parenthesis symbols.

The input parameters can be declared VALUE and OWN with the same meaning that these declarations have in the ALGOL 60 report.

If in the procedure body there are non-local variables, among the procedure declarations there must be the declaration statement "DEFINE" that explains the names of the procedures that define every non-local variable.

Finally, recursive procedures are allowed.

1.2 Statements

While the assignment and the GO TO statements have not been modified with respect to ALGOL 60, the IF statements have been simplified to permit only a decision based on the value of a Boolean expression whether to pass program control to the next following statements or to another statement.

The FOR statements accept only one list element and can control more than one variable in the same loop.

The proceeding type of FOR statement has the form:

FOR V_1: = L_1, V_2: = L_2,....V_K: = L_K

WHILE B*DO

Where: V_1, V_2....V_K are integer variable

L_1, L_2....L_K are elements having the following forms:

a) E_1 STEP E_2 UNTIL E_3 where E_1, E_2, E_3 are arithmetic expressions.

b) E_1 STEP E_2 WHILE B where E_1, E_2 are arithmetic expression and B is a Boolean expression.

c) E_1 WHILE B where E_1 is an arithmetic expression and B is a Boolean expression.

B* is a Boolean expression.

When an L element is of the type b) or c) and B is identical with B* then the part WHILE B can be eliminated in L.

The meaning of this FOR statement is as follows:

the statement following DO must be executed as long as B* is true varying V_i (i = 1, 2.....K) by the rules expressed in the corresponding L_i elements.

When the condition of an L_i is no longer true the corresponding V_i is frozen at the value it last had until the condition becomes true again.

The calling of a procedure is effected by a procedure statement composed of the procedure name followed by the list of the actual parameters. This implies that the formal parameters in the procedure declaration and the actual parameters of the procedure call are in a biunivoque and equiordered correspondence.

When a formal parameter is a scalar variable the identifier of that formal parameter in the procedure declaration may be used as a function identifier and the call of such a function can be executed by writing this identifier followed by the list of the actual input parameters in the procedure call.

For example, the procedure defined by the statement:

PROCEDURE ALFA (A,M : B;C,D) can be called by the statement:

ALFA (A1, M1: BETA GAMMA, DELTA) or, otherwise, by one of the following statements:

B (A1, M1)
C (A1, M1)
D (A1, M1)

which respectively permit the computation of B, C, D, in correspondence to the actual input parameter A1, M1.

This second type of call can be particularly useful when one wants to use, for example, different values of B (or C, or D) in the same arithmetic expression without interrupting the computation. This technique eliminates any side effect in arithmetic expression if we accept the following restriction:
"Non-local variables and parameters called by name cannot appear in the left side of an assignment statement."

The actual input parameters that do not correspond to procedure names can be:

1) Arithmetic expressions;
2) Boolean expressions;
3) Array identifiers.

When actual input parameter corresponds to a formal parameter that signifies a procedure identifier, it must be identified by means of the Church's λ– notation in the form:

[1] LAMBDA ((dummy list), F)

Where F is any form on variables or subscripted variables, that can contain procedure calls with undefined parameters.

The dummy list describes those variables of F which are not defined for the call under consideration.

If [1] is the actual parameter of a procedure named P, a call on P can appear among the possible procedure calls contained in F.

Let us define the following procedure statements:

PROCEDURE F1 (A, W(X,Y), Q(Z):F1)
PROCEDURE F2 (A,B,C: F2)
PROCEDURE F3 (A,B: F3)
PROCEDURE F4 (A,B,W(X): F4)

Thus it is legitimate to write a procedure call of the type:

F1 (A1, LAMDA((B,X), B+F3 (Y,X)), LAMDA ((Y),
F4 (A1,Y,LAMDA ((X),F2(A1,X,G1)+F3(X,B1))))).

The input-output statements accepted by the PALGO system operate with a list that indicates the quantities to be transferred.

Every list element can be a real variable, a subscripted variable, an expression which contains one or more subscripted variables together with a rule for subscript variation.

On such expressions one can describe by means of the parenthesis symbols "(","))" a FOR nesting, using FOR elements of type STEP-UNTIL which define precisely a sequence of subscripted variables.

2. THE PALGO TRANSLATOR

The following explanation of the PALGO translator will demonstrate some of the techniques which permit translation in only one pass, working statement by statement to produce immediately the machine language program.

The PALGO system permits operations on subscripted variables and arrays, which can be either completely contained in the memory or not, and whose subscripts are linear functions of other variables which may or may not be controlled by FOR statements.

Before explaining the treatment of arrays whose subscripts are controlled by FOR statements, it will be useful to introduce the concept of "structure" of a subscripted variable.

Let us assume an array completely contained in memory and stored column wise. Given the integer variables I_K (K = 1,2,— n) which appear in the subscript expressions of a variable, and assuming linearity, the function which defines the address of a typical subscripted variable will be linear in I_K and will have the form:

$$\mathfrak{z} = \sum_{1}^{n} k \; S_K \, I_K + S_{n+1} \qquad [1]$$

where:

I_K represents the actual value of index I_K
S_K for K = 1, 2——n, are integer constants.

The "structure" of subscripted variable with respect to the ordered set I_K is the ordered n-ple of pairs:

$S_1\ I_1\ ,\ S_2\ I_2\ \text{---}\ S_n\ I_n$

From [1] it can be seen that the address of subscripted variable is formed of two distinct parts. The first, which we will call V is given by the sum of the n terms which refer to the structure; the second part is formed of only one term which refers to the array and its location in the memory. This second part is a different constant for each given array, while the first part which is clearly a function of the actual value I_K (controlled by FOR statements) can be common to many subscripted variables.

The translator assigns to every "structure" an index register T, and creates a group of operations in the object program which have the purpose of loading T with the actual value of V at the time that operations must be performed on the corresponding subscripted variable.

On this way the address of the subscripted variable will be obtained simply by modifying T with S_{n+1}. The operation which lead to the loading of T are executed at the beginning and at the end of the loops controlled by the FOR statements associated with the variables which appear in the structure of the subscripted variable.

The calculation of the V part of [1] is obtained giving to each interested FOR statement the task of updating that component of V to which it refers. In this way the speed of the internal loops, which are performed the most times, is increased.

The assignment of registers to the structures is carried out under the following rules:

a) Precedence is given to the structures of the innermost loops.
b) The same register can assigned to structures within different loops at the same level.

The optimization of this assignment process, which is explained only in a general way, is in practice very complex and would in general require a two pass translator. For this reason the translator provides for skipping the optimization process whenever the programmer finds that the total number of different structures in a FOR nesting and their distribution will not require more than 8 index registers.

In this case the registers are assigned to the structures in the order in which they appear, and the compilation can be done in one pass.

2.1. When, at the input, appears a statement of the form:

GO TO l

where l is a label, the translator operates in one of two ways according to whether or not the label is a part of a statement previously translated.

When this is true, the translator constructs a jump instruction to the address of the first of the instructions generated by the statement having label l. When this is false (the label has not previously appeared) the translator generates a jump instruction to another jump instruction (rebound jump) which will be located in a fixed position known to the translator program.

When the input statement is of the form:

GO TO A (I)

where A is a switch designator and I is an integer variable, the translator works almost as in the previous case, with the difference that to the first jump there corresponds an array of rebound jumps which are addressed by the value of I.

2.2. The method adopted for the translation of arithmetic expressions is similar to that presented by Samelson and Bauer in (2).

The memory assignments of volatile variable is obtained using a programmed push-down list.

On the other hand, Boolean expressions are handled by a method analogous to that described by Huskey and Wattenburg in (3) generalized as needed for the introduction of Boolean operators $\supset$ and $\equiv$.

2.3. Finally we think useful to give a brief description of the processing of procedure statements with procedure identifiers as formal parameters. Defining P as the procedure identifier of the procedure statement and F the form which represents the actual parameter, the translator builds a new procedure P′ which computes the form F declaring "value" the bounds variables, and transfers to the body of the procedure P the identifier P′.

In this way when the body of P will call P′, the bound variables will have already been evaluated.

REFERENCES

1. Naur, P. et al. Report on ALGOL 60, Comm. ACM 3, May 1960.
2. Samelson, K. and Bauer, F. L. Sequential Formula Translation, Comm. ACM 3, Feb. 1960.
3. Huskey, H. D. and Watternburg, W. M. Compiling Techniques for Boolean Expressions and Conditional Statements in ALGOL 60, Comm. ACM 4, Jan. 1961.

THE ALGEBRAIC COMPILERS FOR BENDIX G. 20 COMPUTING SYSTEM

Giorgio Savastano and *Bruno Fadini*

Centro di Calcolo Elettronico
Universita di Napoli
Italy

1. THE G.20 COMPUTING SYSTEM

The Bendix G.20 general purpose computing system contains a high speed central processor and a complete set of input/output accessories, connected through a digital communication line. The system is completely modular; peripheral equipment can operate concurrently with, and independently of, the central processor.

The Central Processor is built of solid state components and has random access magnetic core memory expandable from 4096 to 32,768 words, in modules of 4096 words each.

Numerical informations are stored in the memory in one of three formats: Single Precision, Extended Precision and Integer. A Single Precision quantity is a floating-point number that is contained in a single memory location. An Extended Precision quantity is a floating-point number that is contained in two consecutive memory locations. An integer is a single precision number stored with zero exponent. Storage of numbers in the "Pickapoint" mode is provided automatically: a Pickapoint quantity is a floating-point number that is stored in a single memory location without the exponent; during storage the quantity is automatically adjusted to an exponent selected by the programmer. All internal computation is performed in Extended Precision floating-point form.

The memory word contains 33 bits including a parity bit. A single precision quantity consists of 7 octal digits for the mantissa and 2 octal digits for the exponent. An extended Precision quantity consists of 14 octal digits for the mantissa and 2 octal digits for the exponent. A pickapoint quantity consists of 9 octal digits.

Locations 1 through 63 of the memory are special locations which can be used as index registers. Special commands are available which increment, decrement, and test these locations. These locations can also be used for normal storage.

A high-speed, completely addressable magnetic tape system is furnished with the G.20. This equipment provides for storage of 8 million words (32 million characters) on a standard 3,600 foot reel of one-inch tape in standard blocks of 256 words (1,024 characters). Unlimited rewrite over a block is permitted. This tape unit reads or writes at 120,000 characters per second. A printer and card coupler connects the G.20 to conventional 80-column card equipment, or to the Bendix high-speed line printer providing up to 900 lines per minute. The G.20 console contains an 88-character typewriter whose full alphabet can be handled by the computer. The console also contains a set of indicator lights which can be used to inform the operator of the status of the program. A control buffer available for the G.20 system stores 4,096 characters in its core memory, and can handle off-line operations such as tape to line printer without intervention from the G.20.

All of the units of the G.20 system communicate in a common language of 10-bit characters consisting of eight numerical bits, a data flag, and a parity bit. In addition there are four circuits on which the central processor can receive interrupts, and two circuits on which it can transmit interrupts. This feature of the G.20 organization allows for very efficient use of input/output devices at the highest possible speed, with synchronization assured by the "transmit-answer" nature of the communication system. Conversations between units take place via communication lines which physically connect main terminal units to each other in the system. Every communication is a series of individual character transmissions, such that one character is sent from one unit and a reply is given to it from the other unit in the communication, for every character in a transmission.

The accumulator consists of three registers. One register will contain the 14 octal digit mantissa of any operand brought to the accumulator. Another register will contain the sign of that operand. A third register will contain the 2 octal digit exponent and the sign of the exponent.

The G.20 computer uses a single-operand command structure. Each command operates in two phases. The first is the "operand select" phase and the second is the "operation code" phase such as add or multiply. During the operand select phase, the addressing mode is inspected and the operand is assembled in the operand assembly register (OA register), which contains a

14 octal digit mantissa with its sign and a 2 octal digit exponent with its sign.

A command word in the G.20 carries, in addition to the operation code and flag bits, a two-bit mode code and two addresses: a base address, F, of 15 bits and an index address, I, of six bits. The mode bits specify the way in which F and I are to be interpreted to form an operand or an operand address. The information can be summarized as shown in Table I which indicates how the operand designator, X, is formed from the previous contents,

TABLE I

Addressing Modes

Mode	Action
0	(OA) + F + (I) = X
1	(OA) + (F) + (I) = X
2	((OA) + F + (I)) = X
3	((OA) + (F) + (I)) = X

if any, of the operand assembly register, combined with the two addresses; parentheses are used in Table I to denote "contents of".

Bendix G.20 addressing system consists of 106 opcodes including 8 address preparation codes, 56 arithmetic and logic operations, 32 block arithmetic and logical operations, two single character input/output operations, and 8 block input/output operations. In block operations the block can be of any size and two command words are needed.

Average execution times for floating-point fundamental operations are shown in Table II. All times are given in microseconds and include the average amount of time necessary to: bring the command from memory (single precision storage is supposed), assemble the operand, decode and execute the command. It may

TABLE II

Average Execution Times

COMMAND	MODE 0		MODE 1		MODE 2		MODE 3	
	(I) = 0	(I) ≠ 0	(I) = 0	(I) ≠ 0	(I) = 0	(I) ≠ 0	(I) = 0	(I) ≠ 0
Clear & add	6	12	12	18	12	18	18	24
Clear & subtract	6	12	12	18	12	18	18	24
Multiply	39	45	45	51	45	51	51	57
Divide	74	80	80	86	80	86	86	92

be noted that one memory cycle of 6 microseconds must be added when the index field (I) is different from zero.

Some unusual operations are provided which make G.20 system more reliable with respect to other systems and simplify compilation of programs written in modern language such as ALGOL.

Reverse subtract, subtract-subtract. These provide a symmetry with the conventional commands simplifying the ALGOL compiler.

Test commands. The unusual feature is that the former contents of the accumulator are undisturbed by the test operation. In most branch operations, this avoids storage and restoration of the accumulator contents.

Repeat operations. These operations are fast and convenient for column summations (e.g., for accuracy checks of data), logic simulation, and table searching.

Reverse divide. This command is convenient for compiling; usually, the numerator of a ratio is calculated and stored, and then the denominator is calculated. In the G.20 the ratio can be the next operation, avoiding intermediate storage operations. This command is also well suited to evaluation of continuous fractions, which are powerful formulas for functions approximation.

Skip (transfer of control). This feature aids in writing subroutines which can be executed from any location in core memory without modification of any of the commands of the subroutine.

Flexibility in preparation of operands and addresses of operands was provided by using the operand assembly (OA) facility. Computing time and memory space are both conserved by the use of the OA facilities, which are a generalization of three earlier computer design advances: index register, "direct" operands (mode 0), indirect addressing.

With the OA facility, it is possible to avoid costly command modification when dealing with multiple subscripts, automatic relocation of data and programs by an executive routine, and substitution of operand values. Certain computing tricks can be performed; for example, if X is in the accumulator as a result of previous calculation, $X^2 + A^*X + B$ can be obtained using only one address preparation command, one multiply command, and one add command.

2. THE ALCOM LANGUAGE

a) Generalities

The G.20 programming system is called SPACE. Its elements are translators (of which ALCOM is one), an executive system and an expandable library of programs.

The elements of the system are stored on high-speed magnetic tapes. Each translator accepts programs written in its source language and translates them into the system object language or into relocatable machine language. The translated programs can be stored, in object language form, on library tape; it is then, itself, an element of the system. The executive program provides control for the complete system. It provides diagnostic analyses, input/output control, memory assignment, interrupt control and general system control.

The ALCOM compiler is designed to run on any G.20 system which contains at least one G.20 central processor with 8,192 words of core memory, one control console, two magnetic tapes, and one card and printer coupling device with card reader, punch, and line printer. However, ALCOM will generate programs for any larger desired machine configuration, and the compiler itself will run more efficiently in a system having more magnetic tapes and more core memory. The ALCOM compiler makes use of the available flexibility of the G.20 hardware.

ALCOM language resembles both the 1958 and 1960 versions of the ALGOL language, and may be called an "ALGOL dialect". An attempt has been made to retain the best features of ALGOL and to combine them with certain G.20 hardware facilities (logic words, string constants, built-in functions, etc.) which make ALCOM good for the expression of logical as well arithmetic problems.

Apart from the above named facilities, an attempt has also been made especially to clarify ALGOL about what has caused ambiguities, obscurities or criticism. However this has been done without any loss of power of the language, which is still very flexible, while it has made possible the construction of a very compact translator.

In this paper it is not described the ALCOM language in details, but the main differences respect to the ALGOL are evidenced and for this purpose practical examples of programming are shown.

b) Logical Expressions

In ALCOM there is an extension of the ALGOL concept of logical expression; this depends on the extension of the logical value definition and on the introduction of "string variables". A logical value is written in an ALCOM program by means of an octal number* or a decimal integer: the internal representation is a word of 32 bits, which is handled as a code of ones and zeros instead of a numeric quantity. The "truth values" TRUE and FALSE are special cases of logical values: TRUE is a logic word of 32 one bits corresponding to the octal number [8] 37777777777, FALSE is represented by an all-zero logic word corresponding to the octal number [8] 00000000000.

A variable which will assume logical values is declared LOGICAL in a type declaration, a variable which will assume only truth values is declared BOOLEAN. Indeed, the elementary constituents of simple logical expressions are logical functions and variables and constants, and string variables. These are joined by the logical operators $\neg$ (NOT), $\vee$ (OR), $\wedge$ (and), and $. † All logical operations are executed in fixed point on a full 32 bits: A $\vee$ B is the binary number with a one in every bit position in which either A or B has a one, and zeros in all other bit positions.

The introduction in the language of logical values of such a type is useful for performing logic calculations and for business problem programming. Another facility of ALCOM for such a type of problems is the generalization of string use. Apart from being used as actual parameters in procedures, string constants may be directly introduced in an ALCOM program, assigning their values to suitable "string variable" by means of the "string assignment statement" whose form is:

A : = 'S'

---

*An octal number has the form:

[8] N

where N is a numeral, each digit of which is one of the figures 0, 1, 2, 3, 4, 5, 6 or 7.

†In ALCOM there are not the logical operators "$\supset$" and "$\equiv$". They may be easily substituted by a sequence of $\vee$ and $\wedge$. Respect to ALGOL it is added the operator $ (binary shift) : A $ B is the binary number resulting from performing a shift of B positions on A. The shift will be the right or the left one, depending on the sign of B.

where S is a string and A is an identifier declared in a type declaration:

STRING A,

By means of such a statement, the string is immediately stored as logical value and, once the string constant has been assigned to a string variable, any desired logical operation may be carried out on the value of the variable.

In order that a string may be stored in only one machine word it must contain no more than four characters. On the other hand it is possible to work on strings of greater length; in fact there is a special type of statement for the assignment of strings of characters to a string array. The form of this statement is:

I [L:U] : = 'S'

where I is the identifier of the string array and L and U are integers specifying the indexes of the first and last element the assignment must be given to. The string constants are useful for data processing operations, print-outs of various kinds, communication with the executive system and the like. While ALGOL is not designed for the easy handling of string operations, the facilities provided by ALCOM make it possible to specify all such operations more easily than in a hardware-oriented language.

Another facility given by ALCOM is that "Mixed expressions" may be formed. Particularly, logical or Boolean expressions may appear in arithmetic expression if enclosed between parentheses, and, vice-versa, parenthesized arithmetic expressions may appear as sub-expressions of logical expression; as for instance:

A * B + (W $\wedge$ $\neg$ R [I] $\vee$ [8] 377)

A $\vee$ (B * C + 1) $\wedge$ [8] 773)

c) Procedures

As it is well known, ALGOL imposes scanty bounds to the writing of procedures and this kind of large possibility is also known to have caused some criticisms (e.g., the occurrence of "side effects") and remarkable difficulties in the implementation of recursive procedures.

ALCOM introduces new features and excludes those possibilities offered by the ALGOL which were likely to be a cause of

ambiguous interpretations; notwithstanding that, the language remains very flexible and in the meanwhile the compiler is extremely compact.

First of all, one can see that in ALCOM procedures with multiple outputs are clearly distinguished from the single valued ones, which enable the programmer to use a functional notation. If a function can be defined through one statement only, in ALCOM it will be declared through a "function declaration" having the following form

FUNCTION F (I1,I2, ... IN) : = E

wherein the identifier F is the function name, the identifiers I1, I2, ... IN are formal inputs parameters and E is the expression supplying the rule to calculate the value of F.

If a single valued function is not likely to be defined by a function declaration, but needs, in order to be claculated, the execution of more than one statement, one can use the procedure statement. As a matter of fact, in order that such a procedure may be called by a function designator, two conditions are required: the procedure is not supposed to have outputs (excepting, naturally, the one represented by the function designator) and as last statement in the procedure body there shall be "the VALUE statement"* which assigns the value to the function designator.

The activation of a procedure by a function designator cannot therefore, to cause "side effects" on the formal parameters; in this way one may drop all the questions raised by ALGOL as regards the order of evaluating the primaries of expression. Owing to the absence of "side effects" in the ALCOM, the commutative property of the fundamental operations has a full validity and consequently the order for evaluating primaries is chosen for optimizing the compiler's work.

Thus, for instance, the expression

RHO + 1312.1_{10} − 35*((A*B/C)↑2+X) + F (X [1] + W)

is evaluated in the following order:

---

* The "VALUE statement" has the following form:

VALUE E

where E is the rule for calculating the value of the function designator. The word "VALUE" has no relation whatsoever to the *value* of the ALGOL.

```
X [1] + W
Find F of operand
Store temporarily in TS
A * B
/ C
↑ 2
+ X
* 1312.1₁₀ − 35
+ TS
+ RHO
```

It is well known that in the ALGOL language local parameters to a procedure are declared in its body whilst the global ones are not declared in it. In ALCOM real parameters are not declared, therefore there would be no way to distinguish local parameters from global parameters. As a matter of fact, ALCOM, too, distinguishes between the two types of parameters but it is to be remarked that in ALCOM it is the global parameter which is evidenced by means of an EXTERNAL declaration.

The EXTERNAL declaration, besides its normal form

```
EXTERNAL  X
```

may be shown as

```
EXTERNAL  X (M)
```

and asserts that, in the procedure in which this declaration was made, X refers to the entity named by M in the procedure one level above.

As has been hinted to, ALCOM does not accept, without reserves, recursive procedures: this fact depends on the mechanism by which procedures are operated. Generally, it is possible to exit from a procedure either by means of a GO TO (and in such a case, the calculation starts again from the point of the program specified by the GO TO statement), or by regular means, and the calculation starts again with the statement following the one of the procedure call: for this purpose, inside the body of the procedure there shall be a "RETURN statement" which will send back the control to the location marked at the moment of the procedure call.* In the single valued procedures the VALUE

*The presence, in each procedure, of at least one RETURN statement which is likely to be labeled, makes of no use in ALCOM the presence of the "dummy statement".

statement is the one operating an automatic return, and consequently no RETURN statement is used.

The exit from a recursive procedure must not be via RETURN statement. It must be through a label or switch designator which occurs in the set of actual parameters. If a RETURN statement is used, indeed, the address stored in the "mark location" when the procedure is called for the first time, will be replaced by the address stored in the "mark location" when it calls itself; after which there will be any more possibility to return to the main program.

Finally it may be noted that as there are no blocks in ALCOM, they may be substituted by means of proper procedures. Besides also in ALGOL blocks and procedures are closely similar and partially superposed concepts, in the sense that the procedure is an extension of the concept of block and a block may always be seen as a procedure called in only one point of the program.

d) Built-in Functions

A function designator may call, apart from declared functions and single valued procedures, the library functions and the "built-in functions".

The library functions are the common ones of every translator, the "built-in functions" are on the contrary facilities of the ALCOM language: they make certain special hardware facilities of the G.20 central processor available to the ALCOM programmer.

In fact, there are certain common operations which are available in the G.20 hardware, but which are not readily adaptable to algebraic notation. This class of operation includes various table-searching operations, and also such things as sum and logical sum; these latter are adaptable to an algebraic notation, but require a wider range of characters than those provided for the ALCOM language. These operations are available in the ALCOM language as built-in functions.

Four built-in functions which call for operations over a set are available in ALCOM. These are shown in Table III, where X, Y, and Z are array identifiers, and N is a variable or integer specifying the number of elements over which the operation is to be performed. X may be composed of either arithmetic (INTEGER, SINGLE, or real) or logical elements; the elements of Y must be arithmetic, while those of Z must be logical. N must

TABLE III

Function Designator	Definition
SUM (X, N)	$\sum_{i=1}^{N} X_i$
ALTSUM (Y, N)	$(-1)^N \sum_{i=1}^{N} (-1)^i Y_i$
UNION (Z, N)	$\bigcup_{i=1}^{N} Z_i$
INTRSECT (Z, N)	$\bigcap_{i=1}^{N} Z_i$

be no greater than the total number of elements of the array (X, Y, or Z).

ALCOM provides four table searches: TABLKUP, TABGRTR, TABLESS, TABEXTRT. In each of these operations, a table is searched for a word of a specified type. The search may begin with the first element of the table defining the array, or it may begin at some specified later element. If a word of this type is in the table, the value of the function is the address relative to the location at which the search started. If no such word is in the table, the value of the function is undefined, and control is transferred to a specified statement.

The "table lookup" built-in function has the form

TABLKUP (X, A, N, EXIT)

where X is a constant or a variable, A is an array identifier or a subscripted variable, N is an integer or a variable, and EXIT is a statement designator. If A is a subscripted variable designating the k^{th} element of an array, the search begins at that element. N must be no greater than the number of elements in the array beginning at A. The first N elements of this array are searched for an element equal to X. If such an element is found, the value of the function designator is the relative address of that element relative to the location at which the search started;

if no such element is in the table, control is transferred to the statement designated by EXIT. A may designate an array of arithmetic or logical elements. X must be of the same type as A.

Examples:

TABLKUP (RHO, BETATAB, 35, L13)

TABLKUP ($14_{10}-3$, GAM[K], I, ERROR)

The "Table Element Greater" and "Table Element Less", have the forms:

TABGRTR (X, A, N, EXIT)

TABLESS (X, A, N, EXIT)

where X is an arithmetic variable or constant, A is an array of arithmetic elements or a subscripted variable designating an element in such an array, N is an integer or variable, and EXIT is a statement designator. N must be no greater than the number of elements in the array beginning at A. The first N elements of the table defined by A are searched for an element greater than X (less than X for TABLESS). The relative address of the first such element in the table is the value of the function designator; if no such element exists in the table, control is transferred to the statement named by EXIT.

Examples:

TABLESS (MU, GAMMA, 13,LEL)

TABGRTR (13.5, RHO[3], I, EX)

The "Table Element Compared with Extractor" has the form:

TABEXTRT (X, A, N, EXIT)

where X is a logical variable or constant, A is an array of logical elements or a subscripted variable designating an element in such an array, N is an integer or a variable, and EXIT is a statement designator. N must be no greater than the number of elements in the array beginning at A. The first N words of the table defined by A are searched for an element which has a one in any of the bit positions in which X has a one; if there is such an element, the value of the function designator is the relative address of the first such element; if no such element is in the table, control is transferred to EXIT.

Example: the statement

TABEXTRT (1, B, 12, ERROR)

generates instructions to search the first 12 words of B for the first element with a one in the least significant bit position.

Another built-in function is CONTENTS; this will usually be used to find the first element in a table which has a specified bit set.

CONTENTS (E)

where E is any expression yielding a legal address. E will be evaluated, reduced to an integer, if necessary, and used as an address.

For example, if the current value of A is 1024, then

CONTENTS (A + 12)

is the value currently stored in location 1036. This function is particularly useful for logical programming.

3. EXAMPLES OF ALCOM PROGRAMMING

In addition to the described ones there are other differences between ALCOM and ALGOL which, so as to be clearer, will be illustrated by means of some exemplifying programs.

Example No. 1

```
          PROCEDURE ROOTS (A,B,C,RX1,RX2,IX1,IX2,K) ;
          SINGLE C;
          INTEGER K;
          BEGIN

          U:= - B/(2*A);
          V:= (B ↑ 2 - 4*A*C)/(4*A ↑ 2);
          SWITCH S[REAL, COMPLEX];
          GO TO S[K];

REAL      RX1: = U + V ↑ 0.5;
          RX2: = U - V ↑ 0.5;
          IX1: = 0;
          IX2: = 0;
          RETURN;
```

```
COMPLEX  RX1: = U;
         RX2: = U;
         IX1: = (−V) ↑ .5;
         IX2: = −(−V) ↑ .5;
         RETURN;
         END ROOTS;
```

It is to be noticed that in an ALCOM program the statements and the declarations follow one another without any fixed order: the declarations may also be placed at the end or in any place of the program.

SINGLE C is a type declaration. As a matter of fact, in ALCOM the types are: SINGLE, INTEGER, BOOLEAN, LOGICAL and STRING. It has already been reported about the last two (types); INTEGER and BOOLEAN correspond to the analogous ALGOL declarators. The SINGLE declaration tells the compiler that a corresponding variable shall be represented in single precision, by using a single machine word; this fact can be very useful if one intends to reduce the precision of the computation, by gaining room in the memory. To all variables and functions not stated in any type declarations will be assigned two machine words, which corresponds to consider them as "real".

The switch declaration (see line 6) has a syntax differing from the ALGOL one. Moreover the components of the switch may be in ALCOM statement designators only (labels and switch designators) and not common designational expressions which do not exist. As a consequence, the GO TO statement will take in ALCOM the syntactic form: TO TO < statement designator >.

In ALCOM designational expressions cannot be used; these however, may be replaced even in ALGOL. For instance the expression

go to if Ab < c *then* 17 *else* q [*if* w < 0 *then* 2 *else* n]

may be replaced by

if Ab < c *then go to* 17 *else go to* q [*if* w < 0 *then* 2 *else* n]

The RETURN statement is used to mark the dynamic end of a program or a procedure. It must appear at least once inside each body of procedure—except that in case of single valued

procedure—and is used in such a case to call back the control to its right place from where the procedure has been started. RETURN facilitates the compiler's operations, without overloading the writing of the source program. Moreover, in some cases, it may replace the "dummy statement" which takes no place in ALCOM.

Example No. 2

```
      TITLE SIGMA;
      EQUIPMENT CARDREADER:=CARD1 PRINTER:= PRIN1;
      FORMAT XX(5F14.3);
      ARRAY X, Y[20], ALF,GAM,BETA;RHO,CHI,SIG[20,20];
      INPUT XX(CARD1/X,Y);
      FOR I:=1 BY 1 TO 20 DO FOR J:=1 BY 1 TO 20 DO
      BEGIN DISCR:=X[I] ↑ 2-4*Y[J];
      IF DISCR< 0 THEN GO TO LAM;
      DISCR:=SQRT (DISCR);
      BETA [I,J]:= -X[I] + DISCR;
      RHO [I,J]:=-X[I] - DISCR;
      W:= 3.1416;
MU    SIN1:= SIN (X[I] + Y[I]);
      COS1:= COS (X[I] - Y[I]);
      ALF [I,J]:= SQRT (SIN1↑2 + W↑2*COS1↑2);
      GAM [I,J]:= SQRT (2*W*SIN1*COS1);
      END;
      MATRXMLT (ALF,BETA,CHI,20,20,20);
      MATRXMLT (GAM, RHO, SIG, 20, 20, 20);
      OUTPUT XX (PRIN1/CHI, SIG);
      RETURN;
LAM   BETA [I,J]:= RHO [I,J]:=0;
      W:= 6.2832;
      GO TO MU;
      PROCEDURE MATRXMLT(A,B,C,N,P,R);
      ARRAY A,B,C;
      BEGIN
      FOR I:= 1 BY 1 TO N DO FOR J:= 1 BY 1 TO R DO
      BEGIN C[I,J]:=0;
      FOR K :=1 BY 1 TO P DO C[I,J]:= C[I,J] + A[I,K]*B [K,J];
      END;
      RETURN;
      END MATRMLT;
      STOP;
```

SIGMA is the name of the program; the declarations TITLE SIGMA at the line 1 and STOP at the line 34 bound the scope of the program and are mandatory.

Lines 2, 3, 5 and 20 are connected with the input/output instructions, which are not going to be dealt with here.

The ARRAY declaration at line 4 is different from the ALGOL one since there is no "lower bound" which is always thought of as consisting of a unity. This is a limitation imposed by the ALCOM and yet of scanty practical importance. Another limitation imposed to the ARRAY declaration is the one by which the "upper bound" may only be an integer and not a generic expression. In this way the compiler is simplified without the language's possibilities being impaired. As a matter of fact, since declarations may always follow the statements, to any variable can be assigned the value of an expression and such a variable can be inserted in the bounds of the array.

At the line 6 delimiters BY and TO correspond respectively to *step* and *until*. In this case, the for statement is in every sense corresponding to an ALGOL one even if there are differences between the two statements aforesaid. In fact, the for list may be formed in ALCOM by only one element of the type

```
A BY B TO C
```

or of the type

```
A BY B WHILE C
```

which constitutes a limitation (anyway of no practical importance) as concerns the ALGOL. On the contrary one may impose the simultaneous variation of two indexes; this is a useful extension of the *for* statement.

Example No. 3*

```
PROCEDURE RK (X,Y,FKT,EPS,ETA,XE,YE,FI);
BOOLEAN FI; ARRAY Y,YE;
BEGIN
ARRAY Z, Y1,Y2,Y3 [10]; BOOLEAN OUT; EXTERNAL COMP;
INTEGER K, J; EXTERNAL S, HS;
PROCEDURE RK1ST (X,Y,H,XE,YE);
```

* This is the translation into ALCOM of the procedure "RK" shown as Example no. 2 of procedure declarations in the "ALGOL 60 Report"; the number of differential equations is limited to 10.

```
ARRAY Y,YE
BEGIN
ARRAY W[10], A[5]; INTEGER K, J; EXTERNAL FKT, Z;
A [1] : = A [2] : = A [5] : = H/2; A [3] : = A [4]: = H;
XE: = X;
FOR K: = 1 BY 1 TO 10 DO YE [K] : = W [K] : = Y [K];
FOR J: = 1 BY 1 TO 4 DO
BEGIN
FKT (XE,W,Z);
XE: = X + A [J];
FOR K: = 1 BY 1 TO 10 DO
BEGIN
W [K] : = Y [K] + A [J] * Z [K];
YE[K] : = YE[K] + A[J + 1] * Z[K]/3;
END;
END;
END RK1ST;
COMMENT: BEGIN OF PROGRAM;
IF FI THEN BEGIN H: = XE-X; S=0 END ELSE H: =HS;
OUT:= FALSE
AA: IF ((X+2.01 *H-XE) > 0) = (H > 0) THEN
  BEGIN HS:=H; OUT:=TRUE; H:=(XE-X)/2 END;
  RK1ST (X,Y,Z, H,X1,Y1);
BB: RK1ST (X,Y,H,X2,Y2); RK1ST (X2,Y2,H,X3,Y3);
FOR K:= 1 BY 1 TO 10 DO
  IF COMP (Y1 [K], Y3 [K], ETA) > EPS THEN GO TO CC;
FOR K:= 1 BY 1 TO 10 DO Y [K]: = Y3 [K];
IF S = 5 THEN BEGIN S: = 0; H: =2 *H END;
S: = S+1; GO TO AA;
CC: H:=0.5*H; OUT:=FALSE; X1=X2;
  FOR K:=1 BY 1 TO 10 DO Y1 [K]:= Y2 [K]
  GO TO BB;
DD: FOR K:= 1 BY 1 TO 10 DO YE [K] : = Y3 [K]
RETURN;
END RK;
```

Only capital letters are used (among the basic symbols of the ALCOM there are no small letters).

In the specification part there are no specifier *real* (real is missing in ALCOM as declarator, too) and no procedure FKT. In fact, in ALCOM no specification need be given for parameters

which represent procedures, functions, switches or labels. Besides, type declarations are mandatory only for all arrays and for all variables which appear as the dependent variable of an assignment statement or FOR statement, unless the variable or array is to represent double precision quantities.

No value part is given.

The declarations EXTERNAL COMP and EXTERNAL FKT, Z evidence the non-local quantities.

The EXTERNAL declaration is used instead of the *own* declarator which does not exist in ALCOM. This use of EXTERNAL declaration is possible because the identifiers S and HS are not at the external of the procedure.

4. THE 20-GATE LANGUAGE

The 20-GATE algebraic compiler was designed to work with a minimum equipment configuration. The translator does not require magnetic tapes and occupies about 2500 words of the magnetic core memory. Originally the GATE has been elaborated at the Carnegie Institute of Technology in Pittsburgh (Pennsylvania) to be employed on the computing machine "IBM 650" (650 GATE). As a consequence of the installation of a Bendix G.20 system in May 1961, the GATE has undergone some modifications thus assuming the name of 20-GATE.

It cannot be said that the GATE is an ALGOL dialect because the differences between these two languages are sometimes remarkable. Contrary to what has been done for ALCOM, the differences between GATE and ALGOL will not be enumerated, and only the main characteristics of the language will be briefly described.

In GATE the main constituents of the expression are constants and variables. The constants may be either numerical or alphabetical. The former, in the traditional meaning of number, are expressed as in ALGOL, by means of a decimal number followed by an exponent part; the latter are the strings of ALGOL and, as in ALCOM, they may appear at the right part of an assignment statement.

In GATE there are two classes of variables, the integer ones and the floating point ones. The identifiers of the variables cannot arbitrarily be chosen by the programmer, but they must be formed by a special letter followed by an index. Particularly, the letters I, J, K must be used for the integer variables, while

the letters C, D, G, H, X, Y, Z, B must be used for the floating point variables. By index it is meant:

a non negative integer variable
an integer variable
a parenthesized expression
a unary operator followed by a primary

For example, integer variables are:

J32 II1 K(J2 – 5)

and floating point variables are

Y2 Z48 C(J5 – 4).

Therefore, as the type of a variable can be recognized by means of its syntactic form, the *integer* and *real* declaration will not be necessary. The GATE does not deal with *Boolean* variables, even if relations in conditional statements can be used in any way.

Beyond the integer and the floating point variables, the GATE, like the ALGOL, uses variables that are members of array; yet, in particular, the GATE considers bidimensional arrays only whose variables will be expressed in the form

V (α,β)

where V is a letter used for identifying the array (I, J, K or C, D, G, H, X, Y, Z depending on the cases) and α and β are indexes.

Array variables are

J1 (K8, J1 * 5) ZJ1 (K8, J45-167)

The way of putting in the program any type of variable or array is the "dimension statement" whose syntax can be described as follows:

< dimension statement > : : = DIMENSION < dimension element> |

< dimension statement> < dimension element>

< dimension element > : : = < name type> (< integer>) |

< name type> (< integer>, < B or C>, < B or C>)

< B or C> : : = < integer> | < integer> < name type> < integer>

Dimension statements, for instance, are:

DIMENSION J(2) X(15) D(200)

DIMENSION X(100, K6, J2) J(40, 5, 15) Z(50)

The first statement means that in the program there will be introduced integer variables J1 and J2 and floating point variables X1, X2, ... X15 and D1, D2, ... D200; the second statement means that the letter X will be used to identify a matrix of 100 real elements. The number of columns in such a matrix is K6, while J2 is the index of the basic element of the array. The statements J(40, 5, 15) and Z(50) have a similar meaning.

Apart from the usual binary operations of addition (written as +), subtraction (−), multiplication (*), division (/) and exponentation (↑), the GATE provides the unary operations plus (+), negation (−), absolute value (A), truncation (↓) and location (L). Each of these unary operations operates on the primary immediately to its right. The first three operations operate as expected; ↓ converts floating point values to integer form; the operation is one of truncation and reduction module 2↑32. The operator "L", which may only be applied to a variable or a constant, refers to the location (i.e., machine address) of the variable or constant. 'L' is usually used only in function parameters. Arithmetic expressions are written by means of the mentioned operations and by the use of parentheses.

In GATE the assignment statement is called substitution statement and its form is

$V \leftarrow \beta$

where V is any variable and β is any arithmetic expression. A substitution statement in parentheses may be used as an element of an expression. In this case, the substitution will be performed and the value of the parenthesized substitution statement will be the same as the expression to the right of the "←". Thus, the statement

Z1 ← Z1 * Z1 * (Z1 ← SIN.(X1))

will store in Z1 the cube of SIN.(X1). However, it will be run in less time than the statement

Z1 ← SIN.(X1) ↑ 3.

In GATE, labels may be only numbers and are called "statement numbers"; as a consequence of that, the "go to statement" has the form

GO TO φ

where φ is an arithmetic expression. It means "execute next statement whose number is equal to the value of φ". In GATE there are no *switch*.

The iteration statement, corresponding to the "for statement" of the ALGOL, is expressed in the form:

M: K, V, A, B, C

M is the label of the iteration statement, K is the label of the last statement of the block controlled by the iteration statement; V is the variable whose value is to be systematically altered by the iteration statement, and A, B, C are all expressions. In ALGOL, such a statement would be expressed as follows

for V: = A *step* B *until* C

and properly specifying the iteration scope by means of parentheses *begin* and *end*.

The conditional statement is expressed in the form

(statement) IF A $\mathfrak{R}$ B

where A and B may be any expression and $\mathfrak{R}$ a relational operator. In case the relation is true, the statement is executed, otherwise the computation goes on to the next statements of the program. The statement, in its turn, may be conditional thus creating a chain of IF as for example

GO TO 7 IF K1 < 3 IF K2 > 4 IF K3 = K4

As an example a GATE program is here shown for the solution of a system of 40 equations in 40 unknowns whose coefficients have been made by means of iteration statements.

```
   BENDIX, FRANK CVETIC              NAM
   N LINEAR EQUATIONS                 M
     24 IS THE HIGHEST STATEMENT NUMBER
   DIMENSIOND(1650,J8,1)C(60)J(8)I(60)K(60)
 1 J1←40
   J8←J1+1
   K0←J1
   22,I0,1,1,J1,
   21,J0,1,1,J8-I0,
   D(I0,I0+J0-1)←J0
21 D(I0+J0-1,I0)←J0
22 END OF OUTER LOOP                  C
```

```
   24,I0,1,1,J1,
   K0←0
   23,J0,1,1,J1,
23 K0←K0+D(I0,J0)
24 D(I0,J8)←K0
   SET UP TABLE FOR INDIRECT ADDR OF COLUMNS; ROWSC
   2,I0,1,1,J1+1,
 2 Π0←I0; KI0←I0
   SET ROW COUNT C
   12,I0,1,1,J1,
   K0←I0
   J2←J1+1
   CHECK FOR ZERO PIVOTAL ELEMENT C
19 GO TO 3 IF D(Π0,KK0)=0
 8 D(Π0,KJ2)←D(Π0,KJ2)/D(Π0,KK0)
   GO TO 8 IF J2⌐|<K0; J2←J2−1
   J2←1; K0←J8
10 GO TO 11 IF J2=I0
 9 D(IJ2, KK0)←D(IJ2, KK0)−D(Π0, KK0) *D(IJ2, KI0)
   GO TO 9 IF K0⌐|<I0; K0←K0−1
11 GO TO 10 IF J2⌐|>J1; J2←J2+1; K0←J8
12 END OF OUTER LOOP                    C
13 J2←J1+1
   TJ1
   14,I0,1,1,J1,
14 TD(KI0,J2)
   HALT
 3 PERMUTATION OF COLUMNS               C
   J3←K0+1; J4←KK0
 7 GO TO 4 IF J3=J1+1
   GO TO 6 IF D(Π0,KJ3)≠0
   GO TO 7; J3←J3+1
 6 GO TO 8; KJ3←KJ4; KK0←KJ3
 4 ALL ELEMENTS OF ROW =ZERO            C
   GO TO 5 IF D(Π0, J1+1)≠0
   PERMUTATION OF ROWS                  C
   J5←K0; J3←I0+1; J4←Π0
17 GO TO 20 IF J3>J1
15 GO TO 18 IF D(IJ3,KJ5)≠0
   GO TO 16 IF J5>J1; J5←J5+1
   GO TO 15
16 GO TO 17; J3←J3+1; J5←K0
 5 TI0
```

```
   AT$AN I$T$NCON$T$SIST$T$ENCY$T$ EXI$T$STS $
   GO TO 13
20 TI0
   AT$NO U$T$NIQU$T$E SO$T$LUTI$T$ON $
   GO TO 13
18 GO TO 19; LJ3←LJ4; II0←LJ3
   PROGRAM END
```

Apart from what has already been illustrated, the program shows the following features of the GATE.

"24 IS THE HIGHEST STATEMENT NUMBER" is a kind of "system statements", which are not parts of the computation, but only give information to the translator.

The statements ending by the letters M or C are comments.

Several statements may be combined to form a single statement by the use of the combination symbol ";". The order of statement execution is from the right-most listed one to the left-most listed one.

The statements TJ1, TD(KI0,J2) and TI0 mean: print the numeric value of the variables J1, D(KI0,J2) and I0. On the contrary, the statements AT impose to print in alphabetical form the following phrases:

"AN INCONSISTENCY EXISTS"

and

"NO UNIQUE SOLUTION"

A GATE program must always contain a "HALT statement" which causes control to pass out of the compiled program to the supervisory monitor program or to the machine operator. The last statement of each program must be an "end statement" of the form:

"PROGRAM END"

LE MAGE, UN LANGAGE DERIVE DE L'ALGOL ADAPTE AUX PETITES MACHINES

Louis Bosset

Sociéte Européenne pour le Traitement de l'Information
Paris, France

I. INTRODUCTION ET GÉNÉRALITÉS

Il n'est pas dans mon intention de vous décrire en détail le langage MAGE, bien qu'il soit simple, les quinze minutes qui me sont imparties seraient insuffisantes. Je donnerai donc simplement les grandes lignes de ce langage en signalant au passage ce qui le rapproche ou au contraire le différentie de l'ALGOL 60. Je préciserai également l'organisation adoptée pour le compilateur et les raisons qui y ont conduit.

Ainsi que le titre l'indique clairement, notre but était de réaliser un langage symbolique de programmation pour petits calculateurs. Qu'entend-on par "petit," les limites ne sont pas précises, disons pour fixer les idées qu'un compilateur du langage MAGE devait pouvoir être réalisé pour le calculateur PACKARD BELL 250. Ce calculateur dispose en version minimale de 2.320 mots de mémoire rapide et en version maximale de 16.000 mots avec connexion bandes magnétiques possible. Le MAGE devait pouvoir être utilisé avec la version minimale.

L'utilisation la plus courante de ce type de calculateur, dans le domaine scientifique, se rencontre dans les laboratoires de recherche et bureaux d'étude, en tant que "machine ouverte" à la disposition des ingénieurs et des chercheurs. Dans ce cas il n'y a aucun personnel affecté en permanence au calculateur. Il est donc absolument nécessaire de disposer d'une auto programmation suffisamment puissante et facile d'emploi.

Toutefois, la réalisation d'un compilateur d'ALGOL en moins de 1800 mots étant difficile à envisager, nous avons établi un langage plus simple en respectant le plus possible certaines convention de l'ALGOL.

Afin d'utiliser au maximum les possibilités du calculateur, nous avons adopté l'organisation suivante pour le compilateur

1)—Assemblage du programme en langage machine simultanément avec l'introduction du programme symbolique.

2)—Extraction du programme langage machine par bloc, sur bande perforée ou ruban magnétique si le calculateur en dispose. Avec cette organisation, ni le programme symbolique en MAGE ni le programme langage machine ne séjourne dans le calculateur.

II. DESCRIPTION DU LANGAGE

Mais venons en tout de suite à une description sommaire du langage. Nous le décrirons, pour employer la terminologie consacrée, et chaque fois que cela sera possible, comme un langage de référence et non comme un langage de publication. Ceci nous permettra d'utiliser le plus possible les signes conventionnels de l'ALGOL et évitera de vous infliger une liste de symboles qui n'auraient peut-être pas habituellement la même signification pour chacun.

Un programme Mage se présente sous la forme d'une suite d'instructions composées, chacune d'elles ne comportant généralement qu'une déclaration. Ceci n'est toutefois pas entièrement exact en ce qui concerne les sous-programmes rédigés en Mage. On a tenu en effet, ne serait-ce que pour des questions d'encombrement du programme assemblé, à conserver la possibilité de constituer certaines séquences de calcul en sous-programmes. Comme nous le verrons par la suite, l'organisation adoptée dans ce cas tient à la fois du bloc et de la procédure ALGOL. On pourra donc considérer un sous-programme comme une instruction composée comportant plusieurs déclarations.

L'ordre d'exécution des instructions est celui de leur écriture, sauf dans le cas d'instructions de rupture de séquences. Dans ce cas, on a la possibilité d'étiquetter les instructions à désigner.

1—Vocabulaire

Nous allons maintenant décrire le vocabulaire et tout d'abord les symboles de base:

11—Symboles de base. Nous distinguerons les groupes habituels:

- Lettres (Alphabet complet, majuscule et minuscule)
- Chiffres (de 0 à 9)
- Délimiteurs.

Les valeurs logiques ne sont pas désignées explicitement dans un programme Mage, ainsi que nous le verrons dans quelques instants.

Voici la liste des principaux délimiteurs:

— Opérateurs arithmétiques + | - | × | / | ↑

— Opérateurs de relation = | < | > | ≠

— Opérateurs logiques ∧ ∨

— Opérateurs séquentiels SI || POUR | FAIRE | ALLER A |

Rappelons que nous utilisons les termes de l'ALGOL, mais que ceux-ci ne se traduisent pas toujours par le même mot en MAGE; ainsi FAIRE et ALLER A s'expriment en fait par EXECUTER et VERS.

— Separateurs | , | : | ; | := | — | "

Le signe := se traduit en Mage par =.

Le ; par un retour chariot, le moyen d'introduction le plus courant pour le calculateur PB 250 étant la Flexowriter.

On ne peut faire de commentaires.

— Crochets | FIN | [|]

En fait, on emploie des mots différents selon qu'il s'agit de la fin d'un programme, de la fin d'une procédure ou de la fin d'une boucle POUR.

Les crochets d'indices sont en langage de publication sur PB 250, le passage de majuscule à miniscule sur la machine à écrire. De plus, ils ne contiennent qu'un indice.

Le séparateur " est utilisé pour les étiquettes.

— Déclarateurs: ENTIER (traduit en Mage par INDICE)
PROCEDURE
TABLEAU (traduit par DIMENSION)

— Spécificateurs: ETIQUETTE

Après les symboles de base, nous terminerons cette description du vocabulaire par les Nombres et les Identificateurs.

12—Les nombres. On peut faire figurer dans le programme Mage, soit des entiers sans signe, soit des nombres avec signe et comprenant une partie entière et une partie décimale. Les nombres étant de 6 chiffres au maximum plus le symbole de cadrage qui est la virgule.

On ne dispose pas de facteur de cadrage (puissance de 10).

13—Les Identificateurs. Ils sont représentés par un groupe quelconque de caractères dont toutefois seuls les trois premiers sont interprétés, les autres sont simplement destinés à faciliter la lecture.

Les identificateurs d'indice sont constitués par une seule lettre d'ailleurs quelconque.

Les fonctions standards possédant un identificateur sont:

- Valeur absolue
- Sinus
- Cosinus
- Tangente
- Exponentielle e^x
- Logarithme népérien
- Logarithme décimal
- Arc tangente

On dispose également de fonctions de transfert au sens ALGOL: transformation d'un entier sans signe en un nombre positif réel et inversement transformation de la partie entière d'un nombre positif réel en un entier sans signe.

2—Syntaxe

Nous parlons maintenant de la partie syntaxe du Mage et tout d'abord quels sont les différents types d'expressions rencontrés dans le Mage?

Il y en a essentiellement deux:

21—Expressions arithmétiques. Ce sont des expressions arithmétiques simples en ce sens qu'elles ne contiennent pas de proposition SI.

Elles sont constituées à l'aide des opérateurs arithmétiques des identificateurs et des nombres. L'ordre de priorité des opérations est toujours l'ordre d'ecriture, de gauche à droite, il n'y a pas de parenthèses.

Ces expressions constituent des règles de calculs arithmétiques sur nombre entier sans signe (calculs d'indices) ou sur nombres réels.

Les calculs sur nombres réels sont toujour effectués en virgule flottante avec 11 chiffres de mantisse.

22—Expressions de désignation. Il n'y a pas d'expression booléennes ou du moins elles ne figurent que dans le cadre des expressions de désignation, la valeur booléenne qu'elles calculent étant exploiter immédiatement par une proposition SI pour choisir entre deux étiquettes ou entre une étiquette et l'instruction suivante.

Une expression de désignation comporte alors une proposition SI, une expression booléenne et une ou deux étiquettes.

Les expressions booléennes sont composées à l'aide des opérateurs de relation, des opérateurs logiques, ainsi que des identificateurs (variables et fonctions standards) mais les nombres ne peuvent figurer dans une expression de désignation.

L'ordre d'exécution des opérations dans une expression booléenne est le même que pour les expressions arithmétiques. Un opérateur logique porte sur la valeur booléenne de l'expression qui la précède et celle de l'opérateur de relation qui suit.

3—Instructions

Nous terminerons cette description du Mage par l'organisation de quelques types d'instruction:

31—Instructions d'affectation. Une seule variable figure dans la partie gauche:

ALPHA := 2 + sinus (PHI)

32—Instruction Aller à (VERS en Mage). Elle provoque une rupture de séquence vers l'étiquette qui est précisée:

VERS : = "UN"

33—Instruction conditionnelle. Elle se compose d'une expression de désignation:

SI : ALPHA > BETA ∧ ALPHA < GAMMA "UN" "DEUX"

34—Instruction POUR. La syntaxe d'une instruction POUR est la suivante:

⟨identificateur d'indice⟩ , ⟨valeur initiale⟩ , ⟨PAS⟩ , ⟨ valeur finale⟩

Ces trois dernières quantités pouvant être des identificateurs d'indice ou des entiers sans signe.

Il existe de plus une instruction fin de boucle: RETOUR

ex: POUR : i : = m, n, p ⟨.....⟩ ; RETOUR

35—Instruction PROCEDURE. Comme nous le signalions plus haut, les sous-programmes MAGE tiennent à la fois du bloc et de la procédure ALGOL.

En effet, il n'y a pas de liste de paramètres, ce qui l'apparente au bloc, toutefois, une instruction permet d'effectuer le remplacement de nom pour les variables indicées.

De plus, on peut par une procédure définir une valeur de fonction; l'identificateur d'une procédure peut alors être appelé par l'opérateur séquentiel FAIRE comme un bloc ou comme un identificateur de fonction dans une expression arithmétique.

36—Entrée-Sortie. Il reste à dire quelques mots des instructions d'entrée-sortie, ce qui est volontairement ignoré par ALGOL.

Le PB 250 possède comme moyen d'entrée courant, un lecteur de bande perforée et un clavier et comme moyen de sortie, un perforateur de bande et une machine à écrire.

Les instructions d'entrée se composent d'une déclaration de lecture suivie de la liste des indentificateurs, des valeurs à introduire (entiers sans signe ou nombres réels).

Cette déclaration provoque l'introduction des valeurs par le clavier ou le lecteur, le choix étant fait au moment de l'exploitation.

Les instructions de sortie se composent d'une déclaration d'impression suivie de la liste des identificateurs. On peut imprimer soit les valeurs, soit les identificateurs eux-mêmes.

III. ORGANISATION DU COMPILATEUR

Le compilateur du MAGE réalisé pour le PB 250 comporte essentiellement deux parties:

1)—Un ensemble de micro programmes (trente) pouvant être classés en quatre catégories:

11)—Lecture de caractères et de mots

12)—Progression d'adresses

13)—Composition d'instructions langage machine et de séquences d'instructions

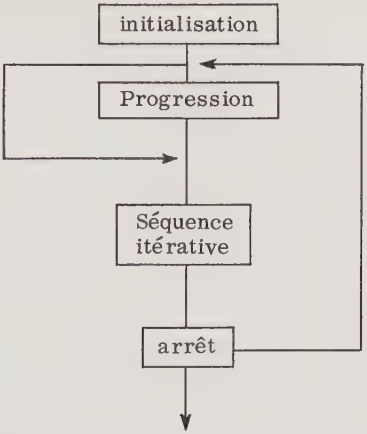
14)—Traitement des tables.

Ils sont organisés de telle sorte qu'ils peuvent être utilisés en chaine, un sous programme se renvoyant dans un autre et ainsi de suite en nombre illimité.

2)—La deuxième partie permet de traduire les instructions composées du MAGE en instructions langages machines, par l'intermédiaire des micro-programmes c'est donc essentiellement un assemblage des micro-programmes.

L'organisation en assemblage instantané par bloc d'instructions langage machine conduit à certaines contraintes en particulier pour les opérations de rupture de séquence.

En ce qui concerne les propositions SI par exemple, l'utilisation dans les instructions conditionnelles d'étiquettes non encore affectées est rendue possible par l'organisation d'une table d'instructions de rupture de séquence servant de relais, ce qui per-



met de coder complètement une proposition SI dès qu'elle est lue, la table d'instructions relais se complétant par la suite.

De même dans une instruction POUR, l'organisation de la séquence itérative est la suivante:

Ceci permet de ne coder l'arrêt qu'après l'assemblage du corps de la séquence itérative en supprimant par contre la possibilité d'effectuer zéro fois la séquence.

Cette organisation a permis de réaliser un traducteur en 1.800 instructions langage machine.

JOVIAL: A GENERAL ALGORITHMIC LANGUAGE

J. I. Schwartz

System Development Corporation, Santa Monica, California, USA

INTRODUCTION

The System Development Corporation's responsibilities include the production of large, complex programming systems. The development of these systems requires that as many as hundreds of thousands of computer instructions be written, checked out, and maintained by large groups of programmers. This makes necessary a large amount of programmer to programmer, and programmer to non-programmer intercommunication. Also, these systems require extremely high reliability and are subject to frequent modification during their production and after their delivery. Experience has shown that programming in pure machine language did little in the satisfaction of these requirements. Thus, to assist in these efforts, a "higher level" programming language called JOVIAL has been developed, and compilers for JOVIAL have been written for a variety of large-scale computers. It is not possible in a relatively short presentation such as this to describe all of the JOVIAL language. Therefore, this paper will describe only what the author considers to be the main area of departure of JOVIAL from ALGOL, i.e., the data description capability of the language. Secondly, a short discussion of the techniques of compiler implementation will be given. Finally, this paper describes some of the experiences in the development of JOVIAL and the conclusions one may infer from these.

I. JOVIAL'S BACKGROUND

JOVIAL was patterned primarily after ALGOL '58. Thus the statement format and, in general, its delimiters, are similar to that language. However, the purpose for which JOVIAL was intended was considerably different from that which the primarily computational nature of ALGOL permits.

JOVIAL was to be used in the production of large programming systems. The nature of these systems is such that many areas of programming are encompassed, including utility, simulation, information retrieval, executive (or control) functions, decision-making, and computation. At the time of JOVIAL's initial development, the decision to incorporate the programming potential for all these areas was made, thus allowing an entire system to be programmed in one language, rather than merging programs written in different languages into the system.

With this "general" concept as a goal, and the influence of other languages, such as ALGOL '60 and COBOL, JOVIAL has grown to be a rather powerful, flexible, although somewhat informal programming language, considerably divergent from its initial model, ALGOL '58.

II. JOVIAL'S MAIN CHARACTERISTICS

Complete generality of a programming language can be achieved in the trivial sense by using nothing but computer instructions. Using computer language, one has all the flexibility which is achievable with a given computer. One of the primary concerns on the part of JOVIAL users has been the possible loss of machine power using a language like JOVIAL.

Indeed, using pure ALGOL for most utility programs is usually not practicable, as is the case for other noncomputational types of programming.

However, since JOVIAL was to be used for these applications, certain steps had to be taken to insure little or no loss of computer power in its design. Thus it had to depart in some important respects from ALGOL. The major difference lies in the operands of the language. An examination of the basic set of operators in JOVIAL and ALGOL shows relatively minor differences between the two languages. However, the structure and definition capability of the operands (variables and constants) shows a distinct difference between the two languages. The following describes in some details the data definition capability of JOVIAL.

III. ITEMS

The smallest piece of datum which can be declared and given an identifier is the *Item*. The *Item Declaration* gives the following information:

1. Identifier
2. Type
3. Size

where applicable

4. Accuracy
5. Signed or Unsigned Indicator
6. A Round or Truncate Indicator

7. "Packing" Information (optional)

Fig. 1

The possible types of items are:

1. Floating—A floating item has the form and significance available with a given computer. Internally, it has a sign, characteristic, and mantissa, and is usually used with computations requiring a large range.

2. Fixed integer—A fixed integer is a quantity which can never have a fractional part. It is normally used for counting, indexing, and addressing. In a binary computer, its representation is the exact binary equivalent of the decimal value.

3. Fixed number with fractional part—This type of value contains both integer and fractional parts. In binary, it is represented by a number which contains the integer and fractional part separated by an (imaginary) binary point. It is used for items which have a small range, require a very precise significance, or in situations where storage requirements are such that the use of floating point quantities would overburden the system.

4. Alphanumeric—This type of item is made up of a set of characters. On binary machines, each character is (normally) six bits in length. In JOVIAL, both Hollerith and Standard Transmission Code character items may be defined. This type of item is normally used for names, titles, input-output, and other such applications.

5. Status—A status item is used to represent an object which has a fixed number (usually "small") of states, each state in turn representable by a unique name. For example, the statuses of an item describing the weather and might be RAIN, SNOW, CLEAR, and FOG. In JOVIAL, one would use these names when setting or questioning the item. Internally, each status would be represented by a unique number, normally assigned in the sequence 0, 1, 2, 3...

6. Boolean—Boolean items are used to represent the two state condition of TRUE or FALSE. Boolean items are always represented by one bit in a binary machine.

The size of an individual item is usually less than or equal to the size of one computer word. However, some of the more recent compilers permit items to be more than one word in length. Mixing of fixed and floating point values in arithmetic expressions is permitted, and the redefinition of an item from one mode to the other normally has no effect on a JOVIAL program.

IV. ENTRIES

In many applications, it is desirable to group all the individual information (or items) describing a particular object in one "area" of the computer. Such a group of items is called an *entry*. Two examples of entry formats are given in Figure 2.

Possible Entry Formats

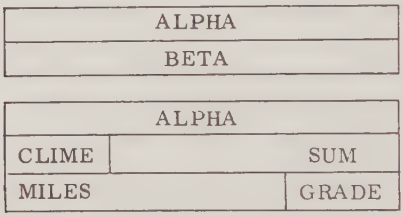


Fig. 2

It will be noted that the second entry format contains several items in one word. This ability to "pack" items is an important feature of JOVIAL which has aided significantly in the production of large systems. Incidentally, the entry formats pictured have contiguous consecutive words. This form of entry is called *serial*. Another form, *parallel*, will be described later.

V. TABLES

The most frequent need in large data processing problems is for more than one of a particular object to be described. Thus there is a need for grouping of entries. A group of entries is called a *table*. A table is declared and given an identifier with a Table Declaration, which contains the information described in figure 3.

Table Declaration

1. Identifier
2. Fixed or Variable Length Indicator
3. Maximum number of entries
4. Size of entry (optional)
5. Serial or Parallel Entry Indicator

Fig. 3

The table declaration is immediately followed by all item declarations for items in that table (enclosed in the brackets "BEGIN, END"). Note that specification of the size of the entry is optional. When this information is not specified, the compiler will automatically "pack" the items of a table in an entry. When it is specified, the programmer must specify the packing with the packing information permitted in the Item Declarations.

Figure 4 shows some tables containing serial entry formats.

Serial Tables

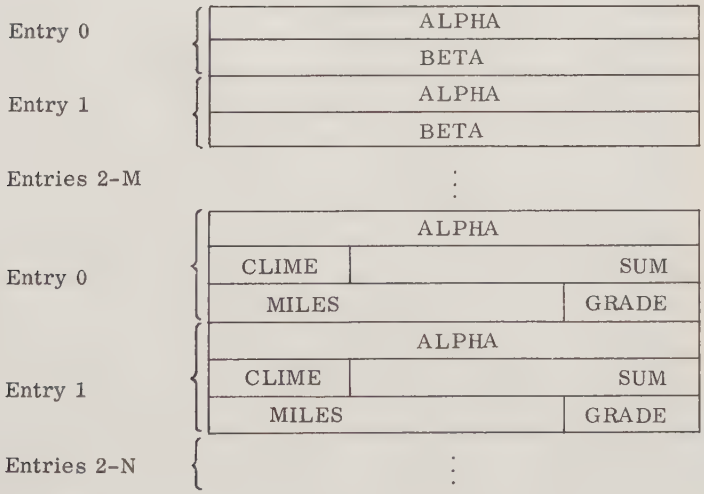


Fig. 4

It should be noted that although the number of words in an entry can be more than one, the size of the entry has no direct bearing on a JOVIAL program.

For example, Figure 5 shows the required program to clear all of the items named ALPHA in either table described in Figure 4.

Clearing Items of a Table

```
FOR      I = ALL(ALPHA);
ALPHA    [I] = 0;
```

Fig. 5

The *parallel* entry format has been designed to produce more efficient programs.

An illustration of the parallel table format is given in Figure 6.

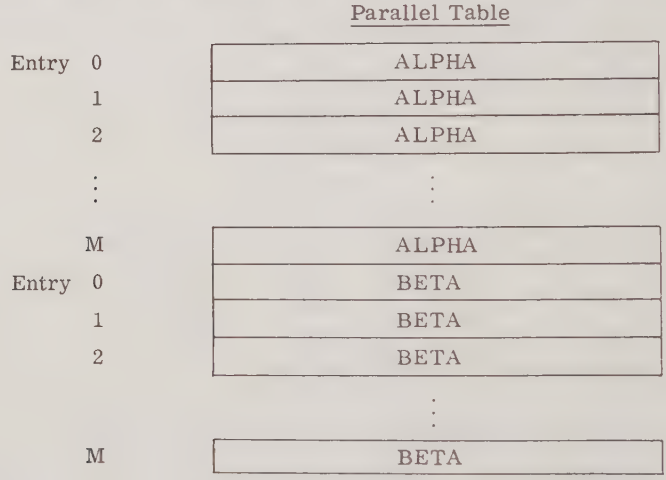


Fig. 6

In parallel entries, all of the ith words in an entry are contiguous. Again, using this format, the program described in Figure 5 would be used to clear all ALPHA's. This independence of the program from the data structure gives considerable flexibility.

The preceding tables have been of *fixed entry* format. That is, every entry contains the same items. Without going into any more detail, it should be mentioned that it is possible to design tables whose entries vary and contain different items and numbers of words per entry.

VI. STRINGS

The entries so far described have all items repeated only once in a computer word. Another form of packing permits the repetition of one item in a single word. This type of packing is described with a *String Declaration*. Figure 7 illustrates a table containing one string, called GAMMA.

Table with String

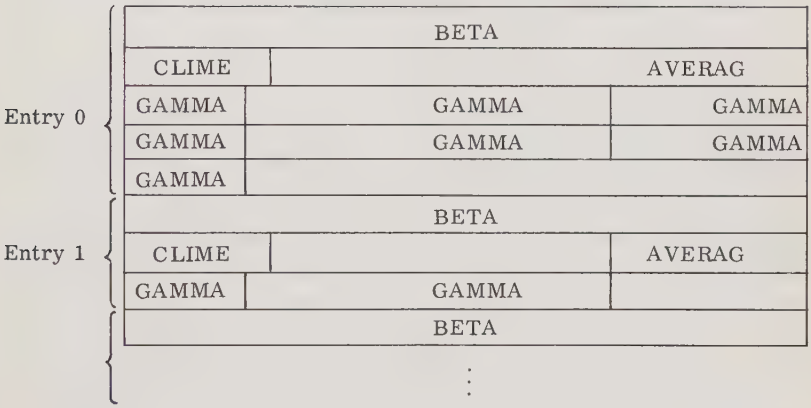


Fig. 7

VII. ARRAYS

All tables are considered to be one-dimensional. The ability to describe elements in a space of more than one dimension is available with the array declaration. There is no specific limit to the number of dimensions of an array, although there is a practical limit due to computer size considerations.

VIII. FUNCTIONAL MODIFIERS

It was stated previously that the smallest element which can be given an identifier is an item. However, it is often necessary to examine or set parts of items. For this and other reasons, a set of functional modifiers has been included in the language. Some of the more commonly used modifiers are shown in Figure 8.

Some Functional Modifiers

1. BIT[I,J] (ALPHA [K])
   This allows access to J bits starting with bit I of the Kth ALPHA.
2. BYTE[I,J] (NAME [K])
   This allows access to J alphanumeric characters starting with the Ith character of the Kth NAME.
3. ENTRY(TAB1 [L])
   This modifier gives access to the Lth entry of table TAB1.

Fig. 8

I, J, K, and L of Figure 8 can be either constant or variable.

The preceding has described very briefly some of the aspects of JOVIAL. Other and more detailed properties of the language can be found in existing documentation. Next, we shall consider some of the techniques used to implement compilers for JOVIAL.

IX. COMPILER TECHNIQUES

In addition to the fact that JOVIAL was to be used for a wide variety of applications, it was also known at its inception that it would be used to program a number of different computers. Thus, it had to be as independent of a given computer as possible. In addition, the ability to produce compilers for different computers fairly fast had to be developed.

The main technique used was the following. The compiler was divided into two major portions. The first part is called the *generator*. The generator is essentially computer independent; its major function is the transformation of the JOVIAL language into an elementary form called the *Intermediate Language*.

The second part of the compiler is called the *translator*. Its function is to translate the Intermediate Language into a specific computer language. Thus the translator is the computer-dependent part of the compiler. This process, with a simple example, is illustrated in Figure 9.

Both the generator and translator are programmed in JOVIAL. The fact that the generator is computer independent permits its use on all computers that need JOVIAL compilers. This, combined with the fact that the compiler which is written to produce programs for one computer can be run on another computer, per-

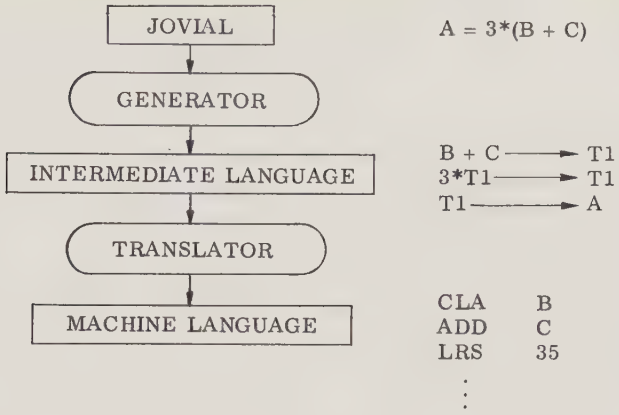


Fig. 9

mits a fairly rapid development of compilers or new computers, since only a new translator need be written for each new compiler. The following is an illustration of the use of this technique for producing compilers on different computers.

Given program P, the following notation will be used:

$$P^{S3}_{S1\ S2}$$

where

S1 = form in which P exists,
S2 = Input which P accepts,
S3 = Output which P produces.

Examples:

$_{A}C^{B}_{J}$ = compiler which runs on computer A, inputs JOVIAL and outputs computer instructions for computer B.

$_{A}G^{IL}_{J}$ = generator which runs on computer A, inputs JOVIAL and outputs the intermediate language.

$_{A}T^{B}_{IL}$ = translator which runs on computer A, inputs the intermediate language and outputs computer instructions for computer B.

Obviously,

$${}_{A}C_{J}^{B} = {}_{A}G_{J}^{IL} + {}_{A}T_{IL}^{B}.$$

One also may assume that there exists a ${}_{J}G_{J}^{IL}$ (a version of the generator programmed in JOVIAL). Now, given ${}_{A}C_{J}^{A}$, one can produce ${}_{E}C_{J}^{E}$ in the following fashion:

First, ${}_{J}G_{J}^{IL} \rightarrow {}_{A}C_{J}^{A} \rightarrow {}_{A}G_{J}^{IL}$, where $\alpha \rightarrow C \rightarrow \beta$, means α is compiled by C to produce β.

Then, ${}_{J}T_{IL}^{E} \rightarrow {}_{A}C_{J}^{A} \rightarrow {}_{A}T_{IL}^{E}$.

Given the previous outputs, ${}_{A}G_{J}^{IL} + {}_{A}T_{IL}^{E} = {}_{A}C_{J}^{E}$, producing a new compiler.

Using this compiler, ${}_{J}G_{J}^{IL} \rightarrow {}_{A}C_{J}^{E} \rightarrow {}_{E}G_{J}^{IL}$,

$${}_{J}T_{IL}^{E} \rightarrow {}_{A}C_{J}^{E} \rightarrow {}_{E}T_{IL}^{E},$$

and then,

$${}_{E}G_{J}^{IL} + {}_{E}T_{IL}^{E} = {}_{E}C_{J}^{E}$$

Of course, expansions of this technique can be used to illustrate the technique of improving the JOVIAL language and compilers.

X. EXPERIENCE

JOVIAL has been under development and in use for approximately three years. At present, compilers for five different computers are in operation and are being used for the production of programming systems. Since the experience we have had with the language and its compilers is probably similar to that of present and future language and compiler writers, a summary of these seems worthwhile at this time.

First of all, what have the problems been and where have original goals not been met?

Undoubtedly the biggest single source of problems has been

the size of the compiler building task. Being the rather powerful language that it is, JOVIAL has required compilers of approximately 50,000 machine instructions for each computer, including the generator and translator. Thus the compilers themselves have been fair-sized programming systems.

Usually, approximately one year was required to complete the first version of a compiler for a given computer. Including the generator effort, about ten people worked on each compiler.

Not including the usual problems of personnel changes and requirements for more computer time than was available, the following developmental problems were encountered.

1. Each translator was financed by individual contracts, each of which had its own pressing schedules. When a choice between implementing all of JOVIAL and meeting a schedule arose, the decision was in favor of meeting the schedule. As a result, none of the initial translators implemented the complete language, and in some cases parts of the language were implemented differently in the different translators.

Another problem, not unlike similar efforts elsewhere, was the fact that most compilers were late. Those that were delivered on time were rather unreliable. Experience has shown that it takes between 6 and 12 months of heavy usage for the compilers to be nearly 100% reliable.

Also, the initial versions of the compilers were not as fast nor as efficient as desired.

2. The first use of the generator-translator technique was not as valuable as it could have been. First of all, it has since become obvious that the cutoff point for computer independence can be moved much further down in the compilation process than the current Intermediate Language. By the effective parameterization of some of the computer's characteristics, the computer-independent generator can include allocation of data, index register assignment, and assignment of fixed-point scaling for intermediate results of calculations. Until now, these have been included in each translator, thus requiring a considerable duplication of effort as well as making the translator the biggest part of the compilation process.

Secondly, although the theory that the generator would go almost effortlessly from one computer to another was substantiated, certain limitations in practice interfered to some extent. In order to make an efficient generator for every computer, modi-

fications in its overall operation had to be made manually as a function of computer size. This was considered to be more worthwhile than having a single generator designed to operate for the smallest computer. Also, the operating utility system under which the compiler was to function necessitated certain changes on the part of the generator. If the various utility systems had been designed in the same fashion, these latter changes would have been unnecessary.

The first model of the generator was itself a fairly big task, and two of the compilers could not wait for its completion. Thus the generator used by these compilers is a different program and can only take advantage of some of the techniques of the central generator.

3. As time progressed, certain deficiencies in the language were noted, but any improvements had to wait for the completion of the initial effort.

4. Because of the size of the compilers necessary to implement the complete JOVIAL, it does not seem practicable to program JOVIAL compilers to run on "small" computers. The smallest computer for which JOVIAL has a compiler has 16384 high-speed storage cells, a large drum, and six tape units. On this computer considerable difficulty has been encountered trying to implement the entire language with a compiler that runs with a tolerable speed. Of course, useful subsets of JOVIAL can be compiled on smaller computers. Some of the positive results of our initial JOVIAL efforts are as follows:

It is true that JOVIAL is adequate for the programming of any type of problem. It has been used for a wide variety of programs, and never has the language been shown to be insufficient.

The use of JOVIAL to program large systems has proven very valuable in the maintenance phase. For example, the ability of inexperienced or experienced compiler writers to maintain and improve the JOVIAL compilers has been considerably easier than the maintenance of similar programs coded in a less powerful language.

The use of one language for complete systems has eased the system integration problem considerably.

Although the areas of efficiency of output, speed of compilation, and reliability of compilers are something less than optimum with initial use of the compilers, a fairly rapid improvement in all of these can be noted after the initial large set of maintenance problems are overcome.

With one compiler we have witnessed an interesting trend. Once this compiler was operational, the improvement period was begun. Improvements were made both in the language it would accept and the efficiency of the machine language which it generated. Thus, a considerable amount of JOVIAL code has been added to the compiler. However, because of the improvements in efficiency of the output code, the number of machine instructions in the compiler is actually *less* than it used to be, and its operating time has improved somewhat.

Although the techniques used to program JOVIAL compilers have not been as ideal in application as they were in theory, a great deal of knowledge has been gained through our first experience, and these techniques show promise of improving considerably.

It might be added that these techniques have proven valuable. The amount of effort necessary to build compilers for more than one computer was considerably less than that required for the same tasks without the computer independent language and generator.

To summarize our experience, one might list the following axioms regarding the design of languages and the building of compilers for a language such as JOVIAL.

1. One frequently spends more time, money, and effort than he expected to get a little less than he hoped for from the *initial* version of a language and compiler of this magnitude.
2. Parallel to the compiler-building effort should be a test-building effort of considerable size. Letting the users find errors, or hoping there won't be many, are unsatisfactory techniques of checkout.
3. After the initial effort of producing a compiler is over, the work on improvements should continue indefinitely in order that all the original hopes and those which arise with experience can be realized.
4. It is possible to develop a general language which will replace machine language in almost all areas.
5. As time goes on, there is an increasing appreciation of a language such as JOVIAL and an increasing value derived from it. The initial set of problems are overcome fairly soon, and the values of a higher level programming language are truly realized.

GEOCOM: THE GENERAL COMPILER

C. Katz

General Electric Computer Department,
Phoenix, Arizona, USA

GECOM, The General Compiler for the GE-225 is not a new source language, but rather a compiling technique. It's source language is made up of four parts: ALGOL, COBOL, FRINGE, and TABSOL. The construction of the compiler is such that languages can be added, extended or removed.

ALGOL is a procedure oriented language designed to solve scientific problems. Although it possesses great power and flexibility in stating procedures, it is extremely weak in its data handling abilities. It has no provision for input/output, nor for the handling of variable length data. It insists that the data be either real or Boolean and of a fixed format.

COBOL is also a procedure oriented language. It was designed for the solution of business data processing problems, and the handling of large volumes of data. The data division of this language allows the user to describe in detail the make-up of his files, records, and fields. Data may be of variable length and format, and may be alphabetic or numeric. The input/output system supplies all of the housekeeping functions that deal with labels, fences, tape swaps, etc. COBOL, however, has a relatively unsophisticated procedure division. All of its statements are of the imperative type. It has no provision for handling algebraic expressions, Boolean expressions, functions, closed procedures, nor many of the other features in ALGOL. Whereas ALGOL and COBOL are each quite effective in the areas for which they were designed, they are each limited in scope. The combination of these languages, in GECOM contains many duplicate methods of accomplishing the same task, but also offers the ability to state many more types of problems strictly in source language.

FRINGE is a problem oriented language for sorting and merging of data, writing reports, and file maintenance. When supplied with a data description or report description and a few parameters, FRINGE will produce a tailored program to sort a file, produce a report, or update a file. FRINGE permits the

user to break into the generated routine at various points with his own coding should he wish to perform some computations or manipulate some fields, or in any way specialize the routine. Incorporating this system into GECOM permits the own coding to be done in one of the other source languages, rather than machine code, and thus supplies high level documentation. There are many who feel that the functions of sorting, reporting, and file maintenance constitute well over half the work that is done by a business data processing installation.

TABSOL is a tabular syntax that is convenient to use in problems that have many multiple decisions leading to a selected set of actions. It allows the user to lay out the control of his program in a concise, visual fashion. Although the same functions can be performed using the COBOL "if statements" or Boolean Algebra, it is many times more convenient to display this logic in a table. Any of the legal operations, names, or statements of the other GECOM languages may be used as entries in the table.

X =	A = B	C	Z =	GO TO
Y	T	= 0	0	23
Y	F	0	f(x)	ABC
0	T	0	g(x)	23

Fig. 1

Figure 1 shows a simple table. It is read as follows: If X = Y, and A = B is true, and C = 0, then set Z = 0 and go to the statement labeled 23. If any one of the conditions does not hold, go on and test the next row, etc. If no row of conditions can be met, go on to the next statement in sequence. The next statement may be another table or any other legal statement.

It might seem that combining all of these ingredients into a single system would be a monstrous undertaking; however, it clearly requires less effort than the duplication of building 2 separate compilers, one for business and one for scientific use. It is also a less formidable task than building these various parts separately and tying them together with an operating system.

It might also seem that the user who wished to use only the

COBOL or ALGOL portion would be taxed by the presence of the other portions and would waste a great deal of compiling time going through things that were not to his advantage. In order to overcome this difficulty, the compiler was built in a completely modular fashion. Only those portions of the language used in a given program contribute to the compiling of that program.

Figure 2 shows how the compiler works. Any combination of the four score languages may come in as the source program.

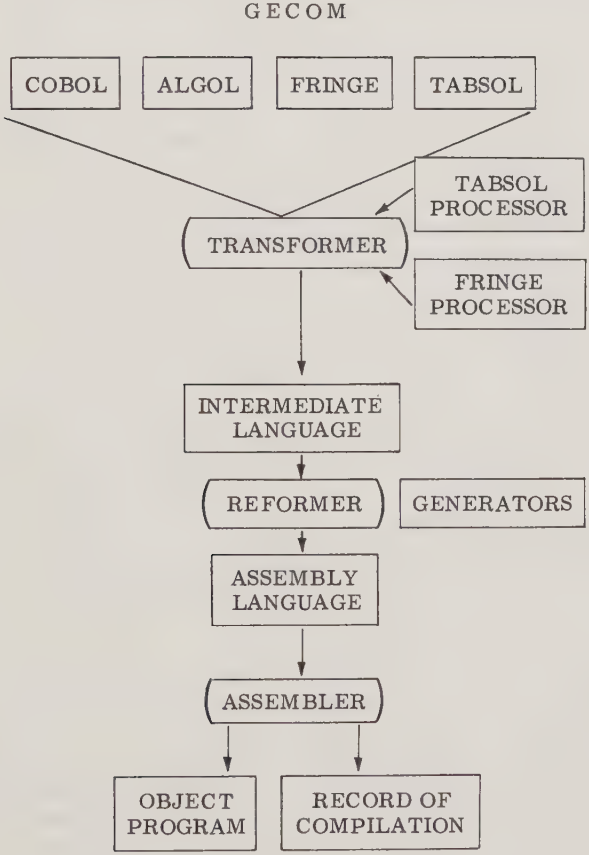


Fig. 2

The transformer decodes the source statements and correlates the related information from the various divisions into complete generator calls. If there are any tables or FRINGE statements, the appropriate processor will be called in. The reformer will then call in the necessary generators to produce assembly language coding. This is then assembled into a relocatable object program. A record of compilation is produced which gives complete documentation on source and object programs, and all of the cross references.

It is my feeling that one of the most important advantages of a compiler is the standardized documentation that it produces. The most difficult thing to have a programmer produce is good documentation of his program. This documentation is important because it permits future changes to be made to the program either by the same programmer or by someone else. It is also extremely important to the installation that is converting to new equipment. However, documentation whether programmer-produced or compiler-produced is useless unless it is kept up to date and reflects what is in the actual running program.

One of the most valid complaints made about compilers is that they are too slow in compiling. No one is really concerned about the length of time of the initial compilation. It is the time of recompilation that proves costly. Rather than spend all of this valuable machine time in recompiling programs as they are being debugged, the programmer will debug by patching the object program. This, of course, invalidates the documentation that he had. Clearly, debugging would be much more effective and efficient if all corrections were made at source language level. In order to make this economical, compilers must be faster than they have been in the past. The following are some of the techniques that were used in making GECOM an efficient compiler.

When building any large system, about half of it, usually the easiest part, is built to handle the normal or expected things. The difficulties in any large system and the things which normally slow it down are that hundreds of special cases must be considered. Most systems that I know of have been built equally around the normal and the special cases. GECOM has been built to efficiently handle the normal cases. Special cases are handled as peripheral. This means that the user pays in compiling time only for those special cases that occur in his program, not for the hundreds of others that could possibly occur. Most compilers of the COBOL type either impose limitations on the language due

to certain restrictions in the machine or else allow for all contingencies and overflow to tape or some other device for removing these restrictions. This technique leads into long tape searches or to the sorting of information housed on tape.

A technique used in GECOM which allows us to impose few restrictions and at the same time maintain fast compilation rates is an extremely simple one. We use what we call overflow runs. Unknown to the user there are a number of limitations imposed by the compiler. Should he exceed one of these limits, the compiler will automatically go through that run a second time. To illustrate: The compiler library is composed of 5 types of generators. In an early pass through the Procedure Division, calls for generators are separated by type. During the generation pass, we attempt to house all required generators of a given type in memory so that no tape search is necessary. Suppose there are 20 generators of Type A. We have made provisions to house as many as 10 of these in memory. Normally, fewer than 10 are required in any given problem. The calls for generators of Type A together with all the necessary parameters are passed through the computer while the generators are in memory, and all generation of that type takes place. Should a given problem call for 12 Type A generators, then 10 of these will be housed. The calls for these 10 will produce the necessary generated coding. Calls for the other 2 will be ignored. The pass will then be repeated with the other two generators being housed in memory and the calls once more passed. This same idea is carried out in the creation and use of many lists.

A unique search method is used throughout the compiler. It involves a first letter branch table which quickly locates the desired group of entries and eliminates all other possible groups. When the desired group is located based on the first letter, a subgroup is found by a similar technique on the second letter. This is continued until the desired entry is located. Judicious use of buffering eliminates wasted time in tape searching or tape rewinding. Files are always maintained in processing order. By use of the overflow technique described above, all table building and searching is entirely memory contained. The modular construction of the compiler is ideally suited for changing and updating as the various committees alter the standard languages.

GECOM is not a new language. It is a compiling technique. Its source language has been derived from the standard languages

produced by the ALGOL and COBOL committees and various extensions to these languages that we have felt necessary. Emphasis has been placed upon efficiency of compiling and techniques for producing efficient object programs. The basic compiler, the portion containing COBOL and ALGOL, was released on February 25, 1962. Prior to that time it had undergone a thorough field test at seven installations. Well over two hundred programs had been compiled. Compilation time ranged from four minutes to twenty five minutes.

THE COLASL AUTOMATIC CODING LANGUAGE*

Karl G. Balke and Glenn L. Carter

Los Alamos Scientific Laboratory,
Los Alamos, New Mexico, USA

INTRODUCTION

At the Los Alamos Scientific Laboratory of the University of California two somewhat similar automatic coding systems for scientific problems have been produced during the last few years. The MADCAP system, which generates codes to be run on the MANIAC computer, is the result of efforts primarily of Mark Wells and Donald Bradford (1). The COLASL system, which generates codes to be run on the IBM-7030 (STRETCH) computer, is the result of work done by the authors under the direction of Edward A. Voorhees. In both systems an attempt has been made to make the languages as natural as possible. The following criteria should clarify what is meant by a natural automatic coding language.

An automatic coding language may be called natural for two reasons. The first is that there is a probability close to one that a scientist who knows nothing about the language will automatically state his problem in it. The authors believe that it will be some time before such a language is implemented. The second reason for calling a language natural is that generally, once a problem has been stated in the language, any scientist who is able to read the literature pertaining to the problem can read the statement of the problem, even though he is totally unacquainted with the automatic coding language. This implies that the automatic coding language be a sub-set of the reflex language of the scientist using it, augmented by those elements of mathematical notation whose semantic content can be coded for a computer. In practice, some compromise with this ideal is necessary to permit implementation of the language before the computer for which it is intended is replaced by a newer model.*

*The authors sincerely hope that the development of UNCOL (2) will eliminate this problem.

INPUT HARDWARE

The first step in the design and implementation of an automatic coding language which can be called natural in this second sense is the procurement of a device which can produce two representations of the language: one, such as a deck of cards or a paper tape, which is easily read by a computer; the other which is easily read by humans. A combination of a card punch and an electric typewriter would satisfy both requirements.

In 1955, several RPQ's intended to explore this area were submitted to IBM. In a paper (3) dated July 16, 1956, Mr. Voorhees proposed a set of specifications for the type of machine desired. In these documents it was proposed that the typewriter have a somewhat expanded character set, and that there be keys on the keyboard whose effect would be the half-line rotation of the platen in either direction.

A more sophisticated set of specifications was prepared on January 29, 1959. These specifications were submitted to several manufacturers who were invited to bid. A contract to build a prototype was subsequently awarded to IBM. The resulting machine, known as the IBM-9210 Scientific Descriptive Printer (4), was accepted by the LASL in May, 1961.

This machine consists of an IBM-26 printing card punch, modified to punch more than three holes per card column, connected to a special IBM electric triple-case typewritter having a platen which may be rotated by half a line in either direction under keyboard control. The basic character set of the typewriter consists of the 132 characters in Fig. 1a. The type is so designed, however, that many more characters may be produced by printing one character over another. A few of the combinations having semantic significance in COLASL are given in Fig. 1b. Fig. 2 lists the carriage and card control functions which are available.

The characteristic of the IBM-9210 which is vital is that all actions taken by the keypuncher which affect the content of a problem listing which she is typing are recorded in a card deck. This deck can later be read by the IBM-9210 to produce a second copy of the problem listing identical to the original. A computer may be programmed to simulate the IBM-9210, constructing in its fast memory a representation of the page which may then be analyzed in some convenient fashion. In practice, a smaller unit of analysis, such as a line of the page, is more tractable. The

∩	⊂	←	#	*	^	|	[	>	]	%	√	(upper case)
∪	⊃	→	′	″	′	~	(	<	)	∞	▽	(lower case)
Λ	Ω	Γ	Φ	Θ	μ	_	{	0	}	○	□	(3rd case)
Q	W	E	R	T	Y	U	I	O	P	Δ		(upper case)
q	w	e	r	t	y	u	i	o	p	–		(lower case)
κ	ω	ϵ	ρ	τ	ψ	1	2	3	π	∂		(3rd case)
A	S	D	F	G	H	J	K	L	Σ	Π		(upper case)
a	s	d	f	g	h	j	k	l	+	×		(lower case)
α	σ	δ	φ	γ	η	4	5	6	∫	°		(3rd case)
Z	X	C	V	B	N	M	:	$	/			(upper case)
z	x	c	v	b	n	m	,	=	?			(lower case)
ζ	χ	ξ	λ	β	ν	7	8	9	.			(3rd case)

FIG. 1a. The 132 characters printable by the IBM-9210 Scientific Descriptive Printer.

≡ ‡ ; ≥ ⊁ ≠ ≤ ∤ ∓ ÷

FIG. 1b. Common combination characters of the IBM-9210 Scientific Descriptive Printer.

Carriage return	Red
New page	Black
Backspace	Read card eject
Tab	Card skip
Tab set	Stop read
Tab clear	Void
Main up	Space
Main down	

FIG. 2. Carriage control functions of the IBM-9210 Scientific Descriptive Printer.

line is a particularly attractive unit in that if a keypuncher follows one simple rule, then it is very easy for the translator to know when a line is complete. The rule is that a carriage return shall not be used during the typing of a line, and shall always be used at the end of a line.

COLASL INPUT FORMAT

In its gross format, the page upon which a problem is typed is divided into four vertical bands. The left margin is of arbitrary width, and is of no concern to COLASL, since nothing can be typed in it. The second band is one type column wide, and is called the *class column*. The next eight columns comprise the

name field. The remainder of the page is called the *statement field.* This format was chosen so that the programmer might interject statements in the STRAP (STRetch Assembly Program) vocabulary into a COLASL code if he felt the need of facilities of the machine (such as variable field length arithmetic) which are not accessable using the COLASL language.

The class column is not used in the COLASL language itself except for a pair of characters which may be written therein to cause the COLASL translator to refrain from examining the material between them. Any other character in the class column is attached to the first instruction generated for the next statement to be coded. Such class column entries are intended to be used in connection with STRAP instructions.

The name field is used to attach labels to selected sentences so that they may be referred to from other areas of the problem. The name of a particular sentence is written in the name field to the left of the sentence to which it applies.

The sentences which define the computation which the programmer wants STRETCH to perform are written in the statement field.

SENTENCES AND SENTENCE UNITS

The sentence structure of COLASL is based upon that of reflex English, except that the method of naming sentences in COLASL is more stylized. It seems natural, then, to use some of the terms of English grammar to describe the COLASL syntax. Where discrepancies arise between the ordinary meaning of such words and the meaning intended in this paper, the latter will be explained.

The basic unit of information in the COLASL language is the *sentence*. Although a sentence in COLASL does not necessarily need to be capitalized, it must be terminated by a period, exclamation point, or question mark. Which of these characters the programmer chooses to employ is of no consequence to the interpretation of the contents of the sentence by the translator; however, proper choice of punctuation may make the code listing more readable. Only sentences may be named.

A sentence may be divided into smaller units called *clauses*, which may be separated from each other by colons, semicolons, or structures which will be defined later as antecedents. A

sentence contains at least one clause, with which it may be identical.

A clause may be divided into smaller units called *phrases*, which may be separated from each other by commas. A clause contains at least one phrase, with which it may be identical. Clearly, a phrase may be a clause which may be a sentence. Clauses, phrases, and antecedents are called *sentence units*, and the various statements which may be written in the COLASL language will be defined in terms of the smallest sentence units which may contain them.

ALPHABET

The alphabet upon which COLASL is based is given in Fig. 3. It is important to note the large number of letters which may be constructed by using letter modifiers in conjunction with basic letters. This large literal alphabet makes possible a simplification of the form of variable names, which simplification in turn facilitates the inclusion of implicit multiplication in the algebraic sub-set of the COLASL language. The construction of labels from this alphabet is the next topic to be discussed.

VARIABLE NAMES

The first label is the *variable name*. A variable name must be of the form:

$$\delta L_{Sss \ldots s}$$

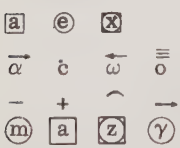
In this form δ is null or represent one of 'Δ' and '∂'. L stands for any letter in the extended sense of the word, and S is null or represents an extended letter. If S is null then the string of s's is null; otherwise, each s represents an extended letter or a digit, and the string of s's is of at least unit length. The total length of any variable name must be eight or fewer columns. If S is not null, then the subscripts on L is called a *descriptive subscript*, as contradistinct to a variable or constant subscript which might be applied to a variable name if it were the name of a vector or matrix. Blank columns in a variable name are not allowed, except that the descriptive script, if any, may be separated from the rest of the name by their use. Examples of variables names are given in Fig. 4.

Basic letters:
a through z
A through Z
α β γ δ ϵ ζ η κ λ μ ν ξ
π ρ τ φ χ ψ ω
Γ Θ Λ Φ Ω

Superprinting characters:
$\bigcirc$ $\square$

Overprinting characters:
+ − × ° = * ′ ^ ~ → ← ‘
” , : .

Extended letters formed from the above:



Digits:
0 1 2 3 4 5 6 7 8 9

Punctuation marks:
: ; , . ? !

Algebraic symbols:
+ − × ° / = ≡ > < √ ÷ ± ≥ ≯ ≠
≤ ≮ ≠

Special symbols:
Δ ∂ # ℑ

Grouping symbols:
( ) [ ] { } |

Miscellaneous:
∩ ∪ ⊂ ⊃ % ∞ ∇ ∫ $

Reserved for future use in the language:
Σ Π

FIG. 3. The COLASL alphabet.

FIG. 4. Examples of variable names in the COLASL language.

In less precise language, a variable name is basically a single letter, which may be prefixed by 'Δ' or '∂', and which may have a descriptive subscript. This convention has been adopted because it seems that it is common in manuscripts and publications that variables are named by single letters.

SENTENCE NAMES

The second label in the COLASL language is the *sentence name*. A sentence name may have a total length of eight or fewer columns, and may be of one of the following three forms:

#dd . . . d

Laa . . . a

Laa . . . a_{dd} . . . d

In the first form # represents itself. In the first and third forms, each d is null or stands for a digit, and the strings of non-null d's must be of at least unit length. In the second and third forms, L stands for a letter in the extended sense, and each a is either null or represents an extended letter or digit. Blank columns are not permitted within a sentence name. The second form of sentence name is a sub-case of the third form. That part of a name of the third form excluding the numeric subscript is called the *base name*. Examples of sentence names are given in Fig. 5.

CONSTANTS

A numeric *constant* may be considered as a third type of label, which differs from a sentence or variable name in that it defines

Type 1:			
#1	#0198	#0000001	#9786491
Type 2:			
a	ⓐdep	Temp	ⓒ100
x10A	$\overline{\omega}$32b		
Type 3:			
$Zilch_{10}$	$A1100_3$	$T943_{001}$	X_1
X_2	X_3	X_{05}	X_{763}

FIG. 5. Examples of sentence names in the COLASL language.

the object that it names. All constants written in the COLASL language are of the form:

σdd...dpdd...d

where σ is null or represents one of '+' and '−', each d is either null or stands for a digit, and p may be null or a decimal point. There must be at least one non-null d, and the maximum number of non-null d's is 15.

The form of a constant is not as resrictive as it may appear to be at first thought. A constant which can not be expressed in this form can always be written as an expression in constants, as is commonly done in so-called scientific notation. No inefficiency is introduced into the object code by this device, since the COLASL translator collects constants wherever possible.

There are two variable names in COLASL which ordinarily represent constants. They are 'π' and 'e', which have the values (to an accuracy of 14 decimal digits) normally associated with them in mathematical literature. The programmer may also cause other arbitrary variable names to represent constants, by a process to be described later. Variable names which represent constants are called *parametric* variable names, and may be used in any context where a constant is legitimate. Examples of constants and expressions thereof are to be found in Fig. 6.

FUNCTION AND ROUTINE NAMES

A fourth type of label is the *function* or *routine name*. Function and routine names are of the form:

Laa...a

where L represents a letter in the extended sense, and each a is null or stands for an extended letter or a digit. Function and routine names must be eight or fewer columns in length. They may be considered as special cases of sentence names.

3	4.	5.326
$\frac{4\pi}{3}$	7.62×10^{-18}	e^{π}
3076	.2765	372.9

FIG. 6. Examples of constants in the COLASL language.

Some function names are standard members of the COLASL language. The elementary mathematical functions, such as sin, cos, ln, etc., are of this class. The privileged position which these function names occupy in the language arises from the existence in the library of pre-coded subroutines to evaluate the functions designated by the names. The standard names may ordinarily be spelled with any of these varieties of capitalization: each letter of the name may be capitalized; only the initial letter may be capitalized; or each letter may be in lower case. Examples of function and routine names are given in Fig. 7.

WORDS

The fifth and last type of label is the *word*. Words are of the form:

Lll...l

where L represents a letter, and each *l* is null or stands for a letter. Some words have special significance to the COLASL translator when they occur in the proper contexts. These words are called *key words*. Other words may later become key words as the language expands. Key words enjoy the same flexibility in their capitalization as has been attributed earlier to standard function names. If the singular form of a noun is a key word, then so also is the plural. Lower case spellings of key words, plurals excepted, are listed in Fig. 8.

Except where appearing in the name field or within an algebraic expression, where special rules apply, wherever a label

sin	cos	arctan
gd	[X] max	z32[π]

FIG. 7. Examples of function and routine names in the COLASL language.

attach	if
end	range
exit	routine
for	stop
from	then
function	through
go	to

FIG. 8. Alphabetical list of COLASL key words.

is used it must be separated from its context by blank columns or by proper punctuation, which latter includes grouping symbols and the characters, '.', '?', '!', ':', ';', ',', '', '"'. This rule is intended to reflect the common practice of people who write in English, among other languages.

EXPRESSIONS

Having defined constants, and variable and function names, it is now possible to present a constructive, recursive definition of what is meant by an *expression*. To do this, five types of sub-expressions, to be called A-expressions, C-expressions, M-expressions, R-expressions, and S-expressions, shall be defined. To make all these definitions intelligible, some remarks on notation are in order.

In what follows, the upper case letters, A, C, M, R, and S, and their subscripted variants, will be used to represent A-expressions, C-expressions, M-expressions, R-expressions, and S-expressions respectively. The lower case Roman letters c, f, and v, and their subscripted variants, will be used to denote constants, function names, and variable names, respectively. The operation, punctuation, and grouping symbols (cf., Fig. 3) will represent occurrences of themselves. The character σ will have the following complicated meaning: whenever it follows a punctuation symbol or '+' or '−' directly, except for possible intervening blank columns, or whenever it is the leftmost character in its context, it may be null or denote one of '+', '−'; otherwise it denotes one of '+', '−'.

The characters μ and ν will always appear in sub-expressions in pairs. They will have the following elaborate characteristics: when an occurrence of μ follows another occurrence of μ directly, except for possible intervening blank columns, then in its leftmost occurrence μ is null, and its corresponding ν is null; when an occurrence of ν precedes another occurrence of ν directly, except for intervening blank columns, then in its rightmost occurrence ν is null, and its corresponding μ is null; in all other cases, μ and ν represent '|'.

Finally, the relationship between symbols in the present metasyntax represents the relationship between entities in the syntax, except for possible intervening blank columns: e.g.,

$\lceil \overline{M_1 M_2} \rceil$* is the concatenation of $\lceil \overline{M_1} \rceil$ and $\lceil \overline{M_2} \rceil$, and $\lceil \overline{v_S} \rceil$ is $\lceil \overline{S} \rceil$ written below and to the right of 'v'.

First, the possible forms of the S-expression may be defined. 'σv', 'σc', and 'σvσc' are S-expressions. $\lceil \overline{\sigma v S} \rceil$ and $\lceil \overline{\sigma v S \sigma c} \rceil$ are S-expressions. $\lceil \overline{S_1, S_2} \rceil$ is an S-expression.

The definitions of the rest of the possible sub-expressions proceed similarly. 'v' and 'c' are M-expressions. $\lceil \overline{v_S} \rceil$ is an M-expression. $\lceil \overline{M_1 M_2} \rceil$, $\lceil \overline{M_1 \times M_2} \rceil$, $\lceil \overline{M_1 \circ M_2} \rceil$, $\lceil \overline{M_1}^{M_2} \rceil$, $\lceil \overline{M_1 / M_2} \rceil$, and $\lceil \overline{M_1 + M_2} \rceil$ are M-expressions. If σ is not null, then $\lceil \overline{\sigma M} \rceil$ is an A-expression. $\lceil \overline{M_1 \sigma M_2} \rceil$ is an A-expression. $\lceil \overline{A_1 \sigma A_2} \rceil$ is an A-expression. $\lceil \overline{M} \rceil$ and $\lceil \overline{A} \rceil$ are C-expressions. $\lceil \overline{[C]} \rceil$, $\lceil \overline{\{C\}} \rceil$, $\lceil \overline{(C)} \rceil$, and $\lceil \overline{\mu C \nu} \rceil$ are M-expressions. $\lceil \overline{\frac{C_1}{C_2}} \rceil$, $\lceil \overline{\sqrt{C}} \rceil$, $\lceil \overline{\sqrt[v]{C}} \rceil$, and $\lceil \overline{\sqrt[c]{C}} \rceil$ are M-expressions.

At this point, attention must be given to certain geometric implications of the last four M-expressions defined. In the first of these, the underbar which separates the symbols representing component contexts consists of two underscore characters. In an instance of this context, a sufficient number of underscore characters must be provided to "cover" both the upper and lower M-expressions. In instances of the other three contexts, the radical sign must be made large enough to contain the enclosed C-expression completely. Furthermore, the particular variable name or constant represented by 'v' or 'c' must be physically small enough to fit in the available space. For reasons of expediency in the design of the COLASL translator, the arbitrary size limit of one character has been imposed upon these labels.

An R-expression may be null. If not, $\lceil \overline{(C_1, C_2, \ldots, C_m)} \rceil$ is an R-expression. In a primitive sense, subject to later qualification, $\lceil \overline{fR} \rceil$ is an M-expression. An *expression* is a largest instance of an A-expression or an M-expression: i.e., an instance which is not itself contained in a larger instance of an A-expression or M-expression.

As the examples of expressions given in Fig. 9 illustrate, the COLASL expression syntax is almost identical to that of usual classical algebraic notation. An expression in the COLASL language is translated into a machine code which, when executed, will produce the same number for the value of the expression as

*The corner notation is described in (6).

Addition:

$a + b$ $a - b$

Multiplication:

ab $a \times b$ $a \cdot b$

$(b + c)(d + f)$

Division:

$\frac{c + d(e + f)}{ghi(k - 1)}$ $\frac{b + c}{d} \div \frac{e + f}{g}$

$4ac/3\pi\omega$

Exponentiation:

y^3 y^{z+3} e^{z^2}

$\lambda\mu^{\frac{x + y}{x - y}\frac{y + z}{y - z} + 3}$

Radication:

$\sqrt{b}$ $\sqrt[9]{b}$ $\sqrt[n]{b}$

$\sqrt[n]{\frac{b + c}{d - e}}$

Function calling:

sin (y) Cos (y)

$\text{EXP}\left[\frac{\sin(\boxed{x}_i)}{\cos\left(\frac{1}{\boxed{x}_j}\right)}\right]$

Absolute value indication:

$|a| - |b|$ $\frac{\partial\tau}{\sigma\tau}\left|\frac{x_{i-1} - x_{i+1}}{2}\right|$

$|(|a| - |b|)|$

Subscripting with constants:

x_0 x_{25} $P_{temp_{-3}}$

$x_{1,5}$ $P_{temp_{15,-1}}$ $x_{-5,-10}$

$P_{temp_{6,4,2}}$ $x_{1,0,-1}$ $P_{temp_{-2,-8,-16}}$

Subscripting with variables:

x_i x_j $P_{temp_{-i}}$

$x_{i,j_{sub}}$ $x_{i,-j}$ $x_{-i_{sub},-j}$

$x_{i,j,k}$ $P_{temp_{i,-j_{sub},-k}}$ $x_{-i,-j,-k_{sub}}$

FIG. 9. (continued)

Subscripting with variables and constants:

$P_{temp_{i,2}}$ $x_{2,i}$ $x_{-21,-i\ sub,j}$

$x_{i\ sub\ +\ 2}$ $P_{temp_{-i+4}}$ x_{j-5}

$P_{temp_{-k-12}}$ $x_{i+2,\ j+4}$ $x_{-i-4,\ k\ sub-3}$

$P_{temp_{i+1,\ j+2,\ k\ sub}+3}$

Subscripting with subscripted variables:

$x_{P_{temp_{i,j}}}$

$P_{temp_{x_{P_{temp_{i,j}}}}}$, $P_{temp_{j,1}}$, $x_{P_{temp_{j,i}}}$, $P_{temp_{i,j}}$

FIG. 9. Examples of various expressions in the COLASL language.

a human computer might be expected to produce if he worked to the same significance as the machine.*

COLASL CONTEXTS: RANGE SENTENCES

The first context of the COLASL language is the *range* sentence which the programmer must use to specify, for each dimensioned variable (i.e., vector or n-dimensional array), the number of dimensions and the maximum and minimum of the possible subscript values of each dimension.

The range sentence must have at least one clause which contains one of the words 'range' and 'ranges', and all the rest of its clauses must be of the form:

m_1 to n_1, m_2 to $n_2, \ldots,$ m_p to n_p for $v_1, v_2, \ldots,$ and v_1

where the subscripted variants of m and n represent constants or parametric variable names (q.v.), and the subscripted variants of v have their usual meaning. The word 'and' may be omitted.

The examples of range sentences given in Fig. 10 are interpreted in the following manner. The first example defines x as a vector of 28 elements: $x_0, x_1, \ldots, x_{27}$. The second example specifies that x, $\boxed{y}$, and z_{min} are distinct three-dimen-

*The ambiguity of constructions like 'v_1/v_2v_3' and '$v_1/v_2/v_3$' has been removed by interpreting the former as $v_1v_2^{-1}v_3$ and the latter as $v_1v_2^{-1}v_3^{-1}$; that is, by interpreting '/' and '+' as reciprocal multiplication.

A range is: 0 to 27 for x.

The ranges of our variables are: 0 to 4, −5 to 6, and 3 to −10 for x, $\boxed{y}$, and z_{min}.

First ranges are: −5 to 7 for y, z_{max}, and q; 3 to −5, 8 to 7, and 5 to 10 for z_{min}: second ranges are: −10 to 10 for λ, ξ, and μ.

FIG. 10. Examples of range sentences.

sional arrays having 840 elements each. In the case of the x-array, the elements are: $x_{0,-5,-10}$, $x_{1,-5,-10}$, . . . , $x_{4,-5,-10}$, $x_{0,-4,-10}$, $x_{1,-4,-10}$, . . . , $x_{4,-4,-10}$, . . . , $x_{0,6,-10}$, $x_{1,6,-10}$, . . . , $x_{4,6,-10}$, . . . , $x_{0,-5,-9}$, . . . , $x_{4,6,3}$. An enumeration of the elements of the $\boxed{y}$ or z_{min} arrays would be similar. The third example defines: y, z_{max}, and q as distinct vectors of 13 elements each; z_{min} as a three-dimensional array of 108 elements; and λ, ξ, and μ as distinct vectors of 21 elements each.

The three sentences in Fig. 10 could not be written in one problem, because the dimensionality of the variable x is defined in both the first and second examples, and the dimensional properties of y and z_{min} are defined in both the second and third examples. Except that multiple definition of this type is not allowed, the number and placement of range sentences are at the discretion of the programmer.

ALGEBRAIC EQUATIONS

Another important context of the COLASL language is the *algebraic equation*. Let E represent an arbitrary expression, and let all the notation used in defining expressions be retained. Then an algebraic equation is a structure of the form:

$$\lceil \overline{v_1 = v_2 = \ldots v_n = E} \rfloor$$

where the variables denoted by the v_i are called the *defined variables* and $\lceil \overline{E} \rfloor$ is called the defining expression. The smallest sentence unit in which an algebraic equation may occur is the phrase, and this phrase may contain no other material following the defining expression. A phrase containing an algebraic equation may not be bounded on the right by an antecedent.

PARAMETRIC EQUATIONS

Let E′ represent any expression restricted as follows. $\lceil \overline{E'} \rfloor$ may contain no S-expressions; in $\lceil \overline{E'} \rfloor$ only standard function

names may appear; each variable whose name appears in $\lceil \overline{E'} \rceil$ must have the special properties to be explained below. Let all other notations be used according to prior definition; then a *parametric equation* is a structure of the form:

$$\lceil v_1 \equiv v_2 \equiv \ldots \equiv v_n \equiv \overline{E'} \rceil$$

where the variables denoted by the v_i are called *defined parametric variables* and $\lceil \overline{E'} \rceil$ is called the *defining expression.* The smallest sentence unit which may contain a parametric equation is a phrase, which may contain no other material following the defining expression, and which may not be bounded on the right by an antecedent. The special properties which variables whose names appear in the defining expressions of parametric equations must have are that they must all be parametric defined variables, and that they must all be defined ultimately in terms of constants and functions of constants alone.

Examples of algebraic equations are given in Fig. 11a; examples of parametric equations in Fig. 11b.

SEQUENCING

It is necessary here to fix some ideas about two important sequences which exist within the COLASL language. The first is the syntactic sequence of the language, in which contexts and sentence names follow each other on the page. This is the common reading sequence which orders the material on a page from left to right, top to bottom. The second is the logical sequence in which the code corresponding to the names and contexts of the language will be executed. These sequences are normally identical; however, there are a number of contexts which can alter the logical sequence according to the programmer's

$$x = y = c - 21.$$

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}.$$

FIG. 11a. Examples of algebraic equations.

$i \equiv g - 21.$ $\qquad$ $j \equiv \sin(i) - \cos(g).$
$g \equiv 4.$

FIG. 11b. Examples of parametric equations.

purposes. For expository simplicity, the contexts and sentence names of the COLASL language will be said to have a logical sequence which is identical to the sequence of execution of the code to which they give rise.

A COLASL program will generally consist of several principal logical units: exactly one *main program*; and some number of sub-programs called *functions* and *routines*. The problem always begins and usually ends in the main program, which can treat its concomitant sub-programs as elements of its sequence as indicated by the programmer. A sub-program can, in turn, treat another sub-program as an element of its sequence. The logical sequence of sub-programs is called the global sequence of the problem. The logical sequence within a main program or any individual sub-program is called the local sequence of that main program or sub-program. The effects of various contexts on local sequences will be discussed next.

Classified according to their effect on the local sequence, there are three main types of contexts in the COLASL language. The first type defines storage or furnishes information to the translator, and, except for any indirect effect it may have thereby, it affects the sequence of the program to the same degree as does an extra blank column between sentences: i.e., as though it did not exist. Of the contexts discussed so far, the range sentence and parametric equation are of this type. The second type of context is incorporated into, but does not change, the sequence of the program. The algebraic equation is such a context. The third type of context is incorporated into, and changes, the sequence.

BRANCHES

The first example of this latter type of context is the *branch*. Let s represent an arbitrary sentence name. Then a branch is of one of the forms:

'go to s'
'from s'.

The minimum sentence unit which may contain a branch is a phrase. If a phrase contains a brahch of the second form, then it may not also contain a structure to be defined later as a quantifier. A sentence may contain at most one branch.

The branch alters the logical sequence of a program by making the sentence whose name it contains follow imediately the sentence in which it occurs. Within a sentence, a phrase containing a branch may not precede an antecedent; however, a branch may precede any other sequence of contexts, since the effect of a branch on the logical sequence of a program is deferred to the end of the sentence in which it occurs. A branch which contains the name of some sentence will be said to *reach* that sentence.

Examples of branches are given in Fig. 12. In the first example, when the generated code is executed, the branch will result in an infinite repetition of the evaluation of z and p. In the second example, the defined variables of the equations will be evaluated in the order: x, z, p, q, j, and k.

TRANSFER VECTORS

There is a special case in which a branch may reach one of several sentences. The names of all of these sentences must constitute a *transfer vector*; i.e., they must all be of the third form of sentence names, and must all have the same base name. Let s be the base name of a transfer vector, and let S′ represent an S-expression which has no commas on its first level, and which has a variable name on its first level. Then the special case of the branch is of one of the forms:

$y = y + 2$

#1 $z = x + \frac{y}{2} + p.$

$p = 5z.$

Go to #1.

$x = y + 2$, from #2.

#1 $p = 5z.$

$q = 3zp.$

From #3, $j = \frac{x}{4y}.$

#2 $z = x + \frac{y}{2} + p$, go to #1.

#3 $k = j.$

FIG. 12. Examples of branches.

⌈go to sS'⌉

⌈from sS'⌉

which may be written in a phrase restricted as though it contained an ordinary branch.

Where the sentence reached by an ordinary branch is unique, the sentence reached by a branch to a transfer vector is a function of the value calculated for ⌈S'⌉ at the end of the sentence in which the branch occurs. If an element of the transfer vector has a numeric script whose value is the greatest integer in the calculated value of ⌈S'⌉, then the sentence whose name is that element is reached by the special branch.

An example of a special branch is given in Fig. 13. In this example, the branch reaches I_1 if $1 < i < 2$, it reaches I_{02} if $2 < i < 3$, and it reaches I_4 if $4 < i < 5$. For all other values of i the branch is undefined.

LOOPS AND QUANTIFIERS

A *loop* is a set of contexts which is to be repeated as a unit for a set of values of a variable called the *quantified variable*. Each time the code corresponding to the contexts of a loop is executed, the loop is said to have been *traversed*. A set of consecutive traversals of a loop is called a *cycle*. Let the subscripted variants of E represent unspecified expressions, and all prior notation be retained; then a *quantifier* is a context of the form:

⌈$(v = E_1, E_2, \ldots, E_3)$⌉

where v is the quantified variable, and the ellipsis is part of the syntax. The number of traversals of a loop in a cycle and the value of the quantified variable during each traversal are specified in the following way by the quantifier associated with a loop.

Go to I_i.

I_1 $x_{i+5} = x_{i+5} - 1/y_i$.

I_{02} $x_{i+4} = x_{i+4} - 2/(y_i + 1)$.

I_4 $x_{i+3} = x_{i+3} - 3/(y_i + 2)$.

J_3 $x_i = x_{i+3}$.

FIG. 13. Examples of a branch to a transfer vector.

Let e_1 be the value of $\lceil \overline{E_1} \rceil$, e_2 be the value of $\lceil \overline{E_2} \rceil$, and e_3 be the value of $\lceil \overline{E_3} \rceil$, all just prior to the first traversal of a cycle. Let q_i be the value of v on the i^{th} traversal. In the metasyntax, if x is any real number let $[x]$ be the integer part of that number as produced by discarding the fraction, if any, so that $[2] = 2$, $[-2] = -2$, $[3.427] = 3$, and $[-3.427] = -3$. Let n be the number of traversals of the cycle, and let Δ be the difference between q_{i+1} and q_i. Let n′ be defined by the equation:

$$n' = \frac{[e_3] - [e_1]}{[e_2] - [e_1]} + 1.$$

Then

$$\Delta = [e_2] - [e_1].$$

$$n = \left\{ \begin{matrix} [n'] \text{ if } n' \geq 1 \\ 0 \quad \text{if } n < \end{matrix} \right\} \text{ and } \Delta \neq 0.$$

$$q_i = [e_1] + (i - 1)\,\Delta, \quad (i = 1, 2, \ldots, n).$$

INTERNAL LOOPS

The set of contexts which comprises a loop may be specified in one of three ways, depending upon the context in which the quantifier associated with the loop is written. In an *internal loop* the quantifier appears in a phrase containing neither of the words 'through' and 'from'. The contexts which comprise an internal loop are all and only those which exist in the same sentence with the quantifier defining the loop and are logically prior to it. Examples of internal loops are given in Fig. 14. In all examples of loops, each loop is enclosed in a box with the context defining it.

EXTERNAL LOOPS

Let Q represent an unspecified quantifier and s represent an unspecified sentence name. Then $\lceil$through s $\overline{Q}\rceil$ and $\lceil \overline{Q}$ through $\overline{s}\rceil$

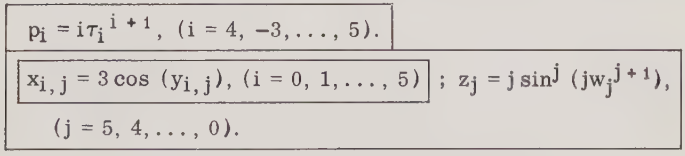


FIG. 14. Examples of internal loops.

are contexts which define *external projective loops*, and ⌈from s Q⌉ and ⌈Q from s⌉ are contexts which define *external reflexive loops*. These contexts may appear in phrases. For 'through' the variant spelling, 'thru', may be substituted.

The contexts comprising an external projective loop are all those logically subsequent to the defining context up to and including the sentence named 's'. The contexts comprising an external reflexive loop are all those logically prior to the defining context up to and including the sentence named 's'. Examples of external loops are given in Fig. 15.

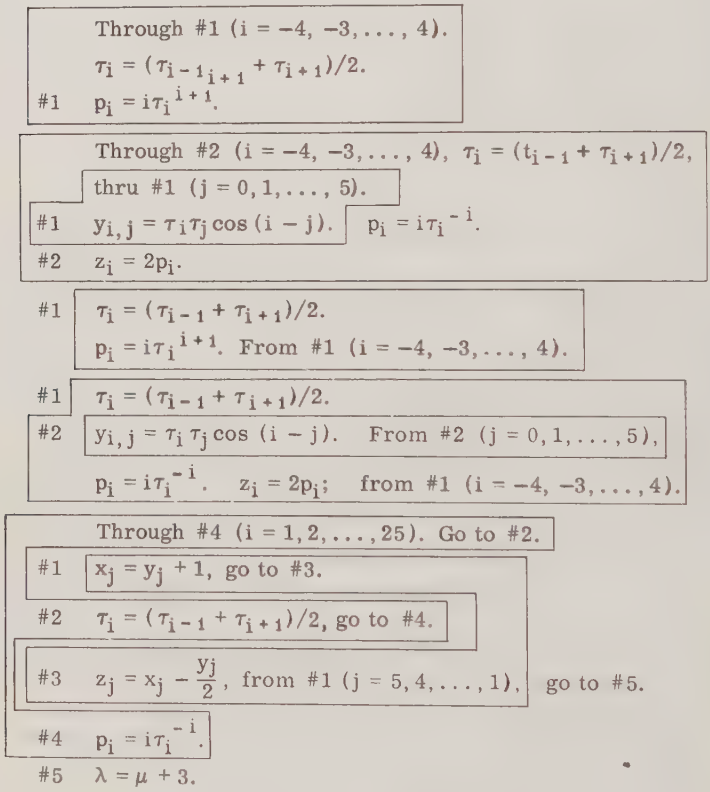


FIG. 15. Examples of external loops.

ANTECEDENTS

The context known as an *antecedent* has been mentioned several times heretofore. An antecedent conditionally alters the sequence of a program. To define the syntax of an antecedent, let each subscripted variant of i be null or represent one of '$=$', '$\neq$', '$>$', '$\not>$', '$\geq$', '$\not\geq$', '$<$', '$\not<$', '$\leq$', and '$\not\leq$'. Also, let each subscripted variant of E be null or represent an arbitrary expression, and let E_j be null if and only if i_j be null for all positive integral j. Then

$$\ulcorner E_0\ i_1\ E_1\ i_2\ E_2\ i_3\ E_3 \ldots \urcorner$$

is called an *inequality* only if E_0 and E_1 are not null. Let the subscripted variants of I be null or represent arbitrary inequalities, let the subscripted variants of o be null or represent one of ',', ';', ':', 'or', ', or', '; or', and ': or', and let I_j be null if and only if o_j is null for all positive integral j. Then

$$\ulcorner \text{if } (I_0\ o_1\ I_1\ o_2\ I_2\ o_3\ I_3 \ldots) \text{ then} \urcorner$$

is called an *antecedent*.

An antecedent alters the sequence of a program depending on the truth value of a statement which is equivalent up to notation to a statement in the propositional calculus in conjunctive normal form, whose sentential variables are algebraic inequalities. If the statement is true when encountered in the logical sequence, then the subsequent contexts in the sentence in which the antecedent occurs are next in that sequence. Otherwise the first context of the sentence following that which contains the antecedent is next.

From a more intuitive view, the interpretation of an antecedent is intended to conform to what most people mean when they say "if ... then ...". Examples of the use of antecedents are given in Fig. 16. These examples should be self-explanatory.

STOP

If a phrase contains 'stop' then the logical sequence of the program is terminated with the last context of the sentence in which the phrase occurs. Fig. 17 has an example of the use of 'stop'.

If $(x > 2)$ then $y = 3x$, $i = j = 2$.
$z = 5y$.

If $(p = 3)$ then $x_i = Py_j$; if $(x_i \geq 5c)$ then
$y_j = 3c$; go to $\alpha 4$. $z_i = y_j - x_j$.

$\alpha 4$ $i = j = 2$.

If $(x > y > 3)$ then go to #1. If $(x \neq y \neq 4)$ then go to #2.

#1 $x = y^3 + 4$.

#2 $z = xy^2 - 3$.

If $(x > y$ or $y > z)$ then $p = q + z$.

If $(x > y$, $y > z$, or $p \neq 3)$ then $z = x^2 + y^2 - p^2$.

If $(x > 2)$ then if $(p = 3)$ then $x_i = Py_i$.

If $(x > 2)$ and $(p = 3)$ then $x_i = Py_i$.

FIG. 16. Examples of COLASL antecedents.

If $(x > 2)$ then stop.

Stop after setting $z = 4\pi r^3$.

FIG. 17. Examples of stop.

FUNCTION AND ROUTINE CALLING

There are two contexts which appear in a local sequence, but affect the global sequence of a program. The first of these has already been mentioned in defining expressions: it is the *function calling context* $\lceil \overline{fR} \rfloor$. The syntax of the second context is similar, but it must appear under different circumstances. Let r represent some unspecified routine name. Then $\lceil \overline{rR} \rfloor$ is a *routine calling context.* This context may appear in a phrase, where it must appear under the same circumstances as would a key word. Whenever a calling context appears in a logical sequence, the code corresponding to the called sub-program is executed, after which the original sequence is resumed.

Both functions and routines in the COLASL language are functions in the mathematical sense. The difference between them is that COLASL functions are single-valued mathematical functions, while COLASL routines are multi-valued functions. This difference explains why only functions may be called within expressions: the COLASL equation syntax has no provision at present for true matrix notation. Both functions and routines may be functionals.

Functions and routines may or may not need explicitly stated arguments. In the case of a pseudo-random number generator, for example, the only argument is the number of pseudo-random numbers of the chain which has been generated so far; the nature of a pseudo-random number generator is such that this datum is not supplied by the calling context, in which, therefore, ⌈R⌉ is constrained to be null. As another example, a sine function generator must have just one argument supplied by the calling context: the angle whose sine is to be computed. Hence, in the calling context ⌈sinR⌉, ⌈R⌉ is constrained to have no commas exterior to its component C-expression.

From the above remarks it may be correctly surmised that, in any instance of a calling context, the form of the R-expression is influenced by the nature of the function or routine being called. The form of a non-null R-expression may vary in two ways: the number of arguments, i.e., the number of component C-expressions, may change; and the form of any individual argument may be subject to constraint. The variations in form of R-expressions are intimately bound to the definition of the functions with whose names they appear in calling contexts. Hence the techniques by which the programmer may define functions and routines will be explained next. Unless specifically mentioned, standard functions are excepted from the following discussion.

FUNCTION AND ROUTINE NAMING BY THE PROGRAMMER

In the syntactic sequence of a program a function or routine name must be defined as such and the number and general type of its arguments stated through the use of a *function* or *routine naming context* before the name may appear in a calling context.

Let n and each of its subscripted variants represent one of '0', '1', . . . , '9', '10', . . . , '100', ... Let a be null or represent one of 'v', '$\exists^{n}$f', or '$\exists^{n}$r'. If a is null, let all subscripted variants thereof be null also; otherwise let each subscripted variant a_i be null, or represent one of ', v_i', ', $\exists^{n_i}f_i$', or ', $\exists^{n_i}r_i$', for positive integral i. Finally, if a is null, let λ and ρ be null; otherwise let λ represent '(' and ρ represent ')', or λ represent '[' and ρ represent ']', or λ represent '{' and ρ represent '}'. Then $\lceil \lambda a\ a_1 a_2 \ldots \rho \rceil$ is an R′-context.

Let R′ represent some R′-context. Then a *function naming context* is a sentence of at least two clauses, at least one of

which contains one of the words 'function' and 'functions' and no other COLASL context, and all phrases of all the rest of which contain precisely $\lceil f R' \rceil$, where 'f' is distinct from any function name appearing in $\lceil R' \rceil$. Similarly, a *routine naming context* is a sentence of at least two clauses, at least one of which contains one of the words 'routine' and 'routines' and no other COLASL context, and all phrases of all the rest of which contain precisely $\lceil r R' \rceil$, where r is distinct from any routine name appearing in $\lceil R' \rceil$.

A function or routine naming context may be interpreted in the following way. Each function or routine name appearing before an R′-context therein is the name of a function or a routine for which the programmer vouchsafes to furnish a subsequent definition. In that definition, each variable name appearing in the R′-context will be treated as a dummy, representing an argument variable whose name or value will be supplied by the R-expression of each calling context which calls upon the function or routine being defined. Further, each function or routine name appearing in the R′-context preceded by '$\exists^n$' will be treated as a dummy representing the name of an n-argument function or routine, which name will be supplied by the R-expression of each calling context which calls upon the function or routine being defined. Whether a dummy subprogram name in an R′-context is that of a function or routine is syntactically indeterminate within the R′-context, and is known only from the contexts in which the name appears in the definition of the function or routine whose name precedes the R′-context.

Examples of function and routine naming contexts are given in Fig. 18. In the first example, 'gd' is the name of a function which will be defined in terms of a dummy variable, x. In the second example, 'verge' and 'sum' are the names of functions which will be defined later; in the subsequent definition of verge, a, n, and m will be dummy variables, while when sum is defined, x, y, and n will be dummy variables, and g will be a dummy function or routine. In the third example, 'sum' is the name of a routine which will be defined in terms of dummy

Function: gd (x).
Functions: verge (a, n, m), sum (x, y, n, $\exists^1$g).
Routine: sum (x, y, z, m, n, p, $\exists^1$g).

FIG. 18. Examples of function and routine naming contexts.

variables x, y, z, m, n, p, and the dummy function g. The second and third examples could not appear in the same program, of course, since they specify a common name (sum) in two different ways.

PROGRAMMER DEFINITION OF FUNCTIONS

Let 'f' have appeared with an R′-context in a prior function naming context. Then the syntactic sequence of contexts which defines f will commence with the appearance of 'f' in the name field, and be terminated with the closest syntactically subsequent sentence having a phrase containing the words 'function' and 'end'. Any ordinary context of the COLASL language may be included in this sequence. In addition 'f' may appear as the leftmost defined variable of an algebraic equation, to indicate what the value of the function is to be. This is the *only* case wherein a label other than a variable name may appear as a defined variable. However, no argument variable may appear as the defined variable of an algebraic equation.

PROGRAMMER DEFINITION OF ROUTINES

Let 'r' have appeared with an R′-context in a prior routine naming context. Then the syntactic sequence of contexts which defines r will commence with the appearance of 'r' in the name field, and be terminated with the closest syntactically subsequent sentence have a phrase containing the words 'routine' and 'end'. Any ordinary context of the COLASL language may be included in this sequence.

EXIT

The *exit* context consisting of the word 'exit' appearing in a phrase is defined only in functions and routines. It is interpreted as meaning that the local logical sequence of the function or routine in which it appears is terminated with the sentence containing it.

Examples of the definition of the functions and routine names in the examples of Fig. 18 are given in Fig. 19. In these examples, all variables which are not arguments are common to the main program and all its sub-programs, except the quantified variables used in writing loops. These examples also

gd gd = arctan (sinh (x)). End function.

verge If $\left(\left|\frac{a_i - a_{i+1}}{a_i}\right| > \epsilon\right)$ then verge = 0, exit. From verge ($i = n, n+1, \ldots, m$), verge = 1, exit, end function.

sum $\Delta d = \frac{y - x}{n}$, $d = x$, $\sigma = 0$.

#1 $\sigma = \sigma + g(d)$, $d = d + \Delta d$.
If $(d < y)$ then go to #1. sum = σ, end function.

sum $z_i = 0$, $(i = m, m+1, \ldots, n)$.

#1 $\Delta d = \frac{y_i - x_i}{p}$, $d = x_i$.

#2 $z_i = z_i + g(d)$, $d = d + \Delta d$.
If $(d < y_i)$ then go to #1. From #1. $(i = m, m+1, \ldots, n)$.
$p = p + 1$. End routine.

FIG. 19. Examples of function and routine definition, corresponding to examples in Fig. 18.

illustrate the manner in which the use of an argument in the definition of a function or routine constrains the form of the C-expressions in a calling context.

CONSTRAINTS ON ARGUMENTS OF FUNCTIONS AND ROUTINES

An argument in a calling context will be said to correspond to an argument in a function or routine definition if and only if the argument in the calling context is preceded in its R-expression by as many commas as precede the argument in the R′-context used in naming the function or routine. Let a and A represent corresponding arguments; a the argument appearing in the definition of a sub-program, and A the argument appearing in the calling context thereof. If 'a' is a function or routine name, then 'A' must be just a function or routine name respectively. If 'a' is a variable name, and never occurs within the definition of a sub-program with a subscript S-expression, or as the defined variable of an algebraic equation, then ⌈A⌉ may be any expression. If 'a' is a variable name occurring in the sub-program definition with at most a p-dimensional subscript, then 'A' must be just a variable name, which is the name of an array of at least p dimensions. If 'a' is a variable name occurring in the sub-program as the defined variable of an algebraic equation, but

never occurring with a subscript, then $\lceil \overline{A} \rceil$ may be any variable name, with or without a subscript. In the case of standard functions, these constraints on arguments may be found in the function write-ups.

In the examples in Figs. 18 and 19, then, gd could have any expression as its argument. The first argument of verge could be only the name of a one or more dimensional array, but its second and third arguments could be any expressions. The first three arguments of the sum function could be any expressions; however, the fourth argument could be only the name of a one-argument function. Finally, the first three arguments of the sum routine could be only names of one or more dimensional arrays; the next two arguments could be any expressions; the sixth argument could be any possibly subscripted variable name; and the seventh argument could be only the name of a one-argument function.

SYMBOLIC CODING

There are two pseudo-contexts and one context left to discuss. These structures are necessary because it is not yet practical to write a translator which will produced a completely optimum code, and it seems impossible to design an algebraic language which can fully use all features of every machine for which it might be implemented. Hence the symbolic coding language, into a subset of which COLASL contexts are translated, is included in the COLASL language. So that the programmer may achieve some degree of compatibility between his symbolic code and that produced by the COLASL translator, a context has been provided permitting him some degree of control over the assignment of the index registers of the computer.

A clause containing no recognizable COLASL context is assumed to be a statement in the symbolic assembly language. Its effect on the logical sequence of a COLASL program is assumed to be that of an algebraic equation, even though it may in fact be otherwise. Thus, it is the programmer's responsibility to be sure that his symbolic statements are compatible with the surrounding COLASL-generated code.

All material following the appearance of '←' in the class column and prior to the first subsequent appearance of '→' in the class column is assumed to belong to the symbolic assembly language, and is sent on to the assembler unaltered. This

material is said to be written in *process exception mode.*

At present, since no input-output contexts have been specified in the COLASL language, all input-output must be handled through a set of symbolic assembly macro-operations called KIT (7). Some of these statements have been included in the sample codes given at the end of the paper.

Let n represent one of '1', '2', '3', ... Then any one of 'n', 'v', and 'n to v' is an A-context. If a clause has a phrase which contains the word 'attach' and ends in an A-context, and all other phrases in the clause are A-contexts, then the clause is an *index attachment context.* If an index attachment context has an A-context of the form 'n', then index register n is withdrawn from the set of index registers which the code generated by the COLASL translator may use. If an attachment context has an A-context of the form 'v' then the variable v is assigned to an index register selected by the translator. If an index attachment context has an A-context of the form 'n to v' then v is assigned to index register n.

COMMENTARY

Any material which appears in a program written in the COLASL language, which does not conform to the definition of one of the contexts or pseudo-contexts given previously, is treated as noise and discarded. This feature permits the programmer to describe rather fully just what he expects each part of his program to accomplish, in the same sentences in which he writes the program. It is necessary that he should display some discretion in taking advantage of this facility, however, for it is easy to change the meaning of a context drastically by inserting an extra key word or punctuation mark in a critical place. To avoid this difficulty while extending still further the range of possible commentary, the programmer may write his program using two pencils, one red and one black. Everything appearing in red is typed in this color on the IBM-9210, and is ignored completely in interpreting the COLASL language.

There is a complete COLASL program given in Fig. 20. The material originally written in red has been underlined with a wavy line to simplify publication. Fig. 23 has an example of a routine which might be used in some larger problem. Figs. 21

We are given the coefficients of a set of quadratic equations of the form:

$$ax^2 + bx + c = 0.$$

We wish to find those equations which have real roots, and print their coefficients and roots; and to print the coefficients of those equations having complex roots. Let there be a marker, m, for each equation such that $m_i = 0$ if the roots are real and $m_i = 1$ if the roots are complex. We read (according to format F1) the number of sets of coefficients, I. If ($I \leq 0$ or $I > 100$) then the data are incorrect, and we stop. Otherwise we read (per format F2) the coefficient triples, a(i), b(i), and c(i); we also set the initial value of each $m_i = 0$, for ($i = 1, 2, \ldots, I$).

We now compute the discriminant of the current equation, which we will call d, from

#1 $d = b_i^2 - 4\, a_i c_i$.

If ($d < 0$) then $m_i = 1$, and we continue from #2. Otherwise, we compute the roots, x_{pos} and x_{neg}, from the relations:

$$x_{pos_i} = \frac{-b_i + \sqrt{d}}{2a_i}.$$

$$x_{neg_i} = \frac{-b_i - \sqrt{d}}{2a_i}.$$

#2 To complete the calculation repeat the above from #1 for ($i = I, I - 1, \ldots, 1$).

The subscripted variables have the ranges: 100 to 1 for a, b, c, m, x_{pos}, and x_{neg}. When the calculation is complete, print (format F3) which provides headings; then print the coefficients and roots as indicated in the following.

#3 If ($m_i = 0$) then print (using F4) each of the variables, a(i), b(i), c(i), x_{pos}(i), and x_{neg}(i); then proceed from #4. Otherwise, print (per F3), the variables a(i), b(i), and c(i).

#4 Do everything from #3 for ($i = 1, 2, \ldots, I$); then stop.

F1 is the format for reading I. It is (E0 * 3).

F2 is the format for reading the coefficients and printing them where the roots are complex. It has the form (3E2.1.13.3).

F3 is the heading format, (S0, X27, H*COEFFICIENTS*, X50, H*ROOTS*, S2).

F4 is the last format, which controls printing of coefficients which have real roots. It is (3E2.1.13.3, X3, 2E2.1.13.3). To make it accessible to the input-output package, we attach i to an index register; this is the end of the problem.

FIG. 20. Coding example in the COLASL language.

FORTRAN VERSION OF A COLASL PROGRAM

```
      DIMENSION A(100),B(100),C(100),M(100),XPOS(100),XNEG(100)
      READ 10,I
      IF(I)1,1,2
    1 STOP
    2 IF(I-100)25,25,1
   25 DO 3 J=1,I
      READ 11,A(J),B(J),C(J)
    3 M(J)=0
      DO 6 J-1,I
      K=I+1-J
      D=B(K)**2-4.*A(K)*C(K)
      IF(D)4,5,5
    4 M(K)=1
      GO TO 6
    5 XPOS(K)=(-B(K)+SQRTF(D))/(2.*A(K))
      XNEG(K)=(-B(K)-SQRTF(D))/(2.*A(K))
    6 CONTINUE
      PRINT 12
      DO 9 J=1,I
      IF(M(J))8,7,8
    7 PRINT 13,A(J),B(J),C(J),XPOS(J),XNEG(J)
      GO TO 9
    8 PRINT 11,A(J),B(J),C(J)
    9 CONTINUE
      STOP
   10 FORMAT(13)
   11 FORMAT(1P3E14.7)
   12 FORMAT(1H1,27X12HCOEFFICIENTS,50X5HRCOTS,//)
   13 FORMAT(1P3E14.7,3X1PE14.7)
      END(0,1,0,0,1,0,0,0,1,0,0,0,0,0,0)
```

FIG. 21. FORTRAN version of the problem in Fig. 20.

and 22 are FORTRAN and Osage ALGOL* versions of the problem in Fig. 20.

THE COLASL TRANSLATOR FOR THE IBM-7030

Having completed the description of the current state of the input hardware and the COLASL language, a very rough sketch of some of the features of its translator may be of interest. Its fundamental memory organization is that of the NSS list (8), ex-

*Osage ALGOL is an implementation of ALGOL for the computer at the University of Oklahoma. The codes in Figs. 20, 21, and 22 are not optimum formulations of the problem: they are intended to illustrate the language. The ALGOL version was kindly provided by Mr. Richard B. Worrell of the University of Oklahoma.

```
Program 001;
Begin Real d;
      Integer i, I;
      Array a, b, c, XPOS, XNEG [1 : 100];
      Format Alpha ('REAL ROOTS:', S3, 5 (DE10.0, S2), J2),
             Beta ('COMPLEX ROOTS:', S3, 3 (DE10.0, S2), J2);
  Comment This program is written in OSAGE ALGOL. The reading and
    printing is accomplished by the standard procedures READMT and
    PRINT. READMT reads in values of type integer for the simple
    variable I and type real for the subscripted variables a, b and c.
    PRINT will print 10 digit decimal numbers with a decimal exponent
    according to one of the formats (Alpha or Beta). In any event, the an-
    notation REAL ROOTS: or COMPLEX ROOTS: will be printed at the
    beginning of each line.

  START PROGRAM;
    READMT (I, a [1], . . . , a [I], b[1], . . . , b[I], c[1], . . . , c[I]);
    For i ← I Step −1 Until 1 Do m[i] ← 0;
    For i ← I Step −1 Until 1 Do
      Begin d ← b [i] * b [i] −4 * a [i] * c [i];
        If d < 0 Then m [i] ← 1 Else
          Begin XPOS [i] ← (−b [i] + SQRT (d))/(2 * a [i]);
                XNEG [i] ← (−b [i] − SQRT (d))/(2 * a [i])
          End
      End
    For i ← 1 Step 1 Until I Do If m [i] = 0 Then
      PRINT (Alpha, a [i], b [i], c [i], XPOS [i], XNEG [i]) Else
      PRINT (Beta, a [i], b [i], c [i])
End of program 001
```

FIG. 22. Osage ALGOL version of Fig. 20.

panded somewhat to take advantage of the index word format of the STRETCH computer.* The STRETCH index word has a 24-bit value field with a sign, an 18-bit count field, and an 18-bit refill field, as well as three flag bits. Since 18 bits can address any word in memory, the 6 extra bits and sign of the value field may be used as extra flags. The basic element in a COLASL list may then have 3 links or pieces of data, with 10 bits left to be used as needed to describe the contents of the 3 field.

Because the order structure of the STRETCH computer seems reasonably well adapted to list processing, and because branching is very slow relative to other orders, most list manipulating routines have been coded as open subroutines. Because of this fact, and the relatively well-defined nature of an

*A complete description of the characteristics of this machine may be found in (9). Only a minimum description of features important to COLASL will be given here.

THE SPHERICAL BESSEL ROUTINE

The routine, $j_p = \sqrt{\pi/2x}\ J_{p+1/2}(x)$ is represented by the notation: j (x, j, P), where x is a normalized floating point number, j is the first word of the output array, $j_0 \ldots j_p$, and the range of p is determined by the integer P, $P \geq 0$.

j Definition:

Let $\bar{P} = P + 10$, and $M = 2x + 10$. If $(\bar{P} > M)$ then set $M = \bar{P}$, the larger of the two.

Since $j_m \to 0$ as $M \to \infty$, the method consists of guessing $\tilde{j}_M = 0$, $\tilde{j}_{M-1} = 10^{-7}$. The remaining j_i are then computed using the recursion formula

$$\tilde{j}_{i-1} = \frac{2i+1}{x}\,\tilde{j}_i - \tilde{j}_{i+1}, \text{ for } (i = M-1,\ M-2, \ldots, 1).$$

However, $\bar{j}_0$ may be obtained directly from the relationship $\bar{j}_0 = \frac{\text{Sin}(x)}{x}$, and the normalization factor $R = \frac{\bar{j}_0}{\tilde{j}_0}$ can be computed.

The remaining $\bar{j}_i$ are obtained by letting $\bar{j}_i = R\tilde{j}_i$, for $(i = 1, 2, \ldots, P)$. This is the end of the routine definition.

Routine; j (x, j, P).

j $\bar{P} = P + 10$, $M = 2x + 10$. If $(\bar{P} > M)$ then $M = \bar{P}$.

$\tilde{j}_M = 0$, $\tilde{j}_{M-1} = 10^{-7}$.

$\tilde{j}_{i-1} \frac{2i+1}{x}\,\tilde{j}_i - \tilde{j}_{i+1}$, $(i = M-1,\ M-2, \ldots, 1)$.

$\bar{j}_0 = \sin(x)/x$, $R = \bar{j}_0/\tilde{j}_0$. $\bar{j}_i = R\tilde{j}_i$, $(i = 1, 2, \ldots P)$. End routine.

FIG. 23. The spherical Bessel routine in the COLASL language.

automatic coding language translator, list formats are quite irregular.

Recursive coding techniques have been used extensively in the major analysis packages. For example, the expression analyzer calls upon itself as necessary to analyze parenthetical levels and exponents.

The general approach taken in designing the translator has led to the presence of almost no restrictions on the complexity of form of a COLASL program. The dimensioned variables are restricted to 12 dimensions, only 12 index registers may be attached, the generated code may require no more than 500 temporary storage locations for the main program and for each sub-program, no function or routine may have more than 100 arguments, and no letter may have more than 13 components. No other important restrictions exist.

As the cards are read, the character and control codes occurring between each pair of carriage returns are used to construct a three-dimensional representation of the line appearing on the programmers' listing. The first dimension is the number of columns across a page. The second dimension is the vertical position, in half lines, of a character within a column, relative to the last preceding carriage return. The third is the number of characters appearing on top of each other.

When a carriage return is encountered, standard multiple characters, such as '≡' and '≠', are replaced with their own unique codes. Non-standard multiple characters, such as 'à', are each replaced with an 18-bit dictionary key after being entered in a dictionary of multiple characters if not already present therein. Each radical sign is scanned to determine its length and height, and all elements thereof are erased except the '√'; this is replaced by a special element describing the configuration of the complete radical sign. Each underbar is scanned to determine its length, and all occurrences of '_' except the leftmost, which is replaced by a description of the underbar, are erased. All of the preceding activities take place during one left-to-right scan of the three-dimensional representation of the lines, and result in a simplified two-dimensional representation.

The content of the class column, if any, is extracted from this matrix for later use. The next eight columns are analyzed for the presence of a name. If there is a name, and if it could be an element of a transfer vector, before the normal treatment of the name is begun it is entered in a list of transfer vectors. If there is a name, and if it is that of a programmer-defined function or routine, appropriate new elements are added to the list of sub-programs and the list of logic lists. Ordinarily, a name is simply added to the list of sentence names, and an 18-bit key representing it is added to the current sub-program list and the current logic list, after which a new sentence list is added to the current sub-program list.

After any name has been disposed of, and if the translator is not in process exception mode, the remainder of the matrix is given to the basic syntactic analyzers. These recursive routines include a main line locator, a parenthetical level analyzer, a script analyzer, a radical analyzer, and a displayed quotient analyzer. These routines scan the matrix from left to right looking for sentence delimiting punctuation and, as they scan,

add the material from the matrix to the current sentence list. In this list, characters following each other directly are connected together. Scripts are attached to the scripted characters. Parenthetical levels appear as sublists from the main list in which they occur. Radicands are converted to parenthetical levels raised to the reciprocal powers of the radical indices, which are assumed to be '2' unless stated otherwise. Numerators and denominators of displayed quotients are converted to parenthetical levels separated by '/'.

When a period, exclamation point, or question mark is located, the current sentence list is given to the context analyzers. When they are finished, that part of the matrix which has been scanned is printed and erased, a new sentence list is added to the current sub-program list, and the basic syntactic analyzers resume their scan. When the end of a line is reached, the remaining part of the matrix is printed, and a new three-dimensional line representation is formed from the next input characters. This process continues until the context analyzers locate the end of the program.

The first context analyzer decomposes the primitive sentence list into a list of antecedents, clauses, and phrases. It looks for sequences of list elements forming key words, and replaces them with single, key-word-type, elements. It looks for '=' and '≡', and identifies equations therefrom. It determines which parenthetical levels are quantifiers. Finally, it constructs an identification mask fro each clause and phrase, which tells the second context analyzer what to look for.

The second context analyzer has three main modes of operation. If the sentence is a range sentence, then the names of variables not already present therein are inserted into a dictionary of variable names, and the 18-bit dictionary keys for all variables mentioned, together with their script limits, are entered into a list of dimensional variables for later reference. If the sentence is a function or routine naming context, the names of the functions or routines are entered into the appropriate dictionaries, with the names of the arguments to be used in their subsequent definitions. In both of these cases, the sentences are then erased and the context analyzers are through.

In all other cases, the second context analyzer proceeds to examine the antecedents, clauses, and phrases of the sentence. Where multiple contexts exist in a single phrase, extra phrases are added so that each has just one context. Where a phrase has no COLASL context, but another phrase in the clause has at least

one, the null phrase is erased. Variable names, constants, and sentence names are identified, entered in dictionaries where appropriate, and replaced by their dictionary keys. Expressions are expanded preparatory to coding. Equations are consolidated into lists of defined variables and defining expressions lists. Parametric equations are removed to a parametric equation list. A branch, exit, or stop context is removed from the place where it occurs and added to the end of the sentence. Antecedents are reduced to condition lists (representing 'or'ed conditions), which are made up of expression lists and inequality signs. Entries are made in the logic list for all contexts which alter the logical sequence. If a function or routine end context occurs, then 'exit' is generated, the current sub-program and logic list are terminated, and the former position in the main program list resumed. If an end context is identified, the second pass of the translator is initiated.

During the second pass, the parametric variables are evaluated. This is done through a recursive interpretive code, which begins with the defining expression of the last parameter defined. Whenever it encounters a parametric variable which has not yet been defined, it remembers what it was doing and goes to evaluate the new variable.

When all parameters are known, the second pass generates the reservations for arrays of data, and requests for needed index multiple.* Finally, the logic lists are traced and index registers assigned to loops.

All the information is then available to permit coding of the problem. This is done during the third pass in an unremarkable manner, and the resulting symbolic code is given to the STRAP-II assembly program.

At the present writing, the COLASL translator has been written and is about to go on field test. Test programs of several hundred orders, using most of the contexts of the language, have been compiled correctly.

Experience with keypunching has been good, apparently due to the extreme freedom given the keypunchers in doing their work. Aside from the rule about the use of carriage returns mentioned early in the paper, the only instruction given them has been to make their listing look as nearly like the manuscript as possible. The test programs have been written in longhand on ordinary notebook filler paper with good results.

*For a complete discussion of these entities, and of their use, see (10).

Accurate timing estimates for the translation process are not yet available. The reason for this is that the compiling times for all programs run so far are less than the time needed for the translator to make some monitoring comments on the console typewriter and dump the generated code for printing, which will not be done in the production system.

FUTURE PLANS

Improvements in the system are planned in two areas. With respect to the translator, an application of the dictionary technique to subscripting and the logic list can result in great improvement in object code efficiency. With respect to the language, the obvious lack of input-output contexts and Σ- and Π-notation can be remedied. An additional set of contexts is being designed which will permit natural and flexible handling of the interruption mechanism of the machine by the programmer in the COLASL language. Finally, some thought is being devoted to the addition of complex and vector arithmetic and an ALGAE-type* notation into the language.

ACKNOWLEDGMENTS

The authors wish to thank Bengt Carlson and Edward A. Voorhees for material assistance in the performance of this work, without which it could never have been done. Roger Lazarus, Mark Wells, William W. Wood, and especially Edward Voorhees have contributed to the formulation of the language through their comments, criticisms, and suggestions. Of the many people who have aided in the implementation of the translator, the authors wish to mention Frank Evans, Charles Folkner, Earl Kinney, and Grover Lewis. The ability of Mrs. Sue Vandervoort and the LASL keypunching staff to correctly interpret nearly illegible writing has been of invaluable assistance.

REFERENCES

1. Wells, Mark B.: MADCAP: A Scientific Compiler for a Displayed Formula Textbook Language. Comm. ACM, 4, 1 (Jan. 1961), 31.
2. Share Ad-Hoc Committee on Universal Languages: The Problem of

*A description of the ALGAE language may be found in (11).

Programming Communication with Changing Machines; a Proposed Solution. Comm. ACM, 1, 8 (Aug. 1958), 12; 1, 9 (Sept. 1958), 9.
3. Voorhees, Edward A.: A Proposal for a Generalized Card Key Punch. Privately distributed paper (July 1956).
4. International Business Machines, Inc.: Scientific Descriptive Printer Reference Manual. IBM form #R60-0034.
5. International Business Machines, Inc.: Reference Manual STRAP-II 7030 Assembly Program. IBM form #C28-6129.
6. Quine, Willard Van Orman: Mathematical Logic, Harvard University Press, Cambridge, 1951.
7. Kinney, Earl H.: KIT: Decimal Input/Output for STRETCH, Programmer's write-up, Los Alamos Scientific Laboratory, Los Alamos, New Mexico; Dec. 15, 1961.
8. Newell, A., et al.: A Command Structure for Complex Information Processing, The RAND Corporation, 1700 Main St., Santa Monica, Calif., Report #P-1277, Aug. 20, 1958.
9. International Business Machines, Inc.: Reference Manual 7030 Data Processing System. IBM form #A22-6530-2.
10. Carlson, Bengt G., Kazek, Chester S., and Lee, Clarence E.: Floco-II Manual. Los Alamos Scientific Laboratory, Report LAMS-2339, Oct. 13, 1959.
11. Voorhees, Edward A.,: Algebraic Formulation of Flow Diagrams, Comm. ACM, 1, 6 (June, 1958), 4.

COMPILER-INTERPRETER FOR USING IN NUMERICAL ORIENTED LANGUAGES TRANSLATION

A. Mazurkiewicz

Institute of Mathematical Machines,
Polish Academy of Science,
Warsaw, Poland

The main disadvantage of current programming systems is the frequent need for human intervention during the many stages involved in the construction of a programme. During the development of the more complex numerical problems the following sequence occurs all too often:

man → computer → man → computer → man → computer

Let us illustrate this by the following two examples:

Example 1

Let us assume that there are two algorithmic methods for the solution of a given numerical problem, i.e., M_1 and M_2. The satisfaction of condition R will be the criteria for the selection of methods to be applied.

Furthermore, let us assume that method M_2 has two variants, M_{21} and M_{22} and that the choice here is based on condition S.

Under these conditions, the man-computer relationship will be as follows:

1. The analyst (programmer) formulates the M_1, M_{21} and M_{22} methods contingent upon the conditions R and S.

2. The computer examines whether the R condition is satisfied.

If true:

2.1. The programmer constructs the programme according to the M_1 method.

Otherwise:

2.2. The computer examines whether the S condition is satisfied.

If true:

2.2.1. The programmer constructs the programme according to the M_{21} method.

Otherwise:

2.2.2. The programmer constructs the programme according to the M_{22} method.

3. The machine compiles the constructed programme.

As a practical example of the above, consider the development of a programme which approximates the function f(x). Here M_1 may be the Taylor series expansion method; M_{21}—the Tchebysheff approximation method of degree n; M_{22}—Tchebysheff approximation of degree m; condition R may concern the number of terms of the Taylor expansion necessary for the required accuracy and condition S may concern the accuracy of the M_{21} and M_{22} methods.

Example 2

The situation appears to be more complicated, if we assume that the algorithmic method M depends upon a certain parameter α, being assigned a finite number of values, e.g., $1, 2, \ldots, n$.

$$M = M(\alpha)$$

and further that the condition $R(\alpha)$ used in the selection of method $M(\alpha)$ cannot be decided before a processor creates a programme according to the $M(\alpha)$ method.

In this case, the interaction between the programmer's function and the computer's function is represented by the following closed loop:

0.	stet
1.	n
2.	n
3.	n
4.	n
5.	n
exit	n

This inconvenience is due to the fact that in current programming systems, the process of programme translation cannot be affected by information gathered from previous translation processes. This problem does not arise during the

course of a computation, since it is well known, that further computations can always be affected by earlier results.

It may appear that when dealing with a computer, sufficiently large, that it would be possible to write a programme that could decide, during the process of programme operation, the proper choice of a method. This in point of fact is not true. In addition to being inefficient with respect to time and computer storage, circumstances may be such that it would be impossible to select an algorithmic method during operation of the resultant programme. To overcome these difficulties, the following solution is suggested:

Conditional statements concerning processor operation should be introduced as "directives" to the processor. To my knowledge, the only method used to transmit information to a processor is the application of affirmative statements. In order to allow a programmer the means to compile the process control, imperative statements should also be introduced as "directives" to the processor. To enable recognition of these statements by the processor, they should be interpreted by the processor during the translation process, i.e. during the process of obtaining the resultant programme.

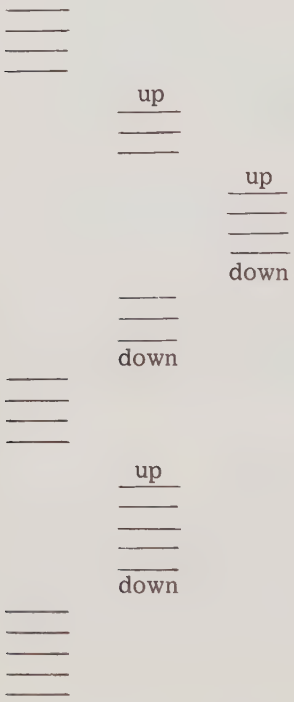
In the author's opinion, the simplest way to attain this is to build a processor that is a compiler as well as an interpreter. All statements of a basic language can then be used as "directives" for the processor, or as statements for the resulting or intermediate programme depending on which level of programme they are used. The possibility of refering to variables of the processor, that characterize a programme being compiled is essential.

The construction of such a translator does not actually differ from the construction of a typical processor. According to the author's opinion the development of a numerically oriented language should be based on a processor of the type being discussed in this paper. Now, the basic concepts of the system will be described.

1. LEVELS OF THE PROGRAMME

The programmes of systems under consideration can be divided into several "levels" using so called "level brackets" *up* and *down*. The *up* bracket raises the level of the instructions following it. The *down* bracket decreases the actual level. There

can be several levels in a programme, and, particularly, there can be no levels, different from the whole programme. There must be as many *up*'s as *down* brackets in the programme, and one level can contain other levels. An example of such a division into levels is given schematically below:



2. THE EXECUTION SCHEME OF DIVIDING A PROGRAMME

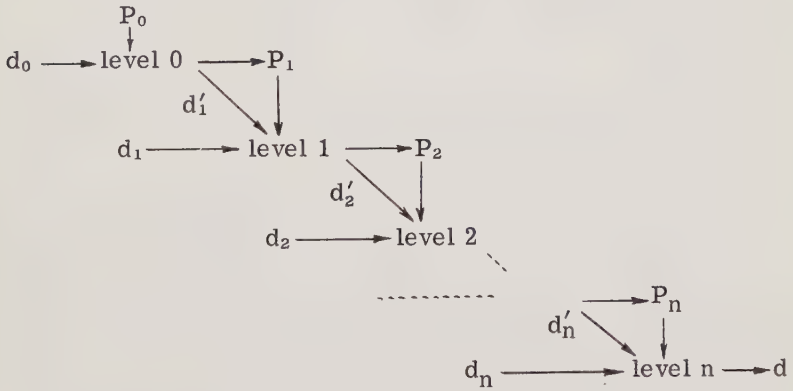
At first, the lowest level is executed. All instructions are performed in the usual sequence until the first *up* bracket is found. Then all instructions written between this *up* and its corresponding *down* are compiled in auxiliary storage. After compilation, the execution of the instructions of lowest level is continued. When the execution of the lowest level terminates, it is possible to pass to executing the just compiled programme. This programme is completed, translated and transfered to the main

storage in a form ready for execution. Before its execution, it can also be written on tape for later use.

Executing such a programme does not differ from execution of the preceding one.

In this way every execution cancels one level. The entire transfer from an object programme to the resulting programme is divided into several stages of compiling and executing. The result of any previous stage forms the input for the next stage.

The variables occuring at any level can be *free* or *bound*. The variable is called bound, if it is defined within a given level. Otherwise, it is called free in this level. The bound variable can be defined by means of an assignment statement or by a executing some input procedure. Generally, the variables of a particlar level may not be all bound; in fact, for almost every level there exist some free variables, which are the only means of communication between lower and higher levels. The values of these variables, assigned by any lower level, form the first set of data for executing the next level. The second set is introduced into machines by means of input procedures. This situation is presented on the diagram below:



The initial data (for executing the lowest level of the programme) are:

1. The multilevel (strictly speaking, three-level) programme
2. The set of external informations, d_0

Having executed the lowest level of this programme one obtains:

1. The programme P_1, in general, multilevel
2. The "linking" informations d_1' as values for free variables of P_1

Together with these informations the external data for P_1 are necessary for executing P_1:

3. The set of external data, d_1.

The program P_1 can be printed and/or immediately executed. If it is executed, one obtains the next programme, P_2 and the linking informations, and so on.

In this manner, every set of data together with its corresponding programme specifies the general purpose programme. During processing time, the general purpose programme becomes a specific one.

4. AN EXAMPLE

Consider the following example, based on ALGOL-60. The function $c = a^b$ is to be evaluated for given b and unspecified a. For the simplicity of the example we suppose that this operation is not available in ALGOL-60. We have the following, general purpose divided programme:

```
begin boolean procedure inputb;
if inputb then
        up
        begin integer i, b; integer procedure input1;
        b: = input1;
                up
                begin real a, c; real procedure input 2;
                integer i;
                c: = 1;
                down
        if b = 0 then go to exit else
                up
                a: = input 2;
                down
        if abs (b) ≥ 8 then
                up
                for i: = 1 step 1 until abs (b) do
                c: = c × a;
                down
        else for i: = 1 step 1 until abs (b) do begin
```

```
      up
      c: = c × a;
      down end
if b > 0 then go to exit else
      up
      if c = 0 then error else
      c: = 1/c;
      down
exit:
      up
 end
 down
      end
      down
else
      up
      begin real b; real procedure input2;
      b: = input 2;
         up
         begin real a, c; real procedure input2;
         if a ≤ 0 then error else
         c: = exp (b × log (a));
         end
         down
      end
      down
end
```

We want to specify this programme for a particular b and for any prescribed real a. At first, we specify the type t of b, by truth values introduced into machine as a result of an input procedure inputb. The value *true* corresponds to the statement: "b is of the integer type", value *false* to the statement "b is of the real type". Further specification is performed by assigning a value to b, as a result of another input procedure. According to this specification there are three levels in the given programme.

Let the input data have the form:

```
true
  3
```

Then, after execution of the instructions of the first level, we obtain the following programme:

```
begin integer i, b; ...
        .
        .
        .
exit:
      up
      end
      down
end
```

Now, by the second specification process, we obtain as a result of second level execution, the value b = 3 and the following piece of programme:

```
begin real a, c; real procedure input2;
integer i;
c: = 1;
a: = input 2;
c: = c × a;
c: = c × a;
c: = c × a;
end
```

For a value of b equal to −10 the corresponding piece of programme will be as written below:

```
begin real a, c; real procedure input 2; integer i;
c: = 1;
a: = input2;
for i: = 1 step 1 until 10 do
c: = c × a;
if c = 0 then error else
c: = 1/c;
end
```

Here the value 10 in the for statement list is the results of abs (−10); this last number is substituted from the lower level as the "linking" parameter. b is the only free variable in the level under consideration. The abs (−10) evaluation is made according to the following principle: if an operation can be performed, perform it.[2]

CONCLUSIONS

The system, briefly presented here, permits description of the compilation process in terms of usual programming languages.

Its main feature is the possibility of making the choice between several computing methods during the translation of the programme. The mutual exchange of executing and compiling process is, in the author's opinion, of great importance for numerical computation. The name "interpreter" has its origins in the fact that several pieces of the programme are executed *before* the compilation is accomplished.

Moreover, in such a system it is easy to define subroutine–generators, which may be very useful in practice. The described system can also be useful in commercial programming languages.

REFERENCES

1. J. W. Backus et al., Report on the algorithmic language Algol - 60.
2. J. K. Iliffe, The use of the GENIE system in numerical calculation, Annual Review in Automatic Programming, vol. 2 Pergamon Press, 1960.
3. R. A. Brooker and D. Morris, An assembly programme for a Phrase Structure Language, Computer Journal, 1960-1961.
4. D. Lukaszewicz and A. Mazurkiewicz, System Automatycznego Kodowania SAKO, Prace ZAM, 1961.

LES ELEMENTS D'UN LANGAGE GENERAL COMMODE VISANT A LA VULGARISATION DES CALCULATRICES DANS L'ENTERPRISE

J. de Guenin

Esso Standard
Recherche Opérationelle
Paris, France

SIMPLE (Simple Intuitive Multi Purpose Language for Everyone) est un langage général, à assemblage rapide, destiné essentiellement à faciliter l'utilisation directe des ordinateurs de l'entreprise* par les non-spécialistes. Il n'est pas conçu pour la rédaction des très grands programmes élaborés, pour lesquels le souci majeur est l'économie de place dans les mémoires, ni pour les programmes particulièrement répétitifs, pour lesquels le souci majeur est l'efficacité du programme résultant. Il existe d'ailleurs, pour ces deux catégories de programmes, des langages particulièrement puissants et pratiques, au premier rang desquels il convient de placer FORTRAN: ce dernier est certes perfectible, mais on ne saurait remettre en cause la plupart de ses principes, ni, par conséquent, lui opposer de concurrent radicalement différent.

SIMPLE, en revanche, est conçu pour la rédaction de programmes moyens, rapidement écrits, fréquemment modifiés, et ce, par des gens qui n'auront jamais a connaître le programme résultant en langage machine. Ses qualités essentielles sont donc les suivantes:

---

*Ces ordinateurs étant le plus souvent à "caractère alphanumérique," c'est pour des machines alphanumériques que SIMPLE est conçu. Il est néanmoins possible, quoique plus compliqué, de l'assembler sur des machines binaires, moyennant quelques modifications.

- il assemble très vite (assez vite pour que l'on juge inutile de conserver le programme résultant dans la plupart des cas)*
- il constitue un guide commode pour la pensée, qu'il permet d'exprimer naturellement
- ses règles d'écriture sont assez souples pour que les erreurs de rédaction formelles soient peu probables
- il permet une communication facile avec la machine (entrée des données, sortie et présentation des résultats).

SIMPLE n'est que le perfectionnement d'un langage existant, SALE II, lequel possède, dans une large mesure, la plupart de ces qualités.

Comme SALE II, SIMPLE utilise, chaque fois que possible, des principes et des instructions empruntés à FORTRAN. Toutefois, il n'est pas compatible avec FORTRAN: on a cru pouvoir faire des langages répondant aux specifications que nous venons d'énumérer à l'aide de sous-ensembles de FORTRAN; c'est oublier qu'il existe des incompatibilités fondamentales entre les spécifications et les qualités que l'on recherche dans les langages les plus puissants.

Ainsi, la multiplication implicite et l'absence de ponctuation, facilités offertes par SALE II, imposent dans l'écriture des variables, des restrictions qui ne gênent nullement pour les programmes moyens, mais qui seraient inacceptables pour un langage aussi puissant que FORTRAN. Réciproquement, GOTRAN, sous-ensemble de FORTRAN, est loin d'offrir les mêmes avantages que SALE II.

Plusieurs années d'éxperience de divers langages symboliques nous ont convaincus qu'à la tendance vers un langage général universel, devrait se substituer la tendance vers deux langages:

- un langage très puissant, mais relativement incommode (c'est la voie FORTRAN - ALGOL),
- un langage très commode, mais limité par sa commodité-même (c'est la voie SALLE II - SIMPLE).

SIMPLE est un langage général permettant de manipuler commodément de l'information sous toutes ses formes, c'est-à-dire:

*Ce programme est assemblé instruction par instruction, l'assemblage d'une instruction commençant dès sa lecture. Les données suivent la dernière instruction, et le programme est exécuté immédiatement après lecture et assemblage de cette instruction, de sorte que l'utilisateur ne voit jamais le programme résultant.

numérique
littérale
symbolique

et par tous les moyens avec lesquels on traite généralement l'information, c'est-à-dire:

- le calcul numérique
- la logique
- les méthodes de présentation matérielle (écriture, disposition, représentation graphique).

LE CARACTERE

Le *caractère* (chiffre, lettre, ou symbole courant d'arithmétique et de ponctuation), est le plus petit élément d'information que SIMPLE permet de manipuler.

Toutefois, les caractères sont le plus souvent traités par groupes de 10 et constituent alors soit des *nombres*, soit des *mots*, le vocable "mot" étant réservé à tout groupe de 10 caractères qui n'est pas traité comme un nombre.

Dans les mémoires de l'ordinateur, un caractère occupe par définition une *position*; un nombre ou un mot occupe un *emplacement*. Un emplacement est donc formé de 10 positions.

LES NOMBRES

Dans un emplacement, un *nombre* est toujours écrit en virgule flottante normalisée, avec deux chiffres pour l'exposant et 8 chiffres significatifs pour la partie décimale.

Ainsi, le nombre $-253.72 = -0.25372 \times 10^{3}$ sera inscrit en mémoire sous la forme:

$$0\overset{+}{3}25372000\overset{-}{0}$$

et le nombre $0.03587 = 0.3587 \times 10^{-1}$ sous la forme:

$$0\overset{-}{1}35870000\overset{+}{0}$$

Conformément aux usages, la combinaison d'un chiffre et d'un signe est représenteé par une lettre: C pour 3, J pour $\overset{-}{1}$, $\overset{-}{0}$ et $\overset{+}{0}$ sont représentés respectivement par $-$ et $+$. Les deux nombres ci-dessus s'écriront donc finalement en mémoire sous la forme:

0C2537200−

et

0J3587000+

N.B. Notons que pour les machines à caractères, il n'est pas nécessaire de faire la distinction entre nombres en virgule fixe (c'est-à-dire nombres entiers) et nombres en virgule flottante: en effet, un nombre comme 387 s'écrit:

$$\overset{+}{0338}\overset{+}{7}00000$$

Il est manifeste que l'addition, la soustraction et la multiplication de nombres entiers donnent des nombres entiers exacts, tant que le résultat reste inférieur à 99 999 999.

Pour les nombres entiers, l'exposant (ici 3) est précisément égal au nombre de chiffres. Un décodage immédiat permet donc de récupérer, sous leur forme entière, tous les nombres utilisés comme indices, donc de trouver simplement l'adresse d'une variable indicée et d'effectuer sur des nombres entiers les calculs d'indices dans les boucles.

Le fait, pour le programmeur, de ne pas avoir à faire de distinction formelle entre nombres en virgule fixe et nombres en virgule flottante permet d'éviter bien des erreurs. Il simplifie, en outre, de façon appréciable le manuel du programmeur.

Dans l'écriture du programme, on utilise les nombres sous trois formes distinctes: les *constantes*, les *variables*, les *fonctions*.

Les Constantes

Ce sont des nombres écrits dans le programme sous leur forme numérique. Ces nombres sont écrits sous leurs formes usuelles.

Ex: 173
173.
−0.012
−.012

N.B. L'emploi de la notation 37 E-3 pour 0.037 est déconseillé, car il complique inutilement la manuel du programmeur par les règles d'écriture qu'il engendre et peut prêter à ambiguités dans certains cas. Il est beaucoup plus naturel d'écrire:

37*10**−3

transcription directe du 37×10^{-3} auquel nous sommes habitués depuis les classes secondaires.

Les Variables

Une *variable* est un symbole qui désigne le contenu d'un emplacement donné, que ce contenu soit un nombre ou un mot.

On peut utiliser comme variable:

- une des 26 lettres de l'alphabet
- n'importe quel groupe de 1 à 10 caractères (blancs compris) encadré par des marques de mémoire

Ex: A
≠TAUX≠
≠X+Y≠
≠X + Y≠

(ces deux dernières variables sont distinctes à cause des blancs. A et ≠A≠ sont également des variables distinctes).

Les Variables Indicées

SIMPLE permet l'utilisation de variables indicées à un seul indice. Une variable indicée s'écrit:

A@I

où: A est l'une des 26 lettres de l'alphabet,
I est un nombre entier strictement positif (constante ou variable)

Ex: C@2
D@J
E@≠INDEX≠

La valeur maximum de l'indice I est indiquée dans une expression DIMENSION, en tête du programme.

Ex: DIMENSION C@10 D@20 E@112

Si, au moment de l'exécution d'une instruction contenant A@I, l'emplacement I ne contient pas un nombre entier strictement positif et inférieur a la dimension spécifiée, un message d'erreur est imprimé.

A et A@I sont deux variables distinctes. L'utilisation de variables indicées ne restreint donc point le nombre de variables ordinaires disponibles.

Les Fonctions

Une fonction est une variable dont la valeur dépend d'un nombre appelé argument. Une fonction se représente toujours par un

groupe de 4 caractères dont le premier est le signe $. L'argument peut être une constante, une variable, ou une expression algébrique, sans limitation de complexité.

Ex: $EXE(U$LNE (V))

représente le nombre

$$e^{U\text{Log } V}$$

Les fonctions les plus utile sont:

$SQR	Racine carrée
$LOG	Logarithme de base 10
$LNE	Logarithme népérien
$EXD	$_{10}$ argument
$EXP	Exponentielle ($_e$ argument)
$SIN	Sinus (argument en radians)
$COS	Cosinus (argument en radians)
$ART	Arc tangente (argument en radians)
$ABS	Valeur absolue
$INT	Partie entière
$RND	Arrondi
$RAN	Nombre aléatoire tiré dans une distribution uniforme d'amplitude (0, argument)
$MIN(A@N)	Indice de celle des variables A@1, ... A@N ayant la val. minimum
$MAX(A@N)	Indice de celle des variables A@1, ... A@N ayant la val. maximum

LES MOTS

Dans l'emplacement désigné par une variable, il peut y avoir aussi bien un mot qu'un nombre.

En fait, à la lecture du programme, un emplacement est affecté à chacune des variables du programme, et le nom de la variable est inscrit dans l'emplacement qui lui est affecté. Ce nom est cadré à gauche et le reste est rempli par des blancs.

Ainsi, lors de la lecture du programme, la variable ≠ALPHA+≠ donnera lieu à l'inscription:

ALPHA+bbbb (b = blanc)

Ce mot subsistera en mémoire tant qu'il n'aura pas été rem-

placé par un autre mot ou un nombre introduit par lecture, transmission ou calcul.

LA REDACTION DU PROGRAMME

Les instructions d'un programme SIMPLE sont exécutées dans l'ordre où elles sont écrites, ordre qui peut être modifié au moyen d'instructions de transfert.

On peut écrire *une ou plusieurs instructions séparées par des virgules* sur une même carte. Mais, lorsqu'il y a plusieurs instructions sur une carte, seule la première est étiquetée. On ne peut donc se transférer qu'à la première.

La première instruction d'une carte commence n'importe où à partir de la colonne 4. Les trois premières colonnes servent à étiqueter les instructions. Cet étiquetage est facultatif. Il n'est nécessaire que pour les instructions auxquelles renvoient des instructions de transfert. L'étiquette se compose de 3 caractères quelconques, blancs exclus.

Carte Suite

Si l'on ne peut faire entrer dans une seule carte la totalité de l'instruction ou des instructions que l'on voudrait y mettre, on peut lui ajouter des cartes "suite": une carte suite comporte le caractère @ perforé en colonne 4.

Dans toute cette brochure, le mot "carte" désigne l'ensemble d'une carte et des cartes "suite" qui la prolongent.

Carte Identification

La première carte d'un programme sert à l'identification du programme et n'est pas considérée comme une instruction. Son contenu est imprimé en ligne avant l'exécution du programme.

Commentaire

On peut introduire des commentaires dans le programme, à condition de les faire précéder de l'instruction COMMENT. Lorsque cette instruction est rencontrée, le programme d'assemblage ignore *le reste de la carte* et passe à la carte suivante.

END

La dernière carte d'un programme contient l'instruction END et cette instruction seulement. L'instruction END a un double rôle:

- Lors de la lecture du programme, elle sépare le programme des données,
- Lors de l'exécution, elle provoque l'arrêt de l'ordinateur.

END SAVE n

Si l'on désire conserver le programme résultant, on peut le sortir sur la bande n en remplaçant l'instruction END par l'instruction END SAVE n, où n est un entier explicité.

Ponctuation et Blancs

Il n'y a aucune ponctuation à l'intérieur d'une instruction SIMPLE. Les blancs ne sont interprétés que dans deux cas:

- A l'intérieur des marques de mémoire délimitant une variable
- Lorsqu'ils séparent deux nombres. (Il ne saurait donc y avoir de blancs entre les chiffres d'un même nombre.)

Ces deux cas mis à part, toute latitude est laissée au programmeur pour tasser ou aérer l'écriture a son gré.

N.B. Ces propriétés permettent d'écrire un programme SIMPLE "au fil de la plume," sans avoir à se référer à un manuel pour verifier le formalisme de l'écriture. Outre l'aisance qu'elles confèrent à la rédaction, ces propriétés permettent d'éviter bien des erreurs.

INTRODUCTION EN MEMOIRE DES NOMBRES ET DES MOTS

Il y a deux façons d'introduire un nombre ou un mot dans un emplacement: par transmission ou par lecture.

Transmission d'Emplacement à Emplacement

L'instruction A = B envoie dans l'emplacement A le contenu de l'emplacement B. B n'est pas modifié.

Si l'on veut envoyer en A une constante définie une fois pour toutes, par exemple 3.1416, on écrira simplement:

```
A = 3.1416
```

De même, si l'on veut envoyer en A un mot donné, par exemple DATE, il suffit d'écrire:

```
A = ≠DATE≠
```

Si rien n'a été transmis dans l'emplacement de la variable ≠DATE≠ par une instruction antérieure, cet emplacement contient précisément le mot DATE, et l'instruction ci-dessus transfère ce mot en A.

Lecture de Cartes

L'instruction

READ Liste

où "Liste" représente une liste de variables, provoque la lecture de la lère carte en attente dans le lecteur, et la transmission des données de cette carte dans les emplacements définis par les variables de la liste.

Ex: READ A B@12 R@J (F@K K=13) ≠TAUX≠

Si la carte qui se trouve dans le lecteur a ce moment-là contient

1.5*10**-3 400 -12.7 -.012 .1 .2 +0.137

1.5×10^{-3}	sera envoyé en A
400	sera envoyé en B@12
-12.7	sera envoyé en R@J
-0.012	sera envoyé en F@1
0.1	sera envoyé en F@2
0.2	sera envoyé en F@3
0.137	sera envoyé en ≠TAUX≠

Le signe d'un nombre se place devant le premier chiffre (ou le point) du nombre, mais peut en être séparé par autant de blancs que l'on désire. Le signe + peut être omis. Deux nombres successifs sont séparés par des blancs en nombre quelconque. Il en résulte que les chiffres d'un même nombre doivent être jointifs et que, dans l'écriture exponentielle, le signe * ne saurait être omis.

Si, au lieu de nombres, on veut introduire des mots dans certaines des variables de la liste, il suffit de faire précéder ces variables de nombres entiers indiquant le nombre de caractères à introduire.

Ex: READ A 4T 25U 30 R(10)

Si la carte à lire est perforée comme suit (b désignant un blanc):

bb5.143TAUXTAUXbINFERIEURbAb−0.1bbbb
RESULTATSbINTERMEDIAIRESbb ...

- 5.143 sera envoyé en A
- TAUX sera envoyé en T et cadré à gauche. Le reste de T sera rempli par des blancs
- TAUXbINFERIEURbAb−0.1bbbb sera envoyé en U (10 premiers caractères), et dans les emplacements suivants, soit V (10 caractères), et W (5 derniers caractères). Les 5 dernières positions de W seront remplies par des blancs.
- RESULTATS INTERMEDIAIRES ira s'inscrire dans les emplacements R(10), R(11), et R(12), les dernières positions de R(12) contenant des blancs.

Une zone non numérique liée à une variable est définie par les deux règles suivantes:

1° - Elle commence immédiatement après la zone qui la précède (si cette dernière est numerique, elle se termine par le dernier chiffre du nombre qu'elle contient)

2° - Elle comporte un nombre de caractères, blancs compris, égal au nombre qui figure devant la variable.

Si les 80 colonnes de la carte sont épuisées avant que la liste suivant le mot READ ne le soit, une erreur est signalée.

Si la liste est épuisée avant que les 80 colonnes de la carte ne soient lues, le reste de la carte est ignoré.

Lorsque la liste appelle un nombre et que le premier caractère non blanc de la zone correspondante n'est ni +, ni −, ni un chiffre, une erreur est signalée.

S'il n'y a pas de cartes dans le fichier au moment de l'exécution de READ Liste, une erreur est signalée et l'ordinateur s'arrête. On peut repartir à l'instruction suivante en appuyant sur le bouton depart. On peut éviter cet arrêt en utilisant l'instruction ELSE immediatement après l'instruction READ. Ainsi,

READ AXU , ELSE 250

signifie: Lire A, X, et U s'il y a une carte à lire, mais s'il n'y en a plus, se transférer à la première instruction de la carte 250.

Lecture de Bandes

Si l'on désire lire les données à partir d'une autre unité

d'entrée dont l'adresse est le nombre entier n (par exemple un lecteur de bandes), on fera précéder l'instruction READ Liste de l'instruction:

DATA ON n

n est une constante entière

Exemple: DATA ON 7

permet de lire les données sur la bande 7.

L'instruction change le sous-programme de lecture chaque fois qu'elle est éxecutée. Le changement effectué reste valable jusqu'à la fin du programme ou jusqu'à l'exécution d'un nouveau DATA ON.

LES OPERATIONS DE CALCUL NUMERIQUE

Les signes opératoires sont:

Addition	+
Soustraction	–
Multiplication	*
Division	/
Exponentiation entière	**

(Pour l'exponentiation quelconque, on utilise les fonctions exponentielle et logarithme.) Le signe * peut être omis, ce qui confère à SIMPLE la même souplesse que l'écriture algébrique courante.

Exemple: X = (–B+$SQR(B**2–4AC))/2A

Si le signe * est omis entre deux constantes, ces deux constantes doivent être séparées par un blanc, faute de quoi elles sont considérées comme formant un même nombre.

Exemple: 34 C signifie 34*C

mais 3 4C signifie 3*4*C

Ordre des Opérations

Quand il n'est pas entièrement défini par des parenthèses, il est le suivant:

1. Exponentiation
2. Multiplication et Division
3. Addition et Soustraction

Exemple: A+B/C + D**E F − G

est calcule

A+(B/C)+((D**E)F) − G

Rang au Sein d'Opérations de Même Ordre

SIMPLE calcule à partir de la droite. Si . représente les opérations d'un des ordres ci-dessus (par ex. * et /), l'instruction

X = A.B.C.D.

sera calculée

X = (A.(B.(C.D)))

Ex: A/B*C est calcule $\frac{A}{BC}$ (FORTRAN l'aurait calculé $\frac{A}{B}$ C).

LES OPERATIONS SUR LES MOTS

Introduction d'un Caractère dans un Mot

L'instruction

PLOT * J A

où
- * est un caractère quelconque
- J une variable ou une constante entière (vérifiée au moment de l'exécution)
- A une variable quelconque

place le caractère * dans la Jème position de l'emplacement A (ou d'un emplacement suivant si J > 10)

Exemple: PLØT . 22 A@1

place un point dans la 22ème position du tableau A.

N.B. Si 12 emplacements consécutifs (par exemple A@1 à A@12) sont utilisés pour simuler une ligne de sortie, l'instruction PLOT permet de sortir très commodément des dessins de l'ordinateur, par exemple, le tracé d'une courbe par points.

Transfert d'un Caractère d'un Mot dans un Autre

L'instruction

MØVE I A J B

où I & J sont des variables ou constantes entières (verifié au moment de l'exécution)
A & B des variables quelconques

place le caractère contenu dans la Ième position de l'emplacement A (ou d'un emplacement suivant si I > 10), dans la Jème position de l'emplacement B. A est inchangé.

Exemple: MØVE 3 ≠ALPHABET≠ 12 B@1

Si au moment de l'exécution ≠ALPHABET≠ contient ABCDEFGHIJ, la lettre C sera inscrite dans la 2ème position de B@2.

N.B. Cette instruction est particulièrement utile pour créer des libellés variables, engendrer automatiquement la partie littérale des données d'un autre programme à entrées standard (comme les programmes linéaires), etc.

ORGANISATION LOGIQUE DU PROGRAMME

Arrêt

L'instruction

PAUSE n

où n est un constante entière

arrête l'ordinateur. n est affiché au pupitre. En appuyant sur le bouton Départ, l'exécution du programme reprend à partir de l'instruction suivante:

Exemple: PAUSE 222

Transfert

L'instruction

GØ TØ N

où N est une étiquette d'instruction

interrompt la séquence normale d'exécution du programme et transfère l'exécution à l'instruction N, à partir de laquelle elle se poursuivra normalement.

N peut être soit une étiquette explicite, soit une variable. Dans ce dernier cas, N doit contenir soit un entier positif ≤ 999, soit 3 caractères non blancs en positions 3, 4 et 5, et des blancs partout ailleurs (vérifié au moment de l'exécution).

Exemples: GØ TØ 024
GØ TØ ETA
GØ TØ J@3

Boucle

On peut exécuter une même série d'instructions pour des valeurs entières successives d'un indice au moyen de l'instruction DO.

DØ n I = J K L

où n est une étiquette de carte
J, K, L des constantes ou variables entières strictement positives (vérifié au moment de l'exécution)

signifie: "Exécuter toutes les instructions suivantes, jusque et y compris la dernière instruction de la carte n, pour toutes les valeurs entières successives de I, variant de L en L, à partir de J et inférieures ou égales à K".

L peut être omis. Il est alors pris égal à 1.

Exemples: DO 127 I = 1 N 2
DO 325 K = 1 12

La valeur de l'indice I d'un DO, ainsi que la limite K et l'incrément L peuvent être altérés par des instructions situées à l'intérieur de la boucle: l'exécution de la boucle se poursuivra alors à partir de la nouvelle valeur calculée de l'indice.

Les transferts à l'interieur ou à l'extérieur d'une boucle sont possibles. Quand on sort d'une boucle, par un transfert, ou parce qu'elle est terminée, I reste défini et conserve la valeur qu'il avait dans la dernière exécution de la boucle.

Il peut y avoir des boucles DO les unes a l'interieur des autres, à condition que toute boucle commençant à l'intérieur d'une autre soit effectivement contenue tout entière dans cette dernière.

L'instruction CONTINUE peut être utilisée comme dernière instruction d'une boucle, afin de fournir un numéro auquel peuvent se référer des instructions de transfert à l'intérieur de la boucle. (Les transferts directs à l'instruction DO provoqueraient la réinitialisation de l'indice.) Cette instruction ne donne lieu à aucune opération.

Transfert Conditionnel

Les instructions de transfert ont la forme générale:

IF proposition n

où "proposition" est l'une des propositions de la liste ci-dessous
n est une étiquette explicite

Exemple: IF A = B 235

Si, pour la valeur qu'ont les variables contenues dans la proposition au moment de l'éxecution, la proposition est vraie, il y a transfert a l'instruction n; sinon, le programme exécute l'instruction suivante.

Si l'on désire se transférer à une autre instruction que l'instruction suivante lorsque la proposition est fausse, on peut utiliser l'instruction ELSE:

IF proposition n , ELSE m

signifiant: si la proposition est vraie, aller en n; si elle est fausse, aller en m.

Enfin, si l'on désire executer l'instruction suivante lorsque la proposition est vraie, on peut utiliser NEX en guise d'étiquette, ce qui permet par exemple d'écrire les instructions suivantes sur la même carte.

```
Exemple:      IF Y ≠ 0  NEX, ELSE 235 , Z = X/Y
              .........
          235 Z = 0
```

Les propositions acceptables par SIMPLE sont les suivantes:

Propositions de Comparaison Algébrique

A et B étant deux *nombres* (variables ou constantes), on peut utiliser l'une quelconque des formes suivantes:

```
A  =  B
A  ≠  B
A  I  B   (pour A < B)
A  S  B   (pour A > B)
A  IE B   (pour A ≤ B)
A  SE B   (pour A ≥ B)
```

Propositions de Comparaison de Mots

Les caractères Hollerith ayant un ordre de référence, il est possible de comparer "alphabétiquement" deux mots (en langage machine, cette comparaison est même encore plus directe que celle de deux nombres en virgule flottante).

A et B étant deux *variables* contenant des *mots*, on peut utiliser toutes les propositions précédentes en les faisant précéder de WORD:

WORD A = B
WORD A ≠ B
etc.

Propositions Composées

Une proposition composée est une proposition formée de deux ou plusieurs propositions élémentaires empruntées aux listes ci-dessus, séparées entre elles par l'une des 2 conjonctions booléennes OR et AND.

p_1, p_2, p_3 étant des propositions elementaires,

la proposition p_1 OR p_2 OR p_3

est vraie si et seulement si l'une au moins des propositions p_1, p_2, p_3 est vraie,

la proposition p_1 AND p_2 AND p_3

est vraie si et seulement si p_1, p_2, p_3 sont vraies toutes trois.

Exemple: IF WORD ≠TITRE≠ = ≠CALCUL≠ AND X ≠ 0 135

N.B. L'emploi simultané de OR et AND dans une même proposition composée est proscrit afin d'éviter les parenthèses qui compliqueraient à la fois la compilation et les règles d'utilisation du langage.

MANIPULATION DE VARIABLES INDICEES

N.B. Les instructions du § précédent (organisation logique du programme) permettent de rédiger facilement les programmes les plus compliqués. Mais, pour facile qu'elle soit, cette rédaction peut devenir longue et fastidieuse lorsqu'elle porte sur des choix d'indices ou des classements de variables indicées, comme on en rencontre fréquemment dans beaucoup d'applications du calcul automatique, du Calcul Matriciel aux Simulations. Il est alors très commode de disposer d'instructions plus synthétiques que celles que nous avons rencontrées jusqu'ici.

Classement

L'instruction RANK UP A@N B@M

où N et M sont des variables ou constantes entières avec $M \leq N$,

permet de classer par ordre croissant les N variables A@I, I=1, 2, ... N, de retenir les M premières, et de mettre en B@J, J=1, 2, ... M, l'indice de la variable A@I qui occupe le Jème rang. Ainsi, si I = B@1, A@I est celle des variables A@1 à A@N qui a la valeur minimum.

L'instruction RANK DØWN A@N B@M

permet, de la même façon, de classer les M plus grandes variables A@I dans l'ordre décroissant.

Recherche d'un Indice

L'instruction FIND I=J K proposition

où I est une variable entière, J et K des constantes ou variables entières

et "proposition" est l'une des proposition du § précédent (organisation logique du programme), contenant au moins une variable indicée avec l'indice I,

permet de sélectionner la première valeur de I comprise entre J et K qui satisfait la proposition. Cette valeur est disponible dans l'emplacement I.

Exemple: FIND I = 1.10 A@I S = B AND A@I ≠ C

Cette instruction permet de trouver la première valeur de I comprise entre 1 et 10 telle que A@I ≥ B et A@I ≠ C.

Si aucune valeur de I ne satisfait la proposition, l'ordinateur s'arrête et l'on peut repartir à l'instruction suivante en appuyant sur le bouton Départ. On peut éviter ce désagrément en utilisant l'instruction ELSE.

Exemple: Find I = 1 N WORD A@I = B , ELSE 327

Transfert Conditionnel

Aux "propositions" du § précédent (organisation logique du programme), on peut enfin ajouter deux propositions fondamentales d'existence et d'exhaustivité:

EXIST I = J K proposition

(il existe I, compris entre J et K, tel que la proposition soit satisfaite)

ALL I = J K proposition

(pour toutes les valeurs de I allant de J a K, la proposition est satisfaite).

Exemples: IF EXIST K = 1 12 A@K IE 100 108

signifie: "s'il existe K, $1 \leq K \leq 12$ tel que $A@K \leq 100$, aller en 108; sinon, exécuter l'instruction suivante".

IF ALL K = 1 12 B@K ≠ C 109

signifie: "si pour toutes les valeurs de K allant de 1 à 12, B@K = C, aller en 108; sinon, exécuter l'instruction suivante".

SORTIE

Impression

L'instruction PRINT Liste

ou Liste représente une liste de variables,

permet l'impression de nombres et de mots, ou la sortie sur une bande simulant l'imprimante. Cette instruction imprime les données contenues dans les emplacements définis par les variables de la liste, suivant le format défini par la carte suivante.

Exemple: 108 PRINT A B R @ 13 C @ J Q

109XX.XXXbXXbb**********bb.XXABCDEFGb
XXXX.

- La carte 109, lère carte suivant une carte PRINT, n'est pas une instruction, mais un dessin de sortie. Les données seront imprimées exactement selon ce dessin, et le numéro 109 n'apparaitra pas.
- Le nombre en A sera converti et imprimé comme indiqué par le ler ensemble de X.
- Le nombre en B sera converti et imprimé comme indiqué par le second ensemble de X.
- Le contenu de R @ 13 sera placé tel quel dans les positions indiquées par les 10 premiers astérisques. Dans les 4 positions indiquées par les 4 astérisques restants, seront placés les 4 premiers caractères de R @ 14.
- Le nombre en C @ J sera converti et imprimé comme indique par les X suivants.

- A,B,C,D,E,F,G seront imprimés comme dans le format.
- Le nombre Q sera imprimé dans le dernier ensemble de X.
- Le signe + n'est pas imprimé. Le signe – est imprimé dans la position qui précède le ler chiffre. Il faut donc le prévoir dans le nombre de X.

Dessin de Page

Lorsque plusieurs instructions PRINT Liste figurent à la suite sur une même carte, leurs formats respectifs doivent apparaître dans le même ordre et l'un au-dessous de l'autre dans les lignes suivantes du programme. En utilisant les cartes "suite", il est possible de grouper un nombre respectable d'instructions PRINT Liste et, par suite, de grouper un nombre égal de lignes de format. Ceci permet de rassembler dans le programme toutes les lignes d'une même page et, partant, d'en avoir une representation visuelle facilitant l'édition.

Erreur d'Ordre de Grandeur

Si la partie entière d'un nombre est plus grande que celle du format:

- elle est quand même imprimée a l'emplacement prévu si elle n'a qu'un chiffre de trop,
- si elle a 2 chiffres de trop ou plus, le nombre est imprimé dans sa forme interne, sans conversion, sur la ligne qui précède la ligne correspondant au format.

SKIP TO n (Espacement)

Si l'on désire sauter une ou plusieurs lignes avant l'exécution d'une instruction PRINT, cette instruction devra être immédiatement précédée de l'instruction SKIP TO n (n est le caractère de contrôle utilisé en espacement programmé. L'instruction place ce caractère au début de la ligne à imprimer).

En l'absence de SKIP TO, il y a saut de page pour la première ligne, et espacement simple pour les suivantes.

Manipulation des Bandes

WRITE TAPE n D F

Cette instruction provoque l'écriture sur la bande n, en un seul enregistrement, du contenu des emplacements D à F.

Exemples: WRITE TAPE 7 A@I A@J

Inscription sur la bande 7 du contenu des emplacements A @ I à A @ J

WRITE TAPE 6 V A@2

Inscription sur la bande 6 du contenu des emplacements VWXYZ A@1 A@2

Si le sticker de fin de bande est détecté, une erreur est signalée et l'ordinateur s'arrête. Si l'on appuie sur le bouton Départ, le programme repart à partir de l'instruction suivante.

READ TAPE n D

Cette instruction provoque la lecture, sur la bande n, d'un seul enregistrement et l'inscription de cet enregistrement en mémoire à partir de l'emplacement D et jusqu'à épuisement de l'enregistrement.

Exemple: READ TAPE 5 B @ 1

Si l'exécution de l'instruction ne peut se faire (détection d'une fin de bande), l'ordinateur s'arrête et émet un message d'erreur. En appuyant sur le bouton Départ, on peut repartir à l'instruction suivante. On peut éviter cet arrêt en utilisant l'instruction ELSE (voir READ Liste).

TAPE MARK n

Provoque l'inscription d'une marque de fin de fichier sur la bande n.

BACKSPACE n

Provoque un espacement arrière sur la bande n.

REWIND n

Provoque le rebobinage de la bande n.

S K I P MARK n

Cette instruction peut être utilisée lorsqu'un fichier est terminé et que l'on veut lire des informations sur le fichier suivant. Elle provoque le franchissement de la marque de fin de fichier, et le positionnement de la bande au début du fichier suivant.

FACILITES COMPLEMENTAIRES

Découpage du Programme

Si la longueur du programme écrit en SIMPLE dépasse la capacité de la mémoire, on peut découper ce programme en plusieurs parties que l'on conservera sur bande magnétique et que l'on appellera en mémoire en temps opportun.

Un segment de programme que l'on désire mettre sur bande doit être immédiatement précédé de l'instruction

n BEGIN SEGMENT AT m

n est l'étiquette de l'instruction. Cette étiquette servira également à désigner le segment de programme considéré. m représente l'étiquette d'une carte précédente ou de la carte n elle-même ($m \leq n$). Grâce à cette instruction, le segment considéré sera chargé à partir de la même position de mémoire que la carte m.

Le segment doit être immédiatement suivi de l'instruction:

END SEGMENT

Les instructions BEGIN SEGMENT et END SEGMENT servent donc à délimiter le segment, à lui donner une étiquette de référence (étiquette de l'instruction BEGIN SEGMENT), et à définir l'emplacement ou il sera chargé. Pour appeler le segment en mémoire, on utilise l'instruction:

GET SEGMENT n

Cette instruction provoque la recherche sur la bande du segment n, et son chargement à partir de la position de mémoire indiquée. L'instruction suivante sera la première instruction du segment.

La bande qui contient les segments est celle qui est utilisée pour l'assemblage.

Mise au Point du Programme

Un certain nombre d'erreurs de rédaction peuvent être décelées par le programme d'assemblage et indiquées à l'utilisateur au moyen de diagnostics. Mais des erreurs de conception peuvent donner des résultats faux, bien que le programme soit formellement cohérent.

La recherche de ce genre d'erreurs est considérablement facilitée par les instructions BEGIN TRACE et END TRACE,

encadrant la section du programme que l'on veut analyser: chaque fois que cette section sera exécutée, tous les nombres calculés par les instructions de calcul seront sortis sous leur forme interne.

```
Exemple: 100   P = Y + Z
         104   BEGIN TRACE
         108   U = VW, X = W/Y
         112   END TRACE
         116   W = X − V
```

Les instructions 100 et 116 sont calculées normalement. L'instruction 108 est exécutée de la façon suivante: U et X sont calculés normalement, puis U et X sont imprimés de la facon suivante:

```
108   0A3442000+  U
108   0J4471568Q  X
```

N.B. BEGIN TRACE et END TRACE ne sont pas des instructions du programme, mais des instructions d'assemblage: elles modifient l'assemblage des instructions de calcul encadrées, de sorte que si l'on se transfère directement à 108, sans passer par 104, les lignes ci-dessus seront quand meme imprimées. On peut supprimer BEGIN TRACE et END TRACE dès que le programme est au point.

DESIGN OF LANGUAGES–FOR COMMERCIAL PROBLEMS

RAPIDWRITE-COBOL WITHOUT TEARS

E. Humby

ICT, London, England

WHAT RAPIDWRITE ACHIEVES

A year's experience in writing programs in COBOL helps to impress the advantages in using such a language for the expression of commercial problems. Some disadvantages on the other hand make themselves evident in a much shorter time. I.C.T. Rapidwrite was developed with the aim of eliminating the disadvantages without losing any of the advantages, particularly those of Readability and Compatibility. Setting the bad points off against the good this is how we viewed COBOL.

1. It is valuable that the people who are not familiar with the computer or its coding can read what the program is about. It is unfortunate that the cost of this is paid by the programmer who has to use data-names which are much longer than is necessary for uniqueness and sentences which contain much redundancy. Ideally the programmer should use abbreviations in a format where the meaning can be deduced by position rather than by context and Readability should be added later by an automatic process.

2. The isolation of the descriptions of the machine and the data from the description of the procedure not only is the key to compatibility and readability but makes for standard documentation. A free format description of data however is not so helpful as a pictorial representation of the data which more closely indicates its layout on input/output media and its occupation of store internally.

3. COBOL can be learnt generally with more ease than particular machine codes. Nevertheless it takes around 100 pages of manual to lay down the rules. It would be a boon to have a language of the power of COBOL but for which the rules could be stated in less than 10 pages and which could be learnt in a few days.

4. It is valuable that programs can be shared by users of different computers. It is unfortunate that this exchange is limited

to users who understand English. Whilst using the same rules and formats it should be possible for people of many tongues to put their programs down and read them back in their natural language with translation to and from other natural languages available as an automatic computer process.

Rapidwrite and its compiler achieves these ends. The program is written on forms on which all the key and noise words are already printed leaving boxes for the data names which may be freely abbreviated by the programmer when writing his procedure. The facilities available are a powerful subset of the COBOL ones. The procedure description is covered by the use of 11 different statement formats. A two-day course is sufficient to explain the rules and to allow exercises at each phase so that the average student is confident to tackle his own program with the aid of an 8 page manual. The Rapidwrite compiler allows for an expansion of the programmers abbreviations to explanatory data-names, for the addition of those words which make the names into readable statements and for the translation if required from one natural language to another.

READABILITY

The requirements that a program should be briefly written down by the programmer and that it should be easily read by manager or systems analyst are often regarded as contradictory. Some language designers elect for symbolism despite the needs of readability and some elect for verbosity despite the cost to the program writer, and yet others attempt an ugly and unsuccessful compromise. The real solution hinges on the fact that management and programmer require their separate readabilities at separate times and it is possible to provide both at the appropriate time. Whilst the manager is pleased to read the COBOL Environment phrase

OBJECT-COMPUTER 1301, MEMORY SIZE 1200 IAS WORDS, 48000 DRUM WORDS

the programmer knows that only the 1200 and the 48000 are significant to the compiler so the Rapidwrite Environment form (see Fig. 1) is designed so that he merely enters these numbers in the appropriate boxes. The COBOL statement

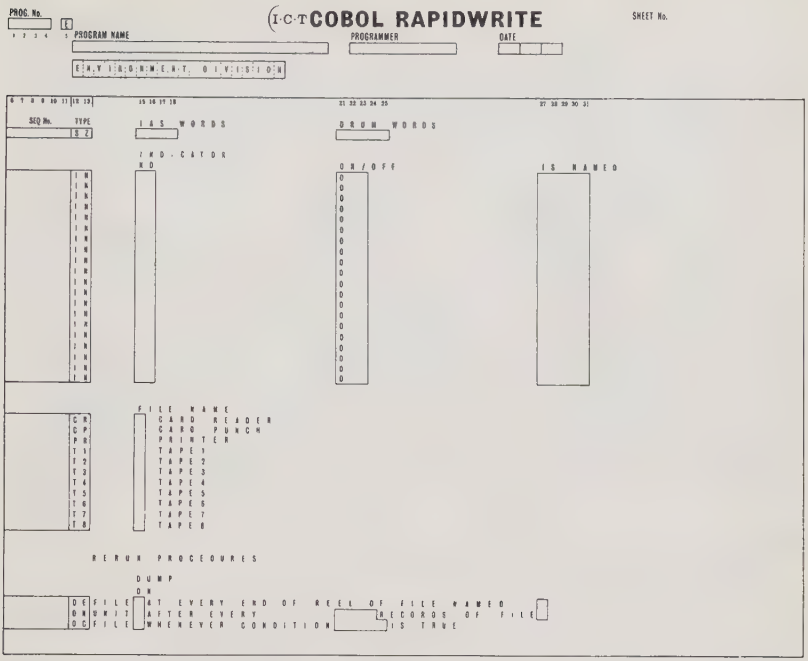


FIG. 1.

SPECIAL NAMES. INDICATOR-NO-1 OFF STATUS IS EMERGENCY.

is indicated by at most 9 characters along one of the lines Type IN.

Writing a "C" in the box next to type CR on the Rapidwrite Environment form is the equivalent of the COBOL expression

FILE CONTROL. SELECT CUSTOMER-BALANCES, ASSIGN TO CARD-READER.

The principles used to keep down the volume of writing for the programmer are:

1. As much of the phraseology as possible is preprinted.
2. The compiler is left to deduce meaning by the position of names on the forms rather than by context in free format.
3. Data-names are confined to 5 characters.
4. File-names are confined to 1 letter.

These principles are continued throughout the Rapidwrite data division form (Fig. 2) and the procedure division cards (see samples Fig. 3). Not only is it valuable to save the programmer writing time but, since the punch operator will punch only the same information (plus a two digit code to indicate to the compiler the form line or card type) then many errors of spelling, punctuation and phraseology are avoided.

The Rapidwrite statements are expanded to fully readable and valid COBOL at translation time. Figure 4 shows the basic action of the compiler program. Since Rapidwrite is a subset of the COBOL facilities and its divisional structure the same, then much of the compiler program is common. The COBOL scanner prints out the COBOL source statements before extracting the significant pieces of information. The Rapidwrite scanner sees its input already in this form but its principal task is to dress up the Rapidwrite input into true COBOL. It makes use of a

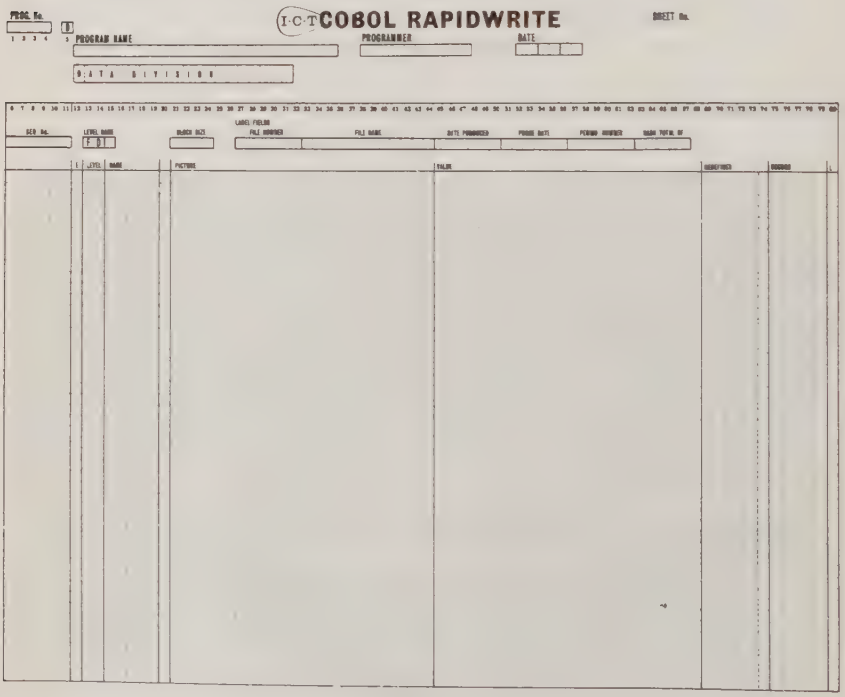


FIG. 2.

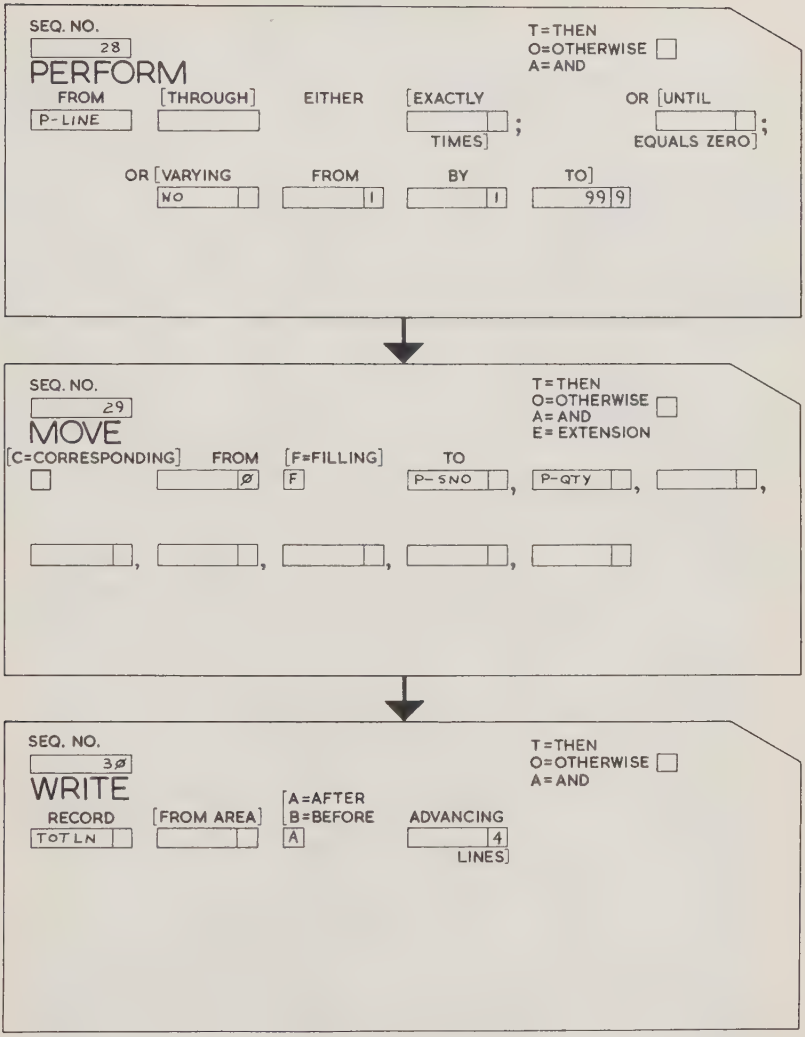


FIG. 3.

format dictionary to supply the balance of noise words for the basic sentences. In order to expand the 5 character data-names and single letter file-names into longer names acceptable to the manager, a synonym table is supplied at translation time. In a previous example therefore if the synonym table had contained

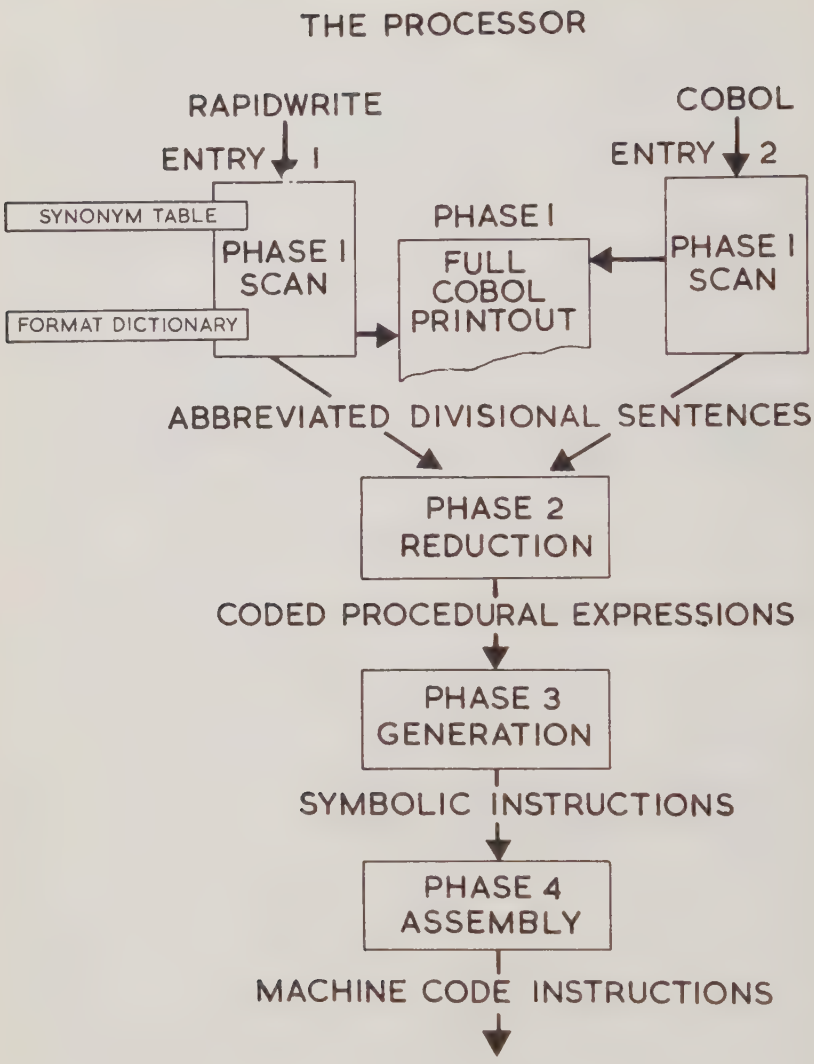


FIG. 4.

one entry C = CUSTOMER-BALANCES then in the printout the word CUSTOMER-BALANCES would appear on each occasion that the programmer had mentioned file C.

SIMPLICITY

In order to reduce the volume of learning, the COBOL facilities were carefully measured as to their usefulness. Some, like EXAMINE, were not frequently required but were responsible for a number of rules, so were not included in the Rapidwrite repertoire.

Some facilities exhibited a certain amount of redundancy. COMPUTE with a formula caters for all the arithmetic and there is no particular use for the other arithmetic verbs. COMPUTE GROSS = RATE * HOURS + BONUS is quite as clear as MULTIPLY RATE AND HOURS GIVING BASIC. ADD BASIC AND BONUS GIVING GROSS.

The use of the ALTER verb creates GO TO expressions in a COBOL program which are always altered before use though there is nothing about them in the written program to show that they are subject to alteration. This not too readable facility can easily be met in other ways. The GO TO DEPENDING ON, for instance, will show clearly what the several switch positions are.

Given certain not too prohibitive restrictions many other rules could be omitted and the sub-set of facilities around which RAPIDWRITE is built are as follows:

Environment.

- Memory capacity
- Indicator names
- File assignment
- Rerun procedures

Data.

- Standard labels for tape files
- Data-names
- Level
- Picture
- Value
- Redefines
- Occurs

Procedure.

- Paragraph Name
- Read
- Write
- Compute

Move
If
Go
Stop and display
Include
Subscript
Perform

As well as reducing the number of rules to be learnt Rapidwrite seeks to guide the user in the employment of the remaining ones. By a careful design of the forms on which he writes his programs the layout of and the description preprinted around the boxes shows him clearly what facilities are available, what options he may exercise, whether file or data-name is required and what size his names and numbers may be. Football pool promoters cottoned on to these ideas directly. I cannot recall a period in which we explained our forecasts on blank paper in free flowing English expressions.

To apply these principles of preprinted fixed format to the Procedure Division requires more flexibility. Rapidwrite meets this by providing a preprinted card for each of its eleven 'verb' formats. The standard key and noise words are preprinted and boxes left for the data-names, paragraph-names, literals, etc., which vary in each phrase. Since data-names are only 5 characters long, file names one letter long and qualification only by file allowed, then six-character boxes suffice for any data-name.

Having completed his Environment and Data sheets the programmer sits before these and his broad flow chart. From a tub file of procedure cards he pulls the appropriate ones in turn, completes with his data-names and can lay the cards out as though building a block diagram with prefabricated blocks. In this form the redirection of jumps and the insertion and removal of pieces of program are considerably simplified. Fig. 3 shows a section of program so prepared.

A conditional sentence will be made up of several cards. The first will be an IF card containing the condition and this will be followed by imperative cards marked for example T,A,A,O,A representing the compound sentence IF ... THEN ... AND ALSO ... OTHERWISE ... AND ALSO ... .

The results of applying these principles has resulted in a Rapidwrite manual which is 8 pages long and it is possible to give a course on its use in 2 days - including practical exercises after the introduction of each group of facilities.

FIXED FORMAT

In data description, in particular, fixed format makes for a much quicker appreciation of data relationships than if free expression is used. The Rapidwrite data form is essentially tabular therefore and as much information as possible packed into a PICTURE. This has necessitated introducing additional codes as was done indeed in some measure in COBOL 61. For example Zero suppression is applied by bracketing that part of the value concerned. Signs and decimal point positions are indicated by symbols. Thus the COBOL descrption "SIZE IS 8 CHARACTERS, SIGNED, POINT LOCATION IS LEFT 3 PLACES, CLASS IS NUMERIC, ZERO SUPPRESS LEAVING 2 PLACES" is given in Rapidwrite as a picture T(999)99V999.

It can be seen from the following extract from a Rapidwrite data division that the picture is a much clearer indication of the layout of the data on output medium and in internal storage:

```
14   FD   T        0024
15   01   TOTLN
16   02   P-SNO    BB999BB
17   02   P-QTY    99999BB
18   02   P-GRS    £(9999)B(19)Be
19
20   FD   W
21   77   TONET    9999(19)e
22   77   NO       999
23   77   TOGRS    9999(19)e
```

Rapidwrite uses in the picture the digits 0-9 and also e for eleven in order to indicate mixed radix values for which the compiler provides conversion program. A non-decimal currency compels this in the first instance but it is a useful facility for times and dates. Ten and eleven appear as single position characters on printers in the U.K. because of the 12 pence in one shilling.

Working storage description is introduced as though it were a special file "W."

NATURAL LANGUAGE TRANSLATION

COBOL is not strictly an international language so long as its readability depends on a knowledge, however,limited, of English.

Rapidwrite, on the other hand, enables the benefits of COBOL to be enjoyed by the user of any natural language which can be contained in a 26 character alphabet. The natural language enters into the picture in four ways.

1. In the noise words printed on Rapidwrite forms and cards.
2. In the choice of data-names abbreviations.
3. In the choice of long equivalents to the programmers abbreviations.
4. In the noise words printed out at translation time.

1. The printing on forms and cards serves only to guide the programmer to their completion. They could be printed in any language without restriction as to alphabet or size. In fact Rapidwrite forms and cards are already (Jan.62) obtainable in two languages besides English.

2. So long as the programmer sticks to the alphabet A-Z, hyphen, 1-9 and the rules for the formation of data-names he can choose mnemonics which are helpful to him in his own tongue.

3. The long equivalents to the abbreviated data and file names are fed in as a synonym table.

4. The noise words for dressing up the Rapidwrite input to the translator are supplied by the format dictionary.

Firstly suppose that an Italian programmer who knows no English wishes to write a program using Rapidwrite. Then he will use Rapidwrite stationery preprinted in Italian. He will use abbreviated data-names that are Italian nouns or shortened nouns. At translation time he will use the compiler with an Italian format dictionary substituted for the English table so that his printout is readable by himself and his colleagues directly.

Secondly, it could happen that an organization in France wishes to use one of the programs of the Italian group. The synonym table which originally was for substituting long English names for short English names could equally well be used for substituting long French names for the Italian abbreviations. One run through the scanning phase of the translator equipped with this synonym table and a French format dictionary would give a printout which would be directly comprehensible to non-English, non-Italian speaking Frenchmen. Used in this fashion Rapidwrite can truly be described as an international computer language.

CONCLUSION

Anyone who has examined COBOL and liked it, and anyone who has examined COBOL and disliked it, should give it a fresh appraisal considering ICT Rapidwrite as an alternative means of expression. ICT Rapidwrite allows the programmer to write in a fashion convenient to him but at translation time there is provided automatically COBOL Compatibility and Readability even across natural language boundaries.

SEAL: A LANGUAGE FOR BUSINESS DATA PROCESSING

R. J. Ord-Smith and T. F. Goodwin

Standard Telephone Cables, Newport, England

1. This paper describes some of our work in the field of Commercial Languages for computers. We feel the paper to be a valuable contribution on two accounts.

1.1 Considerable effort has gone into producing a language which is very explicit in its data description. We show by an example the method of defining data by a fixed format description.

1.2 We have given a lot of thought to the kind of computer which we believe is best suited to this kind of application. Some of our conclusions about the machine are discussed.

2. The commercial language we have developed is called SEAL *S*tandard *E*lectronic *A*ccounting *L*anguage. SEAL does not attempt to be a full free-form English language representation. It is rather a simplified programming scheme but having an external form more natural to the user than a computer code would be. This external code takes the form of simple English language statements which have to be written into preprinted forms. We have, therefore, aimed at producing a scheme which is easy and natural to the commercial user and at the same time not too far removed from the internal structure of a machine code program. The latter property simplifies the task of compilation.

3. It is impossible in a short paper to give full details of a commercial language and its vocabulary. We have felt it better to illustrate the structure and layout of the language by means of part of a simple application to a commercial process. The example we give is updating part of a sales ledger with information from a journal.

3.1 Our simple sales ledger consists of a set of sales accounts each of which contains a list of invoices, cash payments and credit payments particular to this account.

Our journal consists of a collection of invoices, cash payments and credit payments each filed against an account number.

3.2 Data descriptions have to be constructed for each file and records within files and this is done on preprinted forms like the one shown on facing page.

3.3. Let us fill in such a form for our sales ledger:

Not applicable here

File name: Sales Ledger
Global name:

Serial record	
Name	
Sales a/c	

Rest of this part not applicable.

Symbol	A_1	E_1	E_2	E_3		
Name	A/C Code	Name	Address	Delivery address		
Mode and Size	W/N 5	A/N 4V19	A/N 10V120	A/N 10V120		
Contents						

We have to describe each field of which the sales a/c is made up according to the four defining entities.

3.3.1 *The Symbol* is A, E, I, O or U followed by a subscript. A fields (attributive fields) are fixed length fields which come first and must be processed first in the record. The Symbol describes the type of field (fixed or variable length) and the order in which it is to be processed within the record.

3.3.2 *Name* Our only A field in this example is the particular account number. We call this the a/c code i.e. it is the user's name for this information.

3.3.3 *Mode and Size* describes the nature of the data i.e. numeric whole number, mixed number, fraction, or alphanumeric. W/N 5 means allocate a whole number field of 5 characters.

DATA DESCRIPTION

for

Devised by . . . . . on . . . . . File Name . . . . . .

Global Name . . . . . .

Serial Record	Working Record								Table									
Name	Name	Max Size Words	Max List Elements	Core	Name	Max Size Words	Max List Elements	Core	Range arg 1		Range arg 2		Range arg 3					
									Int.	arg 1	Int.	arg 2	Int.	arg 3				

Symbol												
Name												
Mode and Size												
Contents												

Symbol												
Name												
Mode and Size												
Contents												

Group Names		
Group Name	Field Name	Field Name

3.3.4 *Contents* only specified if a field has a fixed absolute form.

3.3.5 E fields (extendible fields) are of variable length but do not normally vary in length during processing.

3.3.6 Names and addresses are obvious examples and our E1 E2 E3 are of this form.

3.3.7 A/N 4V19 means the field may be of any length between 4 and 19 alphanumeric characters. The fields are packed together as the data is set up.

3.3.8 In this example A1 E1 E2 E3 form the fixed part of the sales account known as the *initial standing data.*

3.3.9 The continuation of the sales account consists of a typical expanding list of invoices, cash payments, and credit payments.

Each set of fields is of fixed length but the number of these fixed sets may grow in the list. The fields within a list are symbolized as I fields (iterative fields).

The I fields describing the invoice details in an element of the list in the sales account now follow.

	I 1/1	I 1/2	I 1/3	I 1/4	I 1/5	
	Item type	Invoice no.	Date	Order no.	Account	
	W/N 2	W/N 9	W/N 6	W/N 9	W/N 7	
	1					

Notice that the invoice details are labelled item type 1. The fixed number 1 which is this field content is therefore written explicitly in the contents row.

3.3.10 Since it is convenient to regard the group of fields I 1/1 - I 1/5 as an entity called "invoice details", we define this group name in the special boxes at the bottom of the form.

Similarly the credit details and cash details are inserted and their group names defined.

The group names are defined as before.

	Group Names						
	Group Name	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
	Invoice details	I 1/1	I 1/2	I 1/3	I 1/4	I 1/5	
see below	Credit details	I 2/1	I 2/2	I 2/3	I 2/4		
see below	Cash details	I 3/1	I 3/2	I 3/3			

I 2/1	I 2/2	I 2/3	I 2/4	I 3/1	I 3/2	I 3/3	
Item type	Date	Credit no.	Amount	Item type	Date	Amount	
W/N 2	W/N 6	W/N 9	W/N 7	W/N 2	W/N 6	W/N 7	
2				3			

3.3.11 There may follow some details for particular accounts which vary from account to account and may or may not be present in a particular account. Statistical information is a typical example. To save space where these records are not present the fields can be described in the preprinted form as O fields (optional fields). We omit these for simplicity.

3.3.12 Finally the end of the account is defined by a set of fixed fields (of which there must be at least one) and these are called U fields (ultimate fields). For example our accounts may end with:

U1	U2	U3	U4	U5
Credit limit	Total debits	Total credits	Date last moved	Balance
W/N 9	W/N 7	W/N 7	W/N 6	W/N 9

3.4 A data description of the journal is also required.

3.4.1 Our journal is made up of invoices, cash details, and credit details. These were formed in the journal in the order they occurred. They may have been sorted into order of account numbers for updating a sales ledger, we are assuming this, but the order of the items invoice, cash, credit is unknown. To deal with the situation these items are given a *global name*

"item" by means of which they can be specified without knowing which is which.

DATA DESCRIPTION

for

File Name: Journal
Global Name: Item

Serial Record
Name
Invoice

Symbol	A1	A2	A3	A4	A5	U1
Name	Item type	A/c code	Invoice no.	Date	Order no.	Amount
Mode and Size	W/N 2	W/N 5	W/N 9	W/N 6	W/N 9	W/N 7
Contents	1					

		Group names		
Group name	Field 1	Field 2	Field 3	Field 4
Invoice details	A1	A3	A4	

Similarly for cash and credit items of the journal.

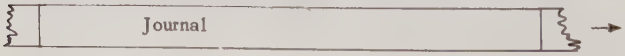
3.5 Let us now suppose that our information is organized in the following way:

A book (stored on magnetic tape) consists of two files:

a commission ledger and
a sales ledger,

with the commission ledger coming first in time on the tape.

Suppose this is called Book 4.
Another Tape contains Book 3 consisting of the journal.

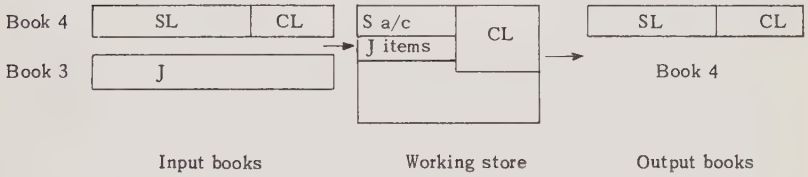


3.5.1 The processing in progress is:

(1) to bring the commission ledger into working storage,
(2) to bring successive sales accounts and journal items into working storage updating sales accounts one by one. At the same time the commission ledger is updated in a random order in working storage.
and (3) to produce an updated book containing updated sales ledger and commission ledger.

The complication is that sales ledger updating is serial and precedes the full updating of the commission ledger.

The output book has therefore to be rewound to place the new commission ledger back on the front.



3.5.2 Let us first consider the program necessary to organize the opening and closing of the books, bringing of information into and out of working storage.

3.5.3 Note one or two things about the preceding program.

(1) The full commission ledger has to be brought into working storage. This means that in the data description of the commission ledger (which we have omitted) the working record part of the form has also to be filled in.
(2) The special operator OPEN AND CLOSE OUTPUT allows space to be left in an output book without permanently closing the book. In this case we are still able to say OPEN OUTPUT later in the program and when we do a rewind is made to the commission ledger part of the book.

PROGRAM SHEET

Program Title — Programmer

Section Name — Date

Page No.

Label	Operator	Operand 1			Adverb	Operand 2			End
		Detail Name		Class Name		Detail Name		Class Name	
	OPEN INPUT	Book 4			[and]	Commission ledger	In	Book 4	.
	OPEN INPUT	Book 3							.
	GET	Commission ledger	In	Book 4	[into]	Commission ledger	In	Working storage	.
	CLOSE INPUT	Commission ledger	In	Book 4					.
	OPEN OUTPUT	Book 4							.
	OPEN AND CLOSE OUTPUT	Commission ledger	In	Book 4					.
	OPEN INPUT	Sales ledger	In	Book 4	[and]	Journal	In	Book 3	.
	OPEN OUTPUT	Sales ledger	In	Book 4					.
		↕		Actual processing follows					
	CLOSE INPUT	Sales ledger	In	Book 4	[and]	Journal	In	Book 3	.
	CLOSE OUTPUT	Sales ledger	In	Book 4	[and]	Book 3			.

Label	Operator	Operand 1			Adverb	Operand 2			End
		Detail Name		Class Name		Detail Name		Class Name	
	OPEN OUTPUT	Commission ledger	In	Book 4					.
	PUT	Commission ledger	In	Working storage	[into]	Commission ledger	In	Book 4	.
	CLOSE OUTPUT	Commission ledger	In	Book 4	[and]	Book 4			.
	GET	Item	In	Journal					.
	GET	Sales A/C	In	Sales ledger	FOR	A/C Code	In	Sales A/C	
	EQUAL [to]	A/C Code	In	Item					
	AND PUT	Sales A/C	In	Sales Ledger					
	IF END GO TO	Error							.
	GO TO	Invoice		[or]		Credit			.
	[or]	Cash							
	DEPENDING [on]	Item type	In	Item					.
Invoice	ADD	Amount	In	Invoice	[to]	Balance	In	Sales A/C	.
	IF	Balance	In	Sales A/C	[is] GREATER	[than] credit limit	In	Sales A/C	
	GO TO	Reject Invoice							
	OTHERWISE ADD	Amount	In	Invoice	[to]	Total debits	In	Sales A/C	.
	CREATE	Invoice details	In	Sales A/C					.
	MOVE	Invoice details	In	Invoice	[to]	Invoice details	In	Sales A/C	.

3.6 We exhibit next a small section of the ledger updating using items from the journal. In this section a test is incorporated to ensure that the balance does not exceed the credit limit of the account.

3.6.1 The CREATE operation is a powerful one which can sert further items in a list. A file on magnetic tape consists of a series of blocks called leading block, several trailing blocks, and an ending block. If CREATE reaches the O and U fields in the ending block it converts the block into a trailing block and makes a new ending block with a fresh copy of the O and U fields.

4. A proposed specification for a computer.

4.1 In considering a computer suitable for commercial language application two distinct requirements have to be taken into account.

4.1.1 The computer at compiler time and its suitability to the special techniques of compilation.

4.1.2 The computer at object time and its suitability to commercial data processing.

Two related facts show themselves of greater importance in both cases, however.

4.1.3 *The variable length strings of characters* which is the form commercial data typically takes.

4.1.4 *The need for dynamic addressing*, i.e. the allocation of absolute addresses only during the actual processing. This is a necessity when dealing with variable length data unless an enormous wastage of storage space is to be tolerated.

4.2 SEAL was originally designed for implementation on an existing computer and, as such, preceded the formulation of a specification for a data processor. The needs of SEAL, however, defined quite clearly the broad concept of such a specification and this is what we want to describe.

4.3 The nature of commercial data, often >50% alphabetic in a given case, dictated the first two points of the specification.

4.3.1 *A character addressed computer with a character as the unit of data. Special characters in the character code indicate beginning and ending of fields. Thus variable length strings are used with the data containing its own information about length.*

4.3.2 *Binary coded decimal arithmetic.* It has been said that the controversy of binary versus binary coded decimal will rage until the perfect computer has been designed!!

4.4 Though the user shall regard information within the computer as a string of characters it is convenient that transfers of information take place in blocks of characters or machine words. The number of characters in a machine word depends on several considerations.

4.4.1 The number of characters conveniently processed in parallel in an arithmetic unit.

4.4.2 The number of characters which conveniently form an address (but see below).

4.4.3 The optimum amount of logic required for word transfers between the extremes of a single character as a word and a large word of many characters.

Careful consideration has led to the third point in the specification.

4.4.4 *A 40 bit word which can consist of 5 7 bit characters + word identification bits or two addresses.*

4.4.5 Since addresses are never used by the external user and are only to be allocated dynamically during operation they are regarded as an internal device and represented in binary. A 17 bit address allows a working store of 131,072 character locations and a machine word can contain two such addresses. Because of the addressing scheme adopted addresses normally occur in pairs so that a machine word ad-

dress pair is a natural unit. The equipment associated with the binary address registers provides the facility for a limited amount of binary operations for logical purposes such as collation.

4.5 *In order that processing variable length strings of characters be a natural process in the computer* (a fact which greatly simplifies the task of the compiler) *instructions are provided which are "cyclic" in form.* These instructions continue to process the string as appropriate until the end of field character is reached.

4.5.1 The precise action of these instructions in the control unit will be clearer later, but a point concerning the manipulation of addresses during the cyclic action must be made here. The address which "points to" the character to be processed is automatically updated to the next character during a cyclic instruction. The next character to be processed does not have to be the next character in the store, however. *A string of characters may form a set of disconnected but within themselves contiguous sections in the store but these are linked together by the addressing scheme and the latter interpreted during execution.* This idea of a *chain store* is not a new one but the facility is usually a programmed one.

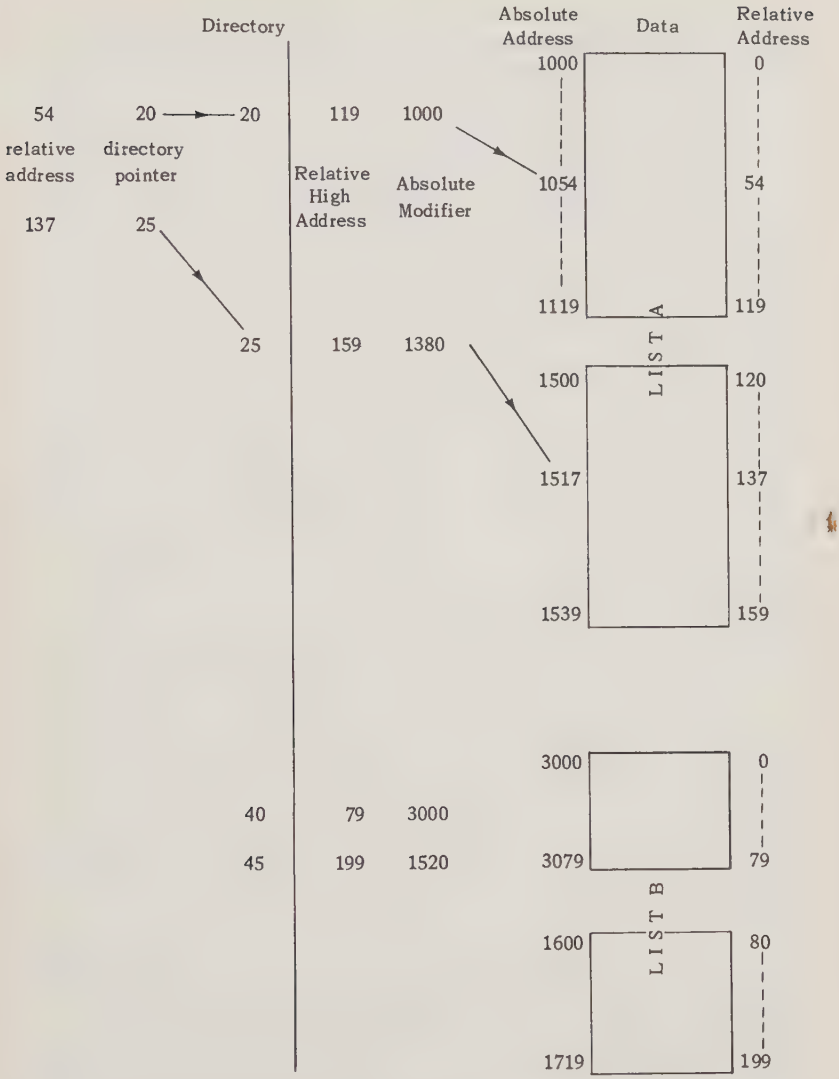
4.6 The chain store addressing scheme. The mechanism of the chain store is, we think, of sufficient interest to be explained in a little more detail.

4.6.1 Addressing of actual data is "indirect". The two addresses used with an instruction are:

(i) the relative address within the record
(ii) a "directory pointer" which gives the absolute address in the directory which in turn will point to the actual section of store containing the data.

4.6.2 The directory, in turn, contains:

(i) the relative address of the end of this contiguous section.

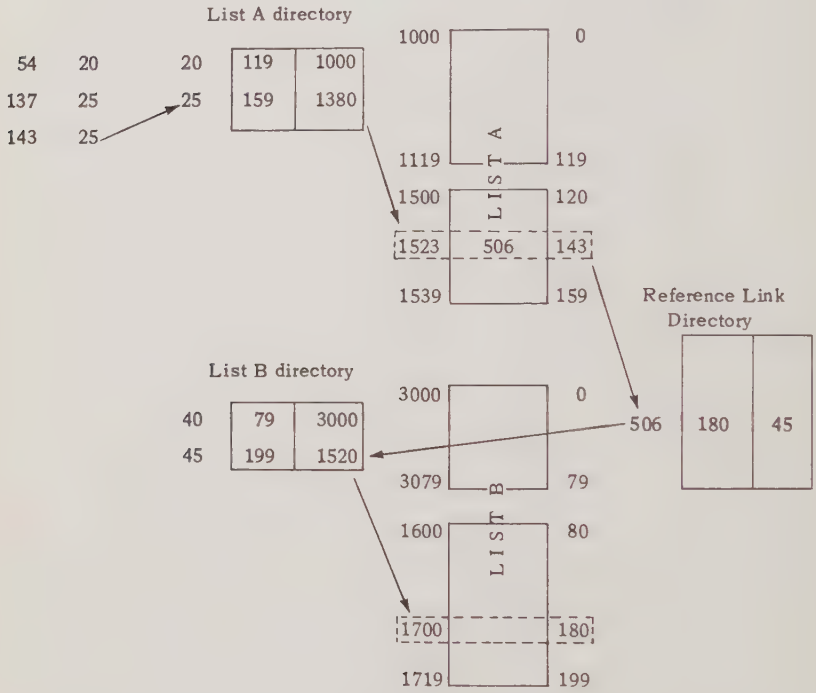


(ii) an absolute modifier which, when added to the relative address, gives the absolute address of the data.

4.6.3 A separate list has a separate directory.

4.6.4 Normally the data contains no addresses but a specially marked address within the data called a reference link, can, via a reference link directory, gain access to information in another list.

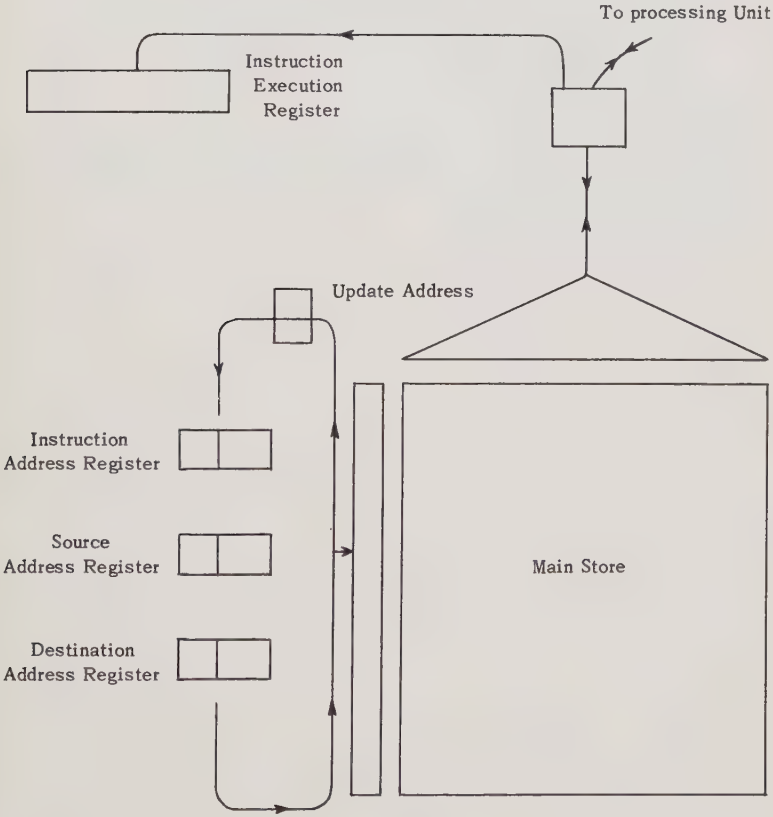
E.g., with the above example suppose location relative address 143, say, in LIST A refers to LIST B we can have:



4.6.5 At first sight extravagant in addresses, the scheme actually saves a considerable amount of storage space. A chain store means that variable length records can be closely packed and this built in scheme saves space and time that a program simulator would take. The facility safeguards against erroneous overwriting in the store which is a real danger when parallel programming is practised with addresses in the hands of the programmer.

4.7 The actual data addresses, then, do not form part of the instructions. A feature of this design is that the directory pointers do not either. *A program contains no addresses at all.* During the execution of the addressless instructions the pointers are assembled and executed from special address registers. The only instructions which directly control addresses are those which initially set the directory pointers from the program directory and those which raise or lower addresses for program jumping purposes.

4.8 The basic structure of the computer is shown in the following block diagram.



4.8.1 During serial processing these relative addresses update automatically. The directory pointer updates only when the relative address exceeds the relative high address in the directory. The address registers form a smaller faster store than the main store and the extraction and execution of the contents of these registers is overlapped with the extraction of the main store contents. Thus access to the main store takes normally a single store cycle time. When processing jumps through the end of a contiguous section, 2 cycle time access is involved. References across lists take 4 cycle times.

4.9 The instruction code and addressing scheme have been tested out in extensive experimental programming and has proved to our satisfaction the advantages of the concept.

The authors are grateful to the Information Processing Division of Standard Telephones and Cables, Ltd. for permission to publish this paper, and to their colleagues and associates who have contributed to the work.

A SYSTEM AND LANGUAGE FOR DATA PROCESSING

R. M. Paine, Manager, Data Processing Services

C-E-I-R (U.K.) Ltd. London, England

The concentration of speech and activity on the two languages of COBOL and Algol has given them in some quarters the reputation of being the complete answer to the expression of the problems of on the one hand commercial data processing and on the other hand of scientific computations. Is this really so? Or have the defenders and disciples of COBOL for instance not been led to make too wide a claim in order to combat similar languages in the field? COBOL is good for several types of data processing applications—especially work involving the systematic up-dating of magnetic tape files and the repetitive printing of large volumes of data from these files. There are some areas, however, in which it is comparatively useless—and such an area is Market Research—that is evaluating into table form the answers on thousands of questionnaires. A similar area to Market Research is the provision of Sales Statistics from orders or invoices. In these two areas a specialized language is required which can more closely and precisely express the requirements of the executives concerned, and the Autostat language and Opal system of C-E-I-R (U.K.) Ltd. have been expressly designed for this purpose. The object of the Opal system is to provide facilities for analysing data, the requirements of the user being specified in a suitable simple language, known as Autostat.

The language is already in use (early work was done on the Ferranti Pegasus) and a compiler has now been written for the IBM 7090, and many problems have been run.

Suppose we wish to question people and find the answers to the following table:

BRAND A

	CANS	SEED	BISCUITS	POWDER	ALL
SHOP					
SERVICE					
ORGANIZATION					
REPRESENTATIVE					
BUYING					
STAND					

That is, information about food for pet animals has to be analysed by brand, with separate tables for the type of food—cans, seed, biscuits and powders. For each brand we wish to count the no. of shops stocking it, whether they were self-service shops or not, whether they were small or large shops. For certain brands we want to know how many shops have a self-service stand, whether sales representatives call on them, and whether they buy through a group or chain of retailers. (Another use might be to find out what are people's view on Britain joining the Common Market, divided by sex, age, social class, income group, political beliefs, political beliefs of parents, no. of children, trade union membership, type of job, area of country, protection afforded to their industry, educational qualifications, travel outside U.K. etc.) The use to which the information is to be put may be fairly trivial, such as or to advance a particular brand of bird seed or soap, or to help serious political decisions.

Stages on Computer

1. Compiling
2. Editing
3. Selection and Counting
4. Statistical Analysis
5. Tabulation

The purpose of the compiling phase is to translate the instructions of the user, given as statements in Autostat language, into computer instructions specifying the classification of the data, how it is to be stored in the computer, what editing processings are to be applied, what counts are to be carried out, what statictical analyses are to be employed, and in what form the results are to be printed.

After compiling successfully—and this may take several runs since the program has its own diagnostic routines for testing the logic of the statements in the Autostat language—data on punched cards is converted to tape by the 1401 and fed into the 7090 under control of the compiled program. Editing operates on the data to pack and add to it, construct weighting tables, and check for errors and inconsistencies. Selection and counting produces a set of tables packed on magnetic tape in a standard form, and statistical analysis can be performed on these results or can be omitted if desired. The tabulation stage produces a magnetic tape which can be transferred to the 1401 for printing, all the format being laid out in the 7090.

But I wish to concentrate not on the computer system or the compiler action but on the language in which to state the requirements.

There are several types of statements used, and each group of statements is preceded by its appropriate heading:

1. DEFINITIONS
2. MULTIPLE ANSWER
3. GROUPING
4. CHECKS
5. WEIGHTS
6. TABLES
7. FORMAT

I will concentrate on 1., 3., and 6.

DEFINITIONS

Assuming that the questionnaires are punched on to cards, we allot to each card item or field an arbitrary label of not more than 12 letters or numbers, which identifies the answer concerned. Information may be recorded in a single hole or several holes in the same or another column—the scheme allows for up to 12 holes in a column. This use of multipunches in a column is not normally dealt with by Cobol, Algol etc. Definition Statements consist of four parts: 1. Label, 2. description of the type of information, 3. the maximum number of allowable answers to the question, 4. the card columns and digits concerned. They are the equivalent to 'Data Description' of COBOL. Each part is separated from the next by a comma, for example take the definition:-

SEX, BIT, 1, CØL 15, 7/8

This means that the label SEX is given to the answer, with two possible groupings defined by whether a 7 or an 8 is punched in column 15. Thus the person is classified as being in the Class SEX = 1 or SEX = 2, i.e. male or female. The description Bit, short for Bittable, implies that the punchings in Col. 15 are of a yes/no category rather than carrying the actual value of a characteristic. The 1 after Bittable says there is only one allowable answer, so either 7 or 8 but not both may be punched, and at least one of the two must be punched—otherwise the card will be rejected.

In this example we had two classes separated by a slash e.g. 7/8, and a slash is what is used to separate classes or different groupings of answer.

Assume we have 4 classes of smokers from the answers on the questionnaires, with Column 25 punched 6 for a frequent smoker, 7 for a moderate, 8 for an occasional and 9 for never smoked. If we had made the definition

SMOKERS, BIT, 1, CØL 25, 6/7/8/9

we should be able to refer later to our four classes as SMOKERS = 1 (i.e. frequently), SMOKERS = 2 (i.e. moderately), SMOKERS = 3 (i.e. occasionally), SMOKERS = 4 (never).

Other information on a card may require the actual value or magnitude of the numbers punched—not just the presence or not of a hole. Thus a person's age say 27 would be recorded in two columns, say 16 and 17, in decimal form, and to distinguish this type of punching from the bittable above we would use DEC in the second part of the definition statement. Thus if we wish to classify respondents by age group a definition like the following could be used:-

AGEGP, DEC, 1, CØL 16-17, 16-24/25-34/35-44/45-64/65-99

The statement defines 5 age groups, and would allocate each card read to a group according to what is punched in columns 16 and 17. Our example of age 27 would be in Class 2, and we will be able later to refer to this class as AGEGP = 2.

Notice here that we have not recorded in the computer the actual age but only the class to which it belongs. If we wish to record the actual value, we adopt the convention of adding a V to our label so that AGEV, DEC, I, CØL 16-17, 16-99 will hold in the computer the actual number punched in columns 16 and

17, as long as the number is between 16 and 99, but will reject the card if the number is less than 16.

Alphanumeric punchings are catered for by writing BCD instead of DEC or BIT.

MORE THAN ONE ANSWER

The above examples have assumed only one answer or possible punching per questionnaire. But if we ask 'which of the following newspapers do you read?'—we must expect several answers, and we must be able to tell the computer how many to expect.

If we said 'which two of the following publications do you prefer'

	Col.
Horror	0
Terror	1
Agony	2
Mystery	3
Lust	4
Gluttony	5

Then the definition READERS, BIT, 2, CØL 5, 0/1/2/3/4/5 would expect exactly two numbers to be punched in COL 5, say a 3 and a 4—if there were less or more the card would be rejected. If an indefinite number of answers up to a maximum of m are expected we use—m in the definition, e.g.

READERS, BIT - 4,CØL 5, 0/1/2/3/4/5

could be the result of asking the question 'State which papers up to a maximum of 4 you read in the last week'.

The card would only be rejected if there were more than 4 punchings in COL. 5.

If any number of answers are expected with no maximum set, then 0 is used in the definition, i.e.

READERS, BIT, O, CØL 5, 0/1/2/3/4/5

You will notice that the groupings though still distinct now overlap—for instance the same people may read both Horror

and Terror. The extent of the overlap may be one of the things the survey is trying to find out.

Multiple Card Questionnaires

I have spoken so far as if there is only one card per questionnaire—but the Opal system allows up to 9 cards per questionnaire. The columns of the second card are referred to as 81-160, the third 161-240 and so on.

CARD COUNT

Most jobs require to have some sort of control total to check the number of questionnaires passed through the system. A definition such as

ALL, BIT, 1, CØL 4, 1,-

would do this for us, and the Label ALL is reserved for this use. The hypen means that any other punching in that column is included in the same class—the only class—as a punch 1 in Col. 4.

TABLE STATEMENTS

Once we have delivered the data we can start to produce useful results.

In many surveys the object is to produce tabulations of the data counted under various classifications. Thus we may wish to count the number of men and women separately in each of a number of age groups so as to produce a table like this:

	SEX		
Age Group	MEN	WOMEN	TOTAL
16-24	157	120	277
25-34	204	320	424
35-44	183	198	381
45-64	142	139	281
65-99	75	84	159

We can specify this by simply counting the data by sex and age group. This is written in Autostat language as

TABLE 1) $ IMP $ AGEGP * SEX + ALL

There are six parts to the statement:

1. Title - TABLE 1) - up to 12 characters
2. A population label (which can be omitted as it is here)
3. The separator $ IMP $ - which stands for IMPLIES
4. A set of *row* labels
5. The separator *
6. A set of *column* labels

The set of row or column labels can be joined by + or comma or slash if required—which will be explained later.

The population label defines one dimension of the tabulation. Thus we may tabulate in different tables information relating to men and women. In these cases we may use population labels such as SEX = 1, or SEX = 2, SOCIAL CLASS = 3, and SOCIAL CLASS = 4, to define the class of people to be counted within the tabulation required, e.g.

TABLE 1) SOCIAL CLASS = 3 $ IMP $ AGEGP x SEX

would only tabulate people in Social Class 3 by Age Group and Sex.

Row labels precede column labels and the separator * means 'by', *i.e.* these rows by those columns.

The + (plus sign) enables us to join tables which have the same row or column dimensions. Thus in the above table "ALL" or the total is treated as an additional column to the table showing the whole population by age group. The separator + allows two or more tabulations to be specified without rewriting those row or column labels which are common.

Another separator is the slash (/) which means 'within', thus A/B in a table statement, means A within B. If Sex has 2 groups and Region 3 groups then

TABLE 3) SOCIAL CLASS = 4 $ IMP $ AGEGP * REGION/SEX

would produce a tabulation for the 4th Social Class of Region within Sex by Age Group, e.g.

MALE

Age Group	Region (1)	Region (2)	Region (3)
16-24			
25-34			
35-44			
45-64			
65-99			

FEMALE

Age Group	Region (1)	Region (2)	Region (3)
16-24			
25-34			
35-44			
45-64			
65-99			

The comma is a third separator which means 'and', and joins classes on the same level. Thus if the columns in the above example were written: SEX, REGION/SMOKERS and there were two classes of Smokers (i.e. those who did and those who did not) the following would be obtained: (the rows would be the same as Example 1 - i.e. age groups).

SMOKERS

Male	Female	Region (1)	Region (2)	Region (3)

NON-SMOKERS

Male	Female	Region (1)	Region (2)	Region (3)

Brackets can be used to get over any ambiguities. The language is very powerful and flexible, for instance the statement:-

TABLE 6) $ IMP $ A/B + C, D/E * F

tabulates the classes of A within those of B against those of F, and then below this table sets out the classes of C and D within E against those of F.

For instance:

		F(1)	F(2)	F(3)	F(4)
B(1)	A(1)				
	A(2)				
B(2)	A(1)				
	A(2)				
E(1)	C(1)				
	C(2)				
	C(3)				
	D(1)				
	D(2)				
E(2)	C(1)				
	C(2)				
	C(3)				
	D(1)				
	D(2)				

It is possible to provide output in the form of percentages in the output phase of the OPAL system by dividing each table element by the element in the ALL row or column and multiplying by 100.

GROUPING

Grouping statements allow the user to create a new label or class from those which have been previously defined.

AGEGP = 2 $ AND $ SEX = 1 $ IMP $ 1 $ INTO $ MARKET

This statement says that if the respondent is in Age Group 2

and Sex Class 1 then he forms Class 1 of a new label called Market.

Similarly

```
(READERS = 1 $ OR $ READERS = 2) $ AND $ SOCIAL
  CLASS = 4 $ IMP $ 3 $ INTO $ UPPERHABIT
```

means that if the respondent reads either Horror or Terror (Reader Class 1 or 2) and is also in the upper class (Social Class 4) then he forms the 3rd class in the Habits of the Upper Class.

The new labels MARKET and UPPERHABIT can now be used in the TABLES statements.

A complete example of a table is shown in Appendix One and Appendix Two.

CONCLUSION

There are many other parts of the Autostat language such as weighting which have not been covered in this paper. But sufficient has been written to show that the language is very powerful and well-suited to its purpose of expressing market research and allied problems.

It also shows us that there may be a need for several special purpose languages to cope with problems for which COBOL or ALGOL were not designed. C-E-I-R (U.K.) Ltd will be very pleased to provide further information.

APPENDIX ONE

To produce the answers to the Table shown in Appendix Two the following statements have to be made:-

DEFINITIONS

```
REGULARITY, BIT, -1, COL 27, 1/2/3
TIME, BIT, -1, COL 28, 1/2/3/4/5,-
USEA, BIT, -1, COL 73, 1/2/3
USEB, BIT, -1, COL 74, 1/2/3
ESIC, BIT, 0, COL 14, 12
ALL, BIT, 0, COL 1, 1, -
```

GROUPING

```
REGULARITY = 1 $ IMP $ TIME $ INTO $ RAGTIME
```

```
USEB = 1 $ AND $ (USE A = 1 $ OR $ USEA = 2) $ IMP $ 1 $
  INTO $ TRYA $ ALSO $ 2 $ INTO $ TRYA
USEB = 1 $ AND $ USEA = 3 $ IMP $ 1 $ INTO $ TRYA $
  ALSO $ 3 $ INTO $ TRYA
USEB $ G = 2 $ IMP $ 2 + USEB $ INTO $ TRYA
```

MULTIPLE ANSWERS

```
RAGTIME (1, 5)
TRYA (2, 5)
```

INDIVIDUAL WEIGHTS

```
ESIC = 0 $ IMP $ 0.0 $ INTO $ TABLE 21
```

TABLES

```
TABLE 21 $ IMP $ ALL + REGULARITY + RAGTIME * ALL +
  USEA + TRYA
```

APPENDIX TWO

Habit Buying—Duration
—By Experience of Coffee Brand A and Brand B

	All	EXPERIENCE OF BRAND A			EXPERIENCE OF BRAND B				
					Use			Tried But Do Not Use	Not Tried
		Use	Tried But Do Not Use	Not Tried	Total	Tried A	Did Not Try A		
Base of percentage									
All who ever serve instant coffee	1145	323	647	274	748	507	239	380	18
Brand to be bought next time has always been regular brand	63	38	65	77	79	77	84	31	50
Some other brand was brought regularly before	22	48	19	9	11	14	6	43	33
No brand in particular was regular before	15	14	15	14	9	9	10	26	17
Brand to be bought next time has always been regular brand and has been buying it for -									
1 Year or less	3	3	2	3	3	2	3	3	-
2-3 Years	10	13	9	10	10	10	11	11	17
4-5 Years	10	12	8	11	10	10	11	8	6
Over 5 years	37	8	43	47	51	51	52	8	17
Don't know/can't remember	4	2	3	6	5	4	7	1	11

AN AUTOCODE FOR TABLE MANIPULATION

J. C. Gower

Rothamsted Experimental Station,
Harpenden, England.

INTRODUCTION

Statistical calculations frequently require the manipulation of multiway tables—examples occur for instance in the analysis of experiments and surveys or when fitting constants to non-orthogonal data.

An n-way table has factors 1, 2, ... n and for $i = 1, 2, \ldots n$ the ith factor has a number of *levels*, 0, 1, 2 ... $(L_i - 1)$. It is convenient to store the contents $x(l_1, l_2, \ldots l_n)$ of the cell with levels $l_1, l_2, \ldots l_n$ in an address:

$$s(l_1, l_2, \ldots l_n) = A + l_1 + l_2 L_1 + l_3 L_1 L_2 + \ldots + l_n L_1 L_2 \ldots L_{n-1} \quad \ldots\ldots (1)$$

where A is the address of the first cell of the table.

In practice the contents of the table need to be picked up in several different orders, depending on which factor or factors are of interest, for use in some arithmetical or organisational operation. Existing autocodes are suitable for operational procedures, but are not suitable for the counting and allied processes concerned directly with the tabular structure. The difficulty is most easily illustrated by an example.

Suppose we have a 3-way table with $L_1 = 3$, $L_2 = 2$, and $L_3 = 2$ and we want the totals for the first factor over the other two factors to overwrite x_0, x_1 and x_2. We could write (in Mercury Autocode for instance):

```
i = 0 (1) 2
    a = 0
    j = 0 (1) 1
        k = 0 (1) 1
            s = i + 3j + 6k
            a = a + x_s
        repeat
    repeat
    x_i = a
repeat
```

If we want the totals for factor 2 we would have:

```
j = 0 (1) 1
    a = 0
    i = 0 (1) 2
        k = 0 (1) 1
            s = i + 3j + 6k
            a = a + x_s
        repeat
    repeat
    x_j = a
repeat
```

If we want the two-way table for factors 2 and 3 totalled over factor 1:

```
j = 0 (1) 1
  k = 0 (1) 1
    t = j + 2k
    a = 0
    i = 0 (1) 2
      s = i + 3t
      a = a + xs
    repeat
    xt = a
  repeat
repeat
```

If therefore all three types of total are required, three separate sets of loops are necessary, which is very laborious. Some autocodes can do rather better. However, as all three totals will usually be required in the one problem it is desirable to be able to produce them in one small loop of instructions, merely changing a parameter specifying which factor totals are required. A further difficulty can be seen to arise in programmes where the number of factors may be variable. These difficulties can be overcome in terms of a concept called *scanning* (Gower, 1962).

SCANNING

To scan over factors i, j, k, for example, the levels of these factors take in turn the values 0, 0, 0,; 1, 0, 0,; L_{i-1}, 0, 0,; 0, 1, 0,; 1, 1, 0,; etc. Scanning is completed when all $L_i L_j L_k$ addresses have been produced. It can be arranged that the levels of factors not appearing in a scan are undisturbed. For each set of levels the corresponding address $s(l_1, l_2 \ldots l_n)$ must be evaluated and a link must be provided to the operational part of the calculation where the tabular value x_s is usually an operand. A second scan can (and often does) appear as part of the operation of another. The only restriction is that the two scans must be over disjoint factor sets. Often two such sets account for all the

factors of a table. This leads to a special notation in the autocode set out below.

We also require a simple way of denoting factor sets for computer storage. A binary notation is often most convenient where i, j, k, ... is stored as $I = \pi + 2^{i-1} + 2^{j-1} + 2^{k-1} + \ldots$; π being a special marker to denote a factor set. If $\pi = 0$, I is taken to represent the single factor i = I. Thus $I = \pi + 3$ denotes factors 1 and 2, but I = + 3 denotes factor 3. This convention is often of use when scanning over factors 1, 2, ... n in turn, since the factor code is often available from an ordinary count outside the scanning operations.

AUTOCODE INSTRUCTIONS

The following autocode instructions are suggested as an addition to the Manchester University/Ferranti Autocode systems (Brooker, 1958). In Autocode

a, b, c, d, e, f, g, h, u, r, w, x, y, z are Variables (Floating Point Representation)

i, j, k, l, m, n, o, p, q, r, s, t are Indices (Integer Representation)

I have tried to keep to these definitions in the following, and where I have diverged, context should make the position clear. I have also taken a few liberties with Autocode in the examples, e.g. in allowing the use of the logical operations which are very useful when working with factor sets.

(i) set up (n, f_1)	Set up the scan subroutine for the number of factors given in store n, the levels of which are stored in $f_1, f_2, \ldots$ etc. Here n may be an index or an actual integer.
(ii) set up (p, q, r, ...)	This is an alternative to (i), giving the actual numbers of factor levels or the names of the stores containing these levels.
(iii) i = (p, q, r, ...)	i is a factor set, with marker $i = \pi + 2^{p-1} + 2^{q-1} + 2^{r-1} + \ldots$

(iv) scan (i) / rescan — These two sets of instructions are parallel to the [i = p(q)r, repeat] sequence already existing in Autocode where, however, i may represent a set of factors preset by (iii).

scan not (i) / rescan — When i has no factor set marker π, it is interpreted as the ordinal number of a set. scan not (i) calls for a scan over all the factors of the current table *not* present in i. When "scan" is encountered an address is evaluated and stored in store s; when "rescan" is encountered the scan count is increased and the correct link determined. This link is the next order when the current scan is complete and is otherwise the scan order of the preceding uncompleted scan.

(v) address set ($i = \alpha$) — This replaces the level l_i of factor i by α; α may be an index, a variable or an integer, and i may be an index or an integer. After replacement the new address given by the scan count is to be calculated and placed in store s. If i has been set up by (iii) and involves more than one factor (say, 1, 3 and 4), the corresponding levels will be set if α is an index or constant ($l_1 = l_3 = l_4 = \alpha$), but when α is a variable f (say) then $l_1 = f_1$, $l_3 = f_3$, $l_4 = f_4$.

(vi) address ($i = \alpha$) — This is similar to (v) but replacement does not occur; the address corresponding to α is placed in s.

These instructions allow a limited number of table operations to be performed.

Example (i)

A simple example is to print all the one-way tables of totals for the factors of a given table:

```
set up (n, f1)
i = 1 (1) n
    scan (i)
    a = 0
    scan not (i)
        a = a + xs
    rescan
    print (a)
    rescan
repeat
```

If the count is $j = 1\ (1)\ 2^n - 1$ and $i = \pi + j$ the above loop will give all the two-way, three-way, etc. tables in addition to the one-way table. If instead of printing totals, we wish to plant them, the storage locations in which they are to be planted become relevant. These often play a central part in the calculations and the autocode should be able to operate on tables which include some or all their margins. To do this we redefine

$$s(l_1, l_2, \ldots l_n) = l_1 + l_2(L_1 + 1) + l_3(L_1 + 1)(L_2 + 1) + \ldots + l_n(L_1 + 1)(L_2 + 1)\ldots(L_{n-1} + 1) \quad \ldots\ldots (2)$$

with the convention that $l_1 = L_1$, $l_2 = L_2, \ldots$ etc. gives the marginal addresses for factors 1, 2, ... etc. The alternative convention that $l_1 = 0$ etc. denotes a margin, has certain advantages and is in fact more convenient in the example of fitting constants given later on. The convention adopted here has the advantage that marginal means etc. of tables are easily printed in their conventional positions using a straight scan.

When a table is stored with its margins we sometimes require to scan over some factors without including their margins—this is called an *exclusive* scan. The autocode thus requires the following additional orders:

(vii)	set up margin (n, f_1) set up margin $(p, q, r \ldots)$	These are complementary to orders (i) and (ii). The scan count is set to evaluate the address in s according to

formula (2); i.e. it is set to work on tables with marginal values.

(viii) exclude (i) Placed before any "scan" order this gives exclusive counts over the factors given in i. At the end of such a scan sequence i is automatically reset for inclusive counts.

Example (ii)

Thus to calculate and store the marginal totals of a table starting in x_0 the following instructions are required:

```
set up margin (n, f1)
i = 1 (1) n
    scan not (i)
    a = 0
    exclude (i)
        scan (i)
        a = a + xS
        rescan
    address (i = f)
    xS = a
    rescan
repeat
```

In this example note that "address (i = f)" gives the margins for the factor set i.

Example (iii)

A more complicated example illustrating the use of tables with margins arises in the fitting of constants to multiway tables containing observations with unequal weights. The method of successive approximation devised by Stevens (1948) is used. A worked example of a two-way table will be found in Yates, (1949).

The procedure is most easily understood for a two-way table. Two sets of quantities are involved, namely the weights w of the individual cells of the table and their marginal totals, and the deviations y from the weighted mean of all the observations, together with their corresponding marginal weighted means. Suppose these are arranged as in Table I, the two factors being denoted by P and Q, with P and Q levels respectively. The marginal total for level p of the weights is denoted by $w_{p.}$; similarly $y_{p.}$ denotes the weighted marginal *mean* of y_{pq}. Thus

$$y_{p.} = \sum_q w_{pq}\, y_{pq} / w_{p.} \qquad \ldots(3)$$

w_{00}	w_{10}	...	$w_{P-1,0}$	$w_{.0}$
w_{01}	w_{11}	...	$w_{P-1,1}$	$w_{.1}$
$w_{0,Q-1}$	w_{1Q-1}	...	$w_{P-1,Q-1}$	$w_{.Q-1}$
$w_{0.}$	$w_{1.}$		$w_{P-1.}$	$w_{..}$

Table I

For each cycle of the approximation the following quantities are successively calculated:

$$\left.\begin{aligned} z_{p.} &= y_{p.} - \sum_q w_{pq} z_{.q} / w_{p.} \quad p = 0, 1, \ldots P-1 \\ z_{.q} &= y_{.q} - \sum_p w_{pq} z_{p.} / w_{.q} \quad q = 0, 1, \ldots Q-1 \end{aligned}\right\} \qquad \ldots(4)$$

It can be shown that these z values will converge to the constants that it is desired to fit. In the first cycle of the approximation z values will not be available for the first set of equations, and we therefore take $z_{.q} = y_{.q}$ initially. The z's are stored in the relevant parts of the two-way table (here the margins arranged as the other tables for w and y). As each new value of each z is computed it is used to replace the corresponding previous value. In order to provide the initial values for z the total table of the y's is planted in the z table.

This procedure can be generalized for three or more factors and for cases in which it is required to fit constants corresponding to the marginal means of two or more way tables. The actual equations used for successive approximation are of the form:

$$z_A = y_A - \sum_D \sum_F z_D w_E / w_A \qquad A = A_1, A_2 \ldots A_q \qquad \ldots(5)$$

Here the A represent control words of factor sets which specify the effects to be fitted, q being the number of sets. It should be noted that the equation for a given value A_i of A in fact represents a whole set of equations and corresponds to one of the equations (4). Thus to fit constants corresponding to the row and column means of two factors P and Q of the two-way example above we have $A_1 = 10$ and $A_2 = 01$. In a three-way table in which constants representing the effects of single factors only are to be fitted the necessary control words will be 100,010,001. If what are known as two-factor interactions are also to be fitted the control words will be 110,101,011. There is in this case no need to specify single factor control words since the single factor effects are included in the constants specified by the two-factor control words.

The control words D, E, F still require to be specified. These are derived from the A control words in the following manner:

If A_i is the current value of A then:

D is taken to represent in turn each of the A except A_i.

E contains all factors in the current D or A_i or both.

F contains all factors which occur in the current D but not in A_i.

The computation for a particular value A_i of A can be performed by the use of two scans. The outer scan scans over all levels of the factors contained in A. The outer summation (over D) is obtained by taking A equal to $A_1, \ldots A_{r-1}, A_{r+1}, A_{r+2}, \ldots A_q$ in turn, each inner summation (over F) being performed by a minor scan.

A programme to do these calculations with the iteration loop written out in full is given below in Table 2.

OPERATIONS ON MORE THAN ONE TABLE

To round off the autocode, instructions dealing with the operations on more than one table must be provided. Such operations are required in survey analysis (Yates & Simpson, 1960, 1961), and also in the analysis of experiments. At Rothamsted we have specialized autocodes (based on scan subroutines) for this type of work. Most, if not all, situations can be covered by operations on two tables $A_1(x)$ and $A_2(y)$ to provide a third table $A_3(z)$. If all these tables are of the same dimension a simple scan will suffice (see for instance the previous example) but with tables

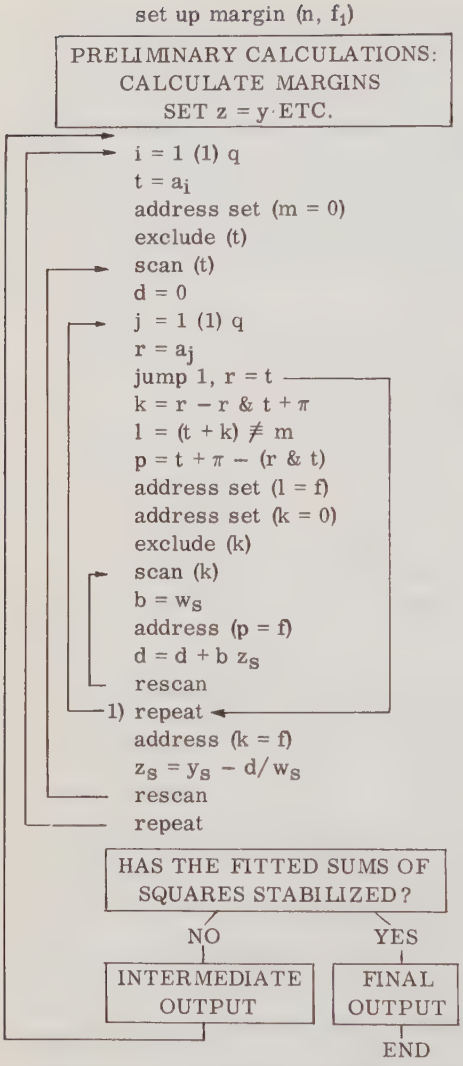


Table 2. A programme for Stevens' method of fitting constants.

$$z_A = y_A - \sum_D \sum_F z_D\, w_E / w_A \qquad A = A_1, A_2 \ldots A_q$$

If A_i is the current value of A then:

D is taken to represent in turn each of the A except A_i.

E contains all factors in the current D or A_i or both.

F contains all factors which occur in the current D but not in A_i.

of different size even simple operations require definition. If a represents a set of factors common to A_1 and A_2, whilst b is a set present in A_1 but not A_2, and c is present in A_2 but not A_1, then A_3 may be classified by the set of factors, a, b, c. Further sets of factors u and v may occur in A_1 and A_2, respectively, which do not directly enter the calculations. These must be set to preassigned levels; usually marginal totals (or means) are required, denoted by U and V below. A very general operation on two tables may then be written:

$$F(X_{a,b,U}; Y_{a,c,V}) = Z_{a,b,c}. \qquad \ldots(6)$$

This is to hold for all levels of factors in a, b, and c.

Expression (6) can be computed by using a separate scan count for each table, but one count with three separate address calculations will suffice if the factors are in the same order in each table. This requires that the same factors occur in the same position in all three tables, with the proviso that the number of levels of the non-existent factors (c in A_1 and b in A_2) are understood to be set to zero for the table concerned. In this way we have a simultaneous scan which produces three addresses r, s, t, one for each table.

In autocode two further facilities are required:

(ix) ...1 (...)	Scans may be numbered with a 1, 2 or 3 immediately preceding the brackets, in any of the previous orders, when more than one table is involved. The addresses of scan 1 are given in r, scan 2 in s, and scan 3 in t. This implies that any unnumbered scan is interpreted as scan 2. This could be avoided if Autocode allowed suffices to indices so that s_1, s_2, s_3 would then be convenient for the addresses.
(x) Scan 1, 2, 3(i, j, k)	A simultaneous scan over factors given in i for scan 1, j for scan 2, and k for scan 3.

DISCUSSION

In calculations requiring a scan most time is spent in picking up the required values, so that for these autocode instructions to be efficient the scan must be made fast. Gower (1962) has shown

how this can be done, either by using a packed count or not. If a packed count is used the address evaluation of (1) or (2) is equivalent to evaluating the binary equivalent of a multiple radix number. Some machines have multiple radix instructions available, but these are often very slow compared with the other arithmetic instructions.

If the fast store is not sufficiently large to hold the whole of the tables being operated on, further problems of efficient timing arise, which will not be discussed here.

If a scan procedure were written for use with ALGOL programmes, for efficient working, it would again be advisable to write it in machine code. Such a procedure could of course be used as a basis for more general procedures as are required for example in the survey programme.

The advantages of scan are that it greatly simplifies the writing of general programmes which have to cope with varying numbers of levels and factors. Even with programmes written for fixed numbers of levels it is a convenience.

We are investigating the possibility of including scan operations in the improved version of Mercury autocode which is now being constructed. Their use should not greatly affect the time factor compared with programmes written in machine code, whereas it should be very much faster than equivalent programmes written in existing autocodes.

REFERENCES

1. Brooker, R. A. (1958), "The Autocode Programmes developed for the Manchester University Computers", The Computer Journal, Vol. 1, p. 15.
2. Gower, J. C. (1962), "The Handling of Multiway Tables on Computers", The Computer Journal, Vol. 4, p. 280.
3. Stevens, W. L. (1948), "Statistical Analysis of a non-orthogonal tri-factorial experiment", Biometrika, Vol. 35, p. 346.
4. Yates, F. (1949), "Sampling Methods for Censuses & Surveys", London: Griffin, p. 137 (3rd Edition, 1961).
5. Yates, F. & Simpson, H. R. (1960), "A General Programme for the Analysis of Surveys", The Computer Journal, Vol. 3, p. 136.
6. Yates, F., & Simpson, H. R. (1961), "The Analysis of Surveys: Processing and Printing the Basic Tables", The Computer Journal, Vol. 4, p. 20.

PANEL DISCUSSION
Reflections from Processor Implementors on the Design of Languages

Moderator:	H. ZEMANEK	(Austria)
Members :	O. J. DAHL	(Norway)
	E. W. DIJKSTRA	(Netherlands)
	H. D. HUSKEY	(U.S.)
	P. NAUR	(Denmark)
	M. PACELLI	(Italy)
	M. PAUL	(Germany)
	K. SAMELSON	(Germany)
	G. SEEGMÜLLER	(Germany)
	W. L. van der POEL	(Netherlands)

ZEMANEK (Austria)—It seems to me that the following points could be considered in this panel discussion: the first concerns the reduction of compiler size. There are three different ways of doing this. The first and very primitive one is to exclude a part of the language. The second and, I believe, the best way, is to make the languages more homogeneous. The third that should be discussed is matching to the machine language. Generally, my feeling is this should not be done too much, but there are certain reasons for doing something in that direction.

The second point would be the description of compilers in a language or in the same language. Now there seems to be the wish to have a clear description and as few as possible interpretations. It seems that a clear language has a clear description, and so it is the basic wish to have a clear language.

Another point would be that a more general language has a shorter processor, so a more general language would be better suited in this sense.

And the last point for discussion could be psychological aspects. The implementor is the first to deal with the language outside the people who have made it, and he is the first to feel the psychological aspects of the language.

In order to commence the discussion, I should now like to ask a few of the panelists to make some comments.

DIJKSTRA (Netherlands)—I am all in favour of short compilers, short in both senses: in the number of instructions they contain and in the time compilation takes.

With regard to the primary task of compilation, to bridge the gap between the process description, on the one hand, and the machine, on the other hand, we wish to shorten this gap. We can do this from both ends. Now, in deciding what way you choose to shorten the gap between the two processes, I have a very strong opinion on that. One may regard a given machine as the master in the relation language-machine, and regard the language only as a servant to the machine enabling you to get problems into it. I prefer the opposite point of view, because if we regard the sequence machine-language, I can continue this sequence by machine-language-man. Man is concerned primarily with the language and not with the machine, and therefore I prefer to regard the machine as the servant of the language. So, in my opinion, the main thing we have to look at is the language, and if we have a language which satisfies our needs and our

tastes, then the only thing we have to do is look for a good machine in order to implement it. The moral cause of this is that the implementor should have as little influence on the design of the language as possible; particularly so if his observations are connected with the special machine he has in mind for implementation. So, one of the ways mentioned by Zemanek is like putting the clock back a couple of years, that is, trying to make the compiler shorter by adapting the language to the machine language. This seems to me exactly the wrong thing.

ZEMANEK (Austria)—Thank you. It was not in opposition to me. I was making almost the same point, that we need some kind of discussion.

NAUR (Denmark)—Speaking purely as an implementor, clearly you want to get the maximum of beneficial effect from the minimum of effort and there are two points which, as far as I can see, have not been mentioned and which I would like to mention. The first is that the implementor would very much like to have a very complete description, including such facilities as manuals for the user, introduction exercises, and so on. And the second point, which again is fairly obvious, is that of course the implementor would like to have as few languages to implement as possible.

HUSKEY (U.S.)—The things that I am going to say now were said yesterday in my speech, but I will give you a two-minute rundown in case you have forgotten.

The points that I mentioned were first of all: bootstrapping techniques as a means of getting a system transferred from one computer to another. And in this bridge-gapping technique, whatever the major languages, you are interested in an as simple as possible bridge-gapping language, so that you minimize the manual effort in this transfer process. Therefore, an adequate language to describe compilers would, for example, have single subscripted variables in it. There would be no local variables, all variables would be universal, there would be no arguments to define in procedures, and even no "for" statements. This simplification, of course, makes the whole translation process much simpler. Perhaps even conditional statements without any alternative parts. Even this bridge-gapping system should be written in its own language. This would make it possible to transform a computer by changing its generating portions and

running on your first computer, and then you have got an object program for the new computer with a minimum amount of change, primarily with respect to primitives dealing with input/output. This will give you the bootstrapping system on the second computer, and then this can be used to transform any language written in this bootstrapping language into a system on the new computer. This bridge-gapping system will again perhaps be a two-stage, at least conceptually, in the pre-processor that simplifies, identifies, and in the translation portion that will process the same types and develop fixed location object programs, the whole system being run on a load-and-go technique. I think that summarizes my position.

PACELLI (Italy)—I believe that programming languages should be able to express comments in declarative form as much as possible, so that a single comment can be translated in the more efficient form, depending on the specifically chosen machine.

VAN DER POEL (Netherlands)—I have something to say about the matching of machine language and the psychological effects of it. I think we must not be too afraid of making a machine language which is a little bit more machine-like. We are always, in our days, making our artificial languages as anthropomorphic as possible. We always want to conform these usual standards of mathematical formula, but I think this is against the whole tendency of automation. I mean, we must not be afraid of designing a language a little bit more machine-like. There are other examples. For instance, we have learned to dial the dial of a telephone: of course, it is designed for our finger to be put into the hole, but for the rest of it, it is purely machine-like depending on the impulses. So have we also learned to drive a car: a gear box is not designed for ease of operation. I have to learn it. And I think there is nothing against designing machine languages which are different from what we usually do. I have not said that we have to speak the machine code, of course - it has got to be a compromise.

DIJKSTRA (Netherlands)—If I may make two comments in the push-down order. First of all a short remark with respect to van der Poel's remarks. O.K., but I am very glad that something like *automatic* gear shift has been invented. Of course, things have to be in the machine, which has to do certain things, and we have to cope with it. Nevertheless, the effort in improving

the machinery is very urgent, and in this respect I should like to switch over to the remarks made by Huskey. The bootstrapping technique which enables you to switch over from one machine to another is very attractive, of course, but I have the idea, perhaps I am wrong, that translation techniques are heavily influenced by the structure of machines. And if bootstrapping techniques really become generally accepted techniques all over the world, it may have the effect that, if somebody turns up with a really new machine, nobody would care to use it, because the translator he could construct from his older ones by bootstrapping would be very inefficient with respect to the new machine. So, I am afraid the bootstrapping technique will slow down machine development.

SEEGMÜLLER (Germany)--To aim at a common language, you should take into consideration, among other things, floating point arithmetic, I believe. We have now reached a certain stage in implementation techniques, in part owing to the great stimulation which comes from languages so highly structurized as, for instance, Algol. We should select the elements of the language so that they should be implementable by means of known techniques which lead to efficient object programs not only for machines which have only subroutines for floating point operations, but also for machines, for instance, which have built-in floating point arithmetic. This would, in my opinion, be a great help in the way to common acceptance. On the other hand, it is of course desirable to carry on research with respect to new techniques for language elements which cannot up to now be converted into efficient object sequences. But such elements should become part of the language only if a good technique is known and if there is a common feeling that this feature is desirable at all. A very important point of view is also to have a language which consists of elements all of which may be implemented efficiently by the same technique, because, in general this produces not only efficient object programs, but also leads to a very clear and efficient concept for the compiler.

HUSKEY (U.S.)--I think there are a couple of points I need to answer here. The example of the gearbox in automobiles I think is an extremely good example to indicate what may happen in our field. It simply illustrates the point that when you have to do with human beings you have to realize that they are devices that operate on a minimum energy basis: they do not do anything they

do not have to do. Therefore, people manufacture automobiles in which you do not have to shift gears even though the one with the gear shift, to your knowledge, might be somewhat more economic.

Getting down to the point that Dijkstra made here, it seems to me that the best example to refute his position is back in the days when we moved around in wagons and the question of using automobiles came up. Then we would be afraid to use automobiles, because that might prevent the development of other points.

ZEMANEK (Austria)—I have the first question from the audience.

MATTHEWMAN (U.K.)—I have one point connected with what Dijkstra started the discussion with. He says that we wish to have a language which we like, which expresses what we want to do, and then we want to look for a machine on which we can implement this. Well, this may be a very good thing, I am sure we all agree, as a long-term project. But, if we have a language which requires 3,000 orders to translate, and we have then got available 2,000 orders of storage base, we are not going to be able to translate this language. Surely, we want, as an intermediate stage, something that we can do now.

SCHWARTZ (U.S.)—I find that some of the comments disagree with some of the experiences we have had with our particular language in compilers. I thought I might comment on these. We have had something like over 500 to 700 people using our language and compiler for some years - this is Jovial. Some of these people have used nothing but this language for the past three years. So we had plenty of opportunities to get both friendly and unfriendly comments on what people like or do not like. First of all, people want everything, however obviously they cannot get it. So the first thing we find is, no matter how bad things are, most people, when faced with using this type of language, want a good, powerful language. Even in the midst of their troubles with compilers they are asking for more in the language.

We have had five versions of Jovial: Jovial 0, minus 1, 1, 2 and 3, which shows how people have asked for more and more. The second thing, they wanted the correctness of the compiler; they go crazy when the compiler produces bad output. Thirdly, they want efficiency of the output code; they want their object programs to run reasonably fast. Lastly, they are interested in

the speed of the compiler. Now, as long as the compiler's speed is within reason, that is, as long as they get their runs back within a normal lapse of time, they are satisfied. As to whether it took 2 minutes or 15 minutes on the compiler, they are not too concerned. Of course, if it took 3 days to accomplish, then we might hear their cries. So, first of all, they do not want to sacrifice the language for the compiler. Secondly, wherever possible, machine independence has been valued. As I pointed out the other day, our initial efforts in this were quite large; it took people a long time and a lot of money in computer time. However, now that we have these things, and we have a very good mechanism for going from one computer to the other, this expense is cut considerably. This would not have been possible if we sacrificed some of the generality or computer independence. So our position is: do not sacrifice the language for the speed or efficiency of the compiler.

STRACHEY (U.K.)—The title of this discussion I think has something to do with feedback from the designed compilers to the language. It does not have anything to do with programmers; it has essentially to do with what efficient compiling ought to be in the language or what changes there should be. Now, I am going to confine myself to this part, about what should be in the language, and in passing I may say it is rather a notable fact that so far we have had practically no discussion about the content of these languages. We have discussed how to work syntax and various things like this, and how to compile them, but we have had practically no discussion about what should be in them.

Now, it seems to me that, from the point of view of compiling, the things that should be in the language are two: one is that it should be possible to make optional comments, in order to assist the problem of compiling, when you know that you have an awkward compiling job to do. Now, the particular difficulties appear, I think, when you have very large problems or very small machines. In fact, when the problems are rather badly matched with the machine, and you are concerned with problems of the data sequence (the order in which the data is accessible), it is possible to make a lot of helpful comments, which can be used by a compiler which is adapted to do it and be neglected entirely if you compile the program with a machine which does not need this facility.

Another point. When you are assembling a program, writing in a large system program, you have a very considerable physi-

cal editing job to do to get your bits of program in the right order and stuck together. I think it is essential for any computer language to be usable as a working language instead of merely a communication language for human beings. I think it is very important that some editing facilities should be in the language. For example, in an Algol program it should be possible to state, on the paper, "insert here declarations for the following procedures", instead of physically having to rewrite them down. This sort of merging procedure or paper editing would make it considerably easier actually to run large programs.

VAN DER POEL (Netherlands)—I should like to make a delayed comment to Schwartz. I like the idea of the Jovial very much, that you can build up, by bootstrapping techniques, object programs for other computers. But I still doubt very much whether it can be done for any type of machine. I mean, the machines for which Jovial works are all alike in their basic structure: they have the usual set of arithmetical operations; most of them are one-address codes; they have the usual set of index operations. But I doubt whether it can be done for unusual machine structures, like Gamma 60 or X 1 - just to name a few of them. And in such a case, the structure of the machine is very unconventional. Then I doubt very much whether an efficient object program will be generated.

DIJKSTRA (Netherlands)—I agree completely. The fears expressed by van der Poel are mine.

SCHWARTZ (U.S.)—It is true that the class of computers for which we have written compilers is all binary. They are rather large, and they do fit into a fairly general similar pattern. However, we have looked at this; we have talked about, for example, implementing a compiler for the 1410, which is, of course, a more radical change than we were used to. One thing we concluded is that in writing a program on a 1410, one would not want to use all the facilities of the language; but, barring that, the techniques of compilation in many of the areas are the same in the Jovial statements for doing the part which I described as producing the intermediate language, which does not change. The language is the same for expressing the way of analyzing an arithmetic statement. When you get to the point of turning out clear enough instructions, of course, this changes on all computers anyway, depending on whether there is one comple-

ment, two complements. We have not had to do it yet, but we have investigated it and seen that it is possible.

SAMELSON (Germany)—I cannot really talk about bootstrapping, because we have not quite proceeded in that way. However, there is a certain similarity. We have in the ALCOR group devised and developed our translator plans completely machine-independently, and they have been implemented for machines of a rather different structure, such as the Z22 for example and the Siemens 2002, the Perm and the Ermeth. The decomposition - the first part of translation which would approximately, I think, coincide with the transformation from the original language into the intermediate language - is about the same for all machines. So we can, with the same input language, and the same transformation to intermediate language, work on different machines with rather different structures, and still be effective. That has been our aim from the beginning, always to try to devise the language in such a way that implementation is possible with good, efficient object programs. But still, I feel that the work which has to be done after decomposition for the specific machine, afterwards, is the main task. I do not know what the situation is in Jovial in this respect, and I would be interested to know. But this was the situation in the ALCOR group. I might add that the translators were one man jobs, and the construction for each translator so far took about one man year, with one exception: this was the decimal Siemens 2002, where we had two people working on the programming, and these people learned machine code programming while building the translator; they had never touched the computer before.

HUSKEY (U.S.)—With respect to the same point, I think I can speak with some experience. The system we have been using has been implemented in both binary and decimal computers. In fact, the decimal computer was one that handled ten word blocks of information from a main memory to a fast memory. Not exactly the same system, but a very similar one has been considered on the Bendix G 15. And these machines are radically different in structure from the single-address computers we have been talking about, and there are no particular difficulties due to the structure of the computer in this process.

VAN WIJNGAARDEN (Netherlands)—I would just like to make a few simple remarks with regard to the opening speech of the

Moderator. He said that one of the reactions of the compiler maker was to restrict the use of the language, and I think he said it to be a very advisable method. I want to stop here and say that every compiler maker is restricting the use of the language, unless the language is not suited specifically for the machine for which he is implementing it. If you take machine-oriented languages, let us say like Fortran, then we have versions referring to a specific machine: for instance, Fortran followed by some identifier, which specifies that particular machine. Then, apart from whether or not the definition may be exact, anyhow the definition of that language by means of its processor is perfectly unique. But if you take a language which is defined in some other way - let us say, for instance, a language like Algol - which is not oriented to a specific machine, then there are a lot of elements in it which cannot possibly be implemented in this physical world. Nevertheless, I do not think anybody would object to a language like Algol for these reasons. Well, if you take this point of view, nobody in the world can implement Algol 60, of course. Because you should realize you can easily write down a program, just adding 1 to i consecutively, which will run over the capacity of all existing computers in the world. The numbers grow so big that you cannot keep them in, and therefore each implementing system will in some way or another restrict the language, let us say, in the use of the integers. Well, these kinds of restrictions are obvious. Everybody knows them. Then there are, of course, questions of identifiers. No machine can ever cope with an identifier of more letters than it could put into a store, an extended store. Therefore identifiers should be limited, let us say, to nine significant letters. But, of course, that again is an undue restriction, because you can write down a program in which there are so many different identifiers, that you cannot distinguish them with this already high amount of letters. Here, of course, again, you have the same difficulty that you restrict the use of your language some way or another. Now, the same things hold on much higher levels. At the Mathematical Center, a couple of years ago, Dijkstra and some of us were making a translator for Algol 60. This was, strictly speaking, for a subset of Algol 60; it had the structure of the type I mentioned; it had other restrictions, actually on the use of own arrays with dynamically varying bounds and own variables in recursive situations. The restrictions were largely the same as the ones that were dis-

cussed by van der Poel two days ago. One might say, in this situation, that the ideal of the implementor is: please, let us change the language so as to fit my requirements, because I find it a bit hard to implement. But in principle there is no difference between this kind of difficulty and the kind of difficulties I mentioned beforehand. In the meantime, just to show that it is possible, some other people at the Mathematical Center make another compiler for Algol 60 which overcomes these little difficulties, in which arrays with dynamically varying bounds are allowed, and also own variables in the recursive situations are allowed. In this last case, however, again you have to introduce a slight concession. This is the point I want to make: I do not think the implementor should bother very much to try to influence language, but he should be willing to state clearly that he is implementing a specific subset of the language or a part of the language under conditions which he can clearly specify. As long as he is able to specify exactly his restrictions, I cannot see that there are great difficulties.

ZEMANEK (Austria)—Thank you, Prof. Wijngaarden, for generalizing my point.

WEIZENBAUM (U.S.)—I think the compiler writer has three principal responsibilities. One, given a machine, and given a language he has to do the best job he possibly can; two, he is to give advice to the language designers coming from his experience as to how to design languages better; three, he is to give advice to machine designers coming from his experience as to how to design machines better.

WEGNER (U.K.)—I should like to make some general comments regarding the implementation of processors. First of all, the question of practicability has been running throughout this discussion: I feel that too much stress is placed on what is practicable, and too little on what is desirable. For instance, I am not too happy when people stand up and say that we have to consider small computers and the practical matters associated with them. This conference is presumably intended to discuss methods or matters of general applicability with a view to the future, rather than special-purpose matters coming from the exigencies of the moment. In this line there seems to be little difference, for instance, other than efficiency, between a language and a machine. Dijkstra pointed out that languages and machines are really

equivalent and it is only the efficiency characterization that makes a difference here, so that when we consider whether we want the machine to have certain characteristics, we can be as ambitious as we like, because we can always interpret the machine of our desire on some other machine. So here I think it is very useful discussing the characteristics of an ideal machine, and the argument that we are not being practicable, I do not think is important. In order to make processor writing easy, it seems to me that the key matter in considering processor implementation is the structure of the machine-oriented intermediate language into which our source language is translated, whether this is a language associated with a given machine, or some ideal intermediate language in which we are interpreting on our machine. It seems to me that this is the problem that should be attacked, and on which we should concentrate. Although I feel that I am not in a very strong position - I have not recently implemented any compilers - I feel that, as a compiler implementor, I would want to stress the intermediate languages into which source languages are translated.

And then I would like to say one or two things which can be said about such languages: the kind of approach that I might like to take in considering intermediate languages. For instance, to what extent are intermediate languages machine-independent. In order to develop machine-independent languages it seems to be necessary to consider invariant characteristics associated with languages, and in this connection I think the approach of Saul Gorn is useful, in that complete pre-fixed languages characterize important invariant characteristics of the language, and for this reason would have a part to play in the characterization of intermediate languages, into which problem-oriented languages are translated. I feel therefore that an intermediate language should have a pre-fixed structure, anyway when it is describing Algol. From the machine point of view I think we can say something about intermediate languages along the lines that Dijkstra has developed for his stack operations in connection with his Algol interpreter. Another direction is the Fortran BSS loader: this is an intermediate language, and there are general considerations of flow between the various components of a language which should be reflected in an intermediate language. Also input-output and supervisory control languages should be characterized in terms of intermediate languages. Well, just to repeat, then: it seems to me that one of the most fruitful directions

in which processor implementors can work is in the direction of characterizing intermediate languages, and I think it would be very worthwhile, for instance, to collect intermediate languages associated with processors, and I would be very interested in collecting intermediate languages and their characteristics, if anybody should want to communicate with me in this connection.

SWINNERTON-DYER (U.K.)—It seems to me important that we should distinguish between a language used for communication between man and man and a language used for communication between man and machine. If you want to communicate between man and man, you are limited essentially only by what the person can understand. You assume that the man who is reading it is a man of very considerable ability and you can afford to have a language of very great complication. If you are communicating with a machine, at least those of us in Europe who have small, slow machines, then you do not have a machine of very great intelligence and capability, and you have to confine yourself to a language suitably simple for that machine, partly because you simply cannot compile a complicated language on a small machine, partly because if you try, you find yourself using a very inefficient compiler and taking interminable amounts of time, and in addition having to cut all sorts of corners in writing a compiler which simply does not work. And if you have a compiler only to compile your own programs, then it does not matter very much if you find two or three mistakes a week. But if you have problems for engineers and such like, and they get the wrong answers, it is no use emerging with a gay laugh, saying "ah, that's very interesting: not a mistake today!". In consequence, you have got to have a very simple language. It is clear that the policy, by and large, has been that you start off with a language such as Algol which, as it turns out, was really designed for communication between man and man, and not between man and machine. You decide that you cannot, with very rare exceptions, implement the whole of Algol, and you cut yourself down to the sort of parts of Algol which you can implement. Now this is very untidy, because it leaves you a language with not very great facilities in relation to the complexity of its structure. And it is on the whole a mistake to have a very complicated language which can do only very simple things, which is what you are going to get if, for example, you compile with Algol without proportions as a popular sport. Consequently, in addition to

having a very good language for communicating with other people, you want a language for communicating with the machine, adapted to the sort of machine you have. And you want the series of languages from the very small simple language right up to things like Jovial, adapted for the computers which we wish we could afford to have. The snag possibly in this is that somebody else sent you a program to be run; it is now written in Algol and you have to translate it into something simpler. But I do not believe, and I do not have much experience of this, that it is really going to be very difficult to translate a practical program in a complicated language into a simpler symbolic language.

My second point is: unless you have a machine so powerful that efficiency does not matter, it is highly desirable that your language should be so devised as to induce the people who use it to do simple things rather than complicated things. The outstanding example of this is quite evidently recursion. Algol, in particular, is so designed, that it is actually easier to write a recursion than a simple count, in spite of the fact that it is a great deal harder to implement on the computer and a great deal more time-consuming when you do implement it. An outstanding example: the natural way for a real Algol fanatic to calculate n! is by saying essentially that if n is equal to 1, then factorial n is 1; if n is greater than 1, then factorial n is n times (n − 1)!. And you really do this, because it is really easier than setting up the counting structure. Well, now, this must be fundamentally wrong in any language which is designed to be implemented on a machine.

DREYFUS (France)—In quality of Chairman, I do not want to get involved in this discussion, and this is merely a suggestion. From what I have heard in this panel, many different thoughts have been expressed and, by and large, one sees that they all come from the different weight given to the essential criteria which one wishes to see implemented in a compiler. I would therefore like to make a suggestion: could not a little committee draft a set of ten criteria, have them circulated, and people here present, who supposedly represent the experts in the field, would just put their own writings on these ten criteria, and we would then have some correlation and statistics. I think it might be very useful.

MILEWSKI (U.K.)—Regarding the suggestion made by the Chair-

man, I think that such a statistical survey by the people present here on the relative importance of various aspects of a language, such as efficiency of object programs, efficiency of compilation and so on, would be of limited use. The reason for this is, I think, in the fact mentioned before, that efficiency of a compiler can only be measured in terms of a specific situation. I think such a survey would only be useful if the questions were further subdivided into different computers, different types of applications, etc. But I think that if this were done, the whole survey would become so unwieldy as to again be of limited value.

PAUL (Germany)—A rather technical point which I should like to be considered is the following. Computers up to now are not very convenient for handling the translation of formal languages into each other. One thing, for instance, that is very often used in translation of formal languages is the distinction between elements of different classes. For example, in Algol you have to distinguish between reals, integers and Booleans, but during translation there is much more distinction needed, for instance slow storage, fast storage, and in future there will be much more. Of course, one can make such distinctions between elements of different classes in computers today, but, at least in machines which have built-in floating-point arithmetic, it consequently leads to simulating, which would not be realistic at all. Therefore, in designing computers, one should consider this point of view which, in my opinion, seems very important.

SAMELSON (Germany)—I only wanted to take up the point that Paul made. I would say that in theory I agree with his idea very much but, looking at the facts of like, I very much disagree. In fact, I see that at the present stage of the development of formal languages I would be very cautious in trying to advise machine manufacturers to make major changes in the design of the machinery. I was materialistic enough to mention money the day before yesterday, and I feel bound to do it today again, because what Paul mentioned costs a lot of money, especially since storage is rather expensive. So I would say our present task is to work with the existing machinery. We will have to work with that for quite a number of years yet, I am sure, and to do just the best that is possible, which means that we have to refine our languages so that they fit the machines.

DIJKSTRA (Netherlands)—I strongly oppose the view proposed by Samelson, because, as a programmer, I feel guilty with respect

to computer manufacturers. Why are present-day machines so extremely boring and dull? The reason for this is that programmers fail to do what they have to do. They fail to tell the engineers what they really want. Therefore, I consider it my duty to pay a great amount of attention to the feedback manufacturers need for the benefit of us all.

HUSKEY (U.S.)—If I may take a position. I have been on the engineering side of this, and I just am forced to make one comment. It seems that the programmers have difficulty telling one another what they want, let alone the engineers.

NAUR (Denmark)—I just have a short remark on the question of design of machines. As far as I can see, machines in the past have primarily been designed with specific problems in mind. As time went on, people found that they had to do a lot of counting when they did matrix problems, these matrix problems having a certain predominance in some applications, and people introduced simplifications for doing this kind of counting. Also floating point arithmetic is the same kind of thing. This is connected with certain specific problems or classes of problems. It seems very logical to me that we should try to get to a high level here, and the language, to me, would be the higher level, the level which would be meta-level in respective problems. It would really, at least, hopefully, be designed to express conveniently not only specific problems, but a whole class of problems, so that this movement in machine design of easing specific problems would be moving as if you eased in the machine design the problems of the language.

SCHWARTZ (U.S.)—I think although we have advertised there are big compilers and powerful languages, you will find we agree with many of the arguments for having subsets. One of the remarks about Jovial which has been made is this, that for many problems which really do not require all the abilities of the language it means using a powerful tool to crack a nut. So we are more or less of the opinion today, if we had the time and the money, that what we would like to do with what we have today is actually make seven different languages, each of increasing complexity, but all subsets of the next one. In other words, the seventh level would be what we have today, but for those people who do not need all this we could supply a compiler that operates in a minute for 10,000 instructions. Now, a professional program-

mer can, if he wishes, make full use of the computer with Jovial and still be programming in a fairly abstract fashion, fairly machine-independent fashion. However, if a programmer does not want to use all this power - and we have non-professional programmers using Jovial fairly frequently - we believe rather strongly in his ability to take subsets of what we have.

WEIZENBAUM (U.S.)--I think that if machine designers had a discussion like this, and 20% of the discussion had to do with how to simplify the task, perhaps at the expense of making the machine less efficient, then everyone would be appalled. One of the things a compiler writer does not have to do is to make life easy for himself. His job is to work very hard, because his product is going to be used by many people. Now, with respect to the assertion that programmers do not know themselves what they want, I think the situation is, at least in the U.S.A., that programmers are very busy writing problems, writing codes for problems that have to be done, and so on and so forth. There is a class of programmers, and this is a class of language specialists, which is represented here, whose job it is to know what they want and not be constrained by existing machines. I think this is extremely important.

WEGNER (U.K.)—In connection with Weizenbaum's statement, I cannot resist making a remark. It is a well-known fact that programmers in the U.S.A. do make life hard for themselves in writing compilers. In Europe there is not quite as much money available for teams to work on compilers, and people in Europe tend to write less ambitious compilers and more straightforwardly. However, I think that if one has to get down to the task of simplifying the task of compiler writing in the end this does not lead to a less efficient compiler. It leads to a greater understanding of the compiler process. And in general, I feel, when one studies the esthetics of the program and when one achieves an esthetically nicer program, this usually in the end results in something more efficient rather than less efficient.

VAN DER POEL (Netherlands)—I would like to make a proposal along the same lines as has been heard already, but perhaps I will phrase it a little bit differently. I think what is needed is an extremely powerful and generalized language, but stripped down to the utmost, stripped down to the possibilities that it can, in itself, by a sort of procedure declaration, declare the rest of

the mechanism needed. For example, a "for" statement is not necessary at all in Algol, you can express it by procedure declaration in other more basic properties, and I think what we have to do is to find a generalized language which enables you, for any machine, to elaborate that language in that machine itself. Of course, it is very difficult to find the core which is necessary, but it is certain to me that this core should be a very efficient generalized language, let us say, along the lines of the Dijkstra language or another generalized Algol with dynamic declarations, or something of the like.

SAMELSON (Germany)—I would like to take up the point of van der Poel. First I would say that in principle I agree with him: we are thinking along the same lines, but I do not think that we should talk about it too much before we can really show results. I feel that the way of devising languages which enabled a programmer to define his own programming languages, will remain academic for quite some time. First I think it will require quite large machines; secondly, with problem oriented languages, I feel that we are concerned mostly with the non-professional users, such as the engineers and the physicists who, as I at least believe although I cannot state it as a fact, are probably more interested in having a language to express what they want to do, than in having a language which allows them to define how to express what they want to do, because this is again removed by one stage from what they want to do and they probably will not want to be bothered too much with it.

VAN DER POEL (Netherlands)—I should like to make a counter-remark to this. I think the non-professional users should not give the main line in influencing a language. It hampers theory very much. Well, I shall put it another way. I think there is a very good use in letting theorists take care of theory, and then a practical use will always appear. We have seen this in mathematics many times and I think it is wrong to let the user influence too much the structure of the language.

ZEMANEK (Austria)--I want to thank the panelists and the audience for their attention.

DESIGN OF SPECIAL LANGUAGES

QUELQUES OPERATIONS AUTOMATIQUES FONDEES SUR LA GRAMMAIRE DU SYNTOL EN DOCUMENTATION AUTOMATIQUES

J. C. Gardin et *F. Levy*

Centre National de la Recherche Scientifique,
Ecole Pratique des Hautes Etudes,
Paris, France.

I. INTRODUCTION

Le thème principal du Symposium est l'étude des langages de programmation; ce n'est cependant pas un langage de programmation qui nous occupera, mais un langage plus synthétique, que nous utilisons aujourd'hui dans divers travaux de documentation automatique, pour formuler les opérations demandées à la machine, pour quelque domaine et quelque "question" que ce soit. Notre sujet est donc probablement "déplacé", au moins au sens propre du mot, puisqu'une fois posées ces opérations, le problème de leur programmation demeure entier. Mais avant d'envisager un langage symbolique adapté aux programmes de documentation automatique - le LANGUEDOC justement souhaité par M. TABORY - il nous a paru raisonnable de définir mieux qu'on ne l'a fait jusqu'ici les objectifs de ce langage, en termes d'*opérations*, fût-ce à un niveau de toute évidence trop synthétique pour fonder encore le projet d'un compilateur.

Ces réserves posées, précisons ce que nous entendons par "opérations", dans le domaine de la documentation automatique; et d'abord de quel aspect de la documentation automatique s'agit-il ici? Nous avons pris l'habitude d'en distinguer trois:

a. La transformation automatique d'un texte écrit dans une langue naturelle en une "représentation" jugée plus adéquate, pour la recherche documentaire. Titre adopté, par convention: "analyse automatique".

b. La détection automatique de certaines correspondances

entre les "documents" enregistrés et les "questions" posées, au niveau de ces représentations. Titre: "recherche automatique".

c. L'enrichissement automatique du réseau d'associations utilisé par la machine dans la phase précédente. Titre: "apprentissage automatique".

Le seul aspect de l'automatique documentaire dont nous traiterons ici est le second, à savoir la *recherche des données*. Les opérations impliquées dans cette recherche dépendent naturellement de la manière dont les données ont été préalablement "représentées", du point de vue linguistique. Il faut par conséquent que nous proposions tout d'abord une sorte de modèle général des "représentations" couramment adoptées dans la documentation automatique d'aujourd'hui.

II. MODELE DES "REPRESENTATIONS" COURANTES

2.1 "Mots" et "Relations"

C'est une banalité que d'opposer en documentation problèmes de "lexique" et problèmes de "syntaxe", expression des "mots" et expression des "relations", etc.; une banalité aussi, que de rappeler le caractère contingent de cette opposition, toute relation entre deux mots quelconques d'une langue pouvant toujours être exprimée par un troisième mot, plutôt que par une phrase (ex.: "insecticide", "pathogène", etc., et plus généralement toute correspondance entre un mot et les phrases qui le définissent).

Nous reprendrons cependant la même distinction, en la formulant de la manière suivante: toute analyse documentaire suppose l'emploi d'un "lexique", où l'on évite de créer un mot nouveau chaque fois qu'apparaît une nouvelle relation entre deux termes préexistants. Au "lexique" s'ajoute par conséquent une "syntaxe", qui peut revêtir les formes les plus diverses, depuis une sorte d'état minimal, où l'on se borne à considérer des combinaisons de mots, sans relations explicites, jusqu'à un état maximal, où toutes les relations utiles sont au contraire distinguées et exprimées entre les mots de l'analyse pris deux à deux. Pour les besoins de l'exposé, nous supposerons que l'on dispose d'un inventaire adéquat de ces relations, et que la procédure suivie pour les exprimer est celle des "chaînes"—ou "Links", en anglais—à savoir la construction de "syntagmes"

formés chacun de deux mots Mi et Mj, liés par une relation Rn, généralement orientée. On sait en effet que l'analyse documentaire est dans ce cas aussi précise que l'on veut, soit que l'on enrichisse la définition des mots et des relations employés, en "compréhension", soit que l'on allonge les chaînes de syntagmes, en "extension". Les opérations définies sur ces représentations maximales pour la recherche des informations, pourront donc être considérées elles-mêmes comme suffisantes, dans tout projet de documentation automatique; et l'on aboutit à un premier classement de ces opérations, selon qu'elles portent:

—sur les "mots" (substitutions de termes plus ou moins voisins, quant au sens, dans un contexte donné; passages du particulier au général, etc.)

—sur les "relations" (changements de configuration, dans l'enchaînement logique des mêmes termes; insertion ou supression de chaînons intermédiaires, entre deux termes donnés, etc.)

2.2 "Syntagmes" et "Paradigmes"

Une deuxième opposition sépare parmi ces relations, celles qui sont fournies par l'analyse des documents eux-mêmes, au fur et à mesure de leur mise en mémoire, et celles qu'il faut au contraire établir a priori, avant toute exploitation, parce qu'on les trouve rarement posées de manière explicite dans la littérature scientifique. A ces liaisons *implicites* entre ces concepts correspondent les classifications hiérarchiques préétablies, telles qu'on les utilise en documentation; c'est, si l'on veut, la dimension verticale, ou "paradigmatique" de l'analyse. Aux liaisons *explicits* données par les textes eux-mêmes correspondent inversement les enchaînements "syntagmatiques", selon la dimension horizontale de l'analyse.

La distinction est ici encore contingente; et l'apprentissage n'est d'ailleurs rien d'autre, en documentation, que la génération automatique de nouveaux "paradigmes", au fur et à mesure -et à partir- de l'accumulation des "syntagmes". Nous admettrons cependant la nécessité *pratique* d'enregistrer a priori un certain nombre de liaisons "paradigmatiques", entre les termes du lexique; et nous poserons par conséquent une seconde dichotomie, dans les opérations requises pour la recherche des données documentaires, selon que ces opérations portent:

—sur les liaisons "paradigmatiques", i.e. l'organisation "innée" de la mémoire

—sur les liaisons "syntagmatiques", i.e. l'organisation "acquise" de la mémoire.

III. LA REPRESENTATION SYNTOL

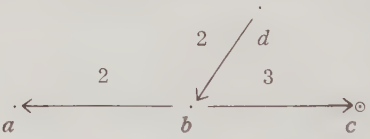
Le SYNTOL (Syntagmatic Organization Language) est un système documentaire parmi d'autres, conforme au modèle général que l'on vient d'esquisser.*

On y retrouve:

a. la distinction entre mots et relations, celles-ci étant enregistrées indépendemment de ceux-là.
b. la distinction entre relations syntagmatiques et relations paradigmatiques, qui constituent deux réseaux entièrement distincts.

Les relations paradigmatiques sont les rapports invariants entre les notions élémentaires du lexique; ce sont essentiellement des rapports d'inclusion, que l'on enregistre sous forme de chaînes ou d'arêtes disjointes, chaque mot du lexique pouvant appartenir a une ou plusieurs lignées ascendantes, selon différents types d'organisation hiérarchique utilisables conjointement.

Quant aux relations syntagmatiques, elles sont en SYNTOL au nombre de quatre, complétées ou non, du point de vue de la signification, par quelques opérateurs facultatifs, et exprimées en pratique par des groupes "Rn p/q", où n indique l'une des 4 relations logiques du SYNTOL, et p, q les numéros d'ordre des mots constitutant les pôles du syntagme correspondant - "orienté", de p vers q - dans la suite arbitraire de l'énumération des termes. Soit par exemple le schéma SYNTOL suivant:



les termes a, b, c, d seront enregistrés dans un ordre quelconque—par exemple:

* MOTS * a, b, d, c *

*Ce système à été construit et mis à l'épreuve sur ordinateur IBM 7090, grâce au concours financier de l'EURATOM. Rapport final à paraître en 1962.

les relations seront exprimés ensuite au niveau des numéros d'ordre de ces mots, dans la suite arbitraire précédente (a : 1 – b : 2 – d : 3 – c : 4), soit:

* RELATIONS * R2 2/1, 3/2 * R3 3/4 *

A ces données vient s'ajouter une spécification supplémentaire, dite "Thème", destinée à *centrer* l'analyse sur celui ou ceux de ses termes qui désignent le sujet principal de l'étude, son "foyer" - ici, par exemple: * THEME * c (ou 4) *

IV. LES OPERATIONS AUTOMATIQUES

Ces schèmes valent naturellement aussi bien pour l'expression des données documentaires que pour l'expression des "questions" dont celles-ci font l'objet. Le problème de la recherche automatique des informations se ramène par conséquent à la comparaison de schèmes de ce genre, pour déterminer les équivalences ou les voisinages cherchés, entre les "questions" posées et le contenu des "documents" enregistrés.

La coincidence n'est pas nécessairement totale, entre le schème de la question et tout ou partie d'un schème correspondant à l'analyse d'un document pertinent. Dans la plupart des cas, l'on admet au contraire un certain "jeu" de l'un à l'autre; et la recherche doit s'instituer le plus souvent à partir d'une question librement "modulée" par le demandeur, en fonction de ses critères propres. Par "modulations", nous entendons les transformations admises dans l'énoncé d'une question, pour élargir à volonté le faisceau de ses correspondances avec les données enregistrées.

Les règles mêmes de la représentation adoptée, ci-dessus, déterminent la nature des opérations concevables:

A. *Substitution de mots* à d'autres, en un sommet donné du graphe.

Ou distinguera trois cas, selon que les termes permutables appartiennent:

—à une liste fournie par le demandeur; cas de "*disjonction*" simple, notée comme suit en SYNTOL: "(a, b, c. . .)"

—à une liste fournie par l'organisation paradigmatique du lexique; cas de "Sommation" ou de "Généralisation", où l'on admet la substitution à un mot p de tout autre terme appartenant à une classe donnée, que cette classe soit désignée par le terme p lui-même - "*sommation*", notation,:

"(p)" - où qu'elle se trouve 1, 2...n niveaux au-dessus de p, dans l'organisation hiérarchique choisie - "*généralisation*", notation: "n G p".

—à l'ensemble de lexique; cas extrême de la généralisation—dit "indétermination"—équivalant à remplacer le mot p par une inconnue. Notation conventionnelle: "9 G p".

B. *Substitution de relations* à d'autres entre deux sommets donnés du graphe.

On distingera deux cas, selon que l'incertitude porte:

—sur la nature de la relation logique, entre deux mots contigus p et q d'une chaîne; cas d'"*altération*", notation "Rn p, q". Cette opération équivaut à la levée de toute restriction sur n, dans l'expression du syntagme "A p/q".

—sur le *nombre n* (et la nature) des syntagmes intermédiaires, entre deux mots p et q initialement posés comme contigus, dans l'énoncé de la question; cas de "*médiation*", notation "n M p/q".

Cette dernière transformation conduit à admettre 2, 3...n arêtes au lieu d'une seule, entre les termes p et q; elle est rendue nécessaire par le fait que pour des raisons pratiques, l'opération réciproque, dite de "développement" (p R m...m R q —> p r q) n'est pas toujours admise dans les analyses de documents, même lorsque les conditions sémantiques de la transitivité sont satisfaites.

Un cas extrême de médiation est la "*scission*", où l'on supprime toute restriction sur la proximité de p et q, dans les enchaînements, tout en maintenant l'exigence de leur présence conjointe dans la graphe; notation conventionnelle: "9 M p/q". La "scission" revient à couper localement les arêtes du graphe, pour ne garder que la représentation par sommets, ou mots-clé.

C. *Suppression de mots* et des relations qui leur sont attachées.

Cette opération rend *facultative* la présence de tel ou tel sommet p dans le graphe et par conséquent celle de toutes les arêtes passant par p; elle porte le nom de "*partition*", et se note "P p".

La partition ne doit naturellement pas être confondue avec la "*négation*", où l'on *exige* l'absence d'un ou plusieurs mots, dans les schèmes obtenus en réponse à la question.

D. *Substitution de thèmes* à d'autres, i.e. suppression de cette indication dans l'énoncé de la question.

L'on admet alors la possibilité d'obtenir des réponses "excen-

trées'', où le schème de la question posée est bien attesté, mais d'une façon en quelque sorte marginale, sans que les documents correspondants aient cette question pour ''Thème'' principal. Cette opération est dite ''défocalisation''; notation ''D''.

V. RESUME

Les ''modulations'' précédentes ont été incorporées dans un programme de documentation automatique écrit pour l'ordinateur IBM 7090. Les symboles correspondants forment autant de mots de contrôle, qui appellent chacun un sous-programme. L'usage des modulations est ainsi mis à la portée du demandeur lui-même, qui peut apprendre en quelques minutes les règles du jeu. L'expérience montre que ces opérations sont indispensables à l'efficacité des programmes de recherche, et nullement liées à tel ou tel champ d'application particulier. Reste à préciser les fonctions invariantes que renferment ces opérations, pour faire mieux apparaître l'opportunité de tel ou tel langage de programmation particulier.

LES LANGAGES DOCUMENTAIRES–MODELE DESCRIPTIF ET PROBLEMES FONDAMENTAUX*

Mlle. Fouquet, Mm. Bertier, Céron, Darnaut, Félix, Lattès, Le Boulanger, Roy, Sandier

Sema, Paris, France

INTRODUCTION

Il semble que la compréhension d'un langage documentaire, ainsi que la connaissance de ses limites et possibilités, puisse résulter de l'examen des six points suivants:

(a) *les mots* ou unités linguistiques du langage documentaire utilisées pour caractériser les concepts;

(b) *les rapports paradigmatiques* ou rapports essentiels et universellement acceptés—par conséquent implicites—qui existent entre les mots;

(c) *les rapports syntagmatiques* ou rapports contingents et occasionnels—par conséquent explicites—qui sont susceptibles d'exister entre les mots;

(d) *la représentation formelle des documents*

(e) *la représentation formelle des questions*

(f) *la règle de pertinence* ou règle qui sert à décréter si un document donné répond à une question posée.

Tout en précisant chacun de ces six points de vue, nous essaierons de montrer sur plusieurs exemples que leur adoption successive constitue un canevas fécond et commode pour la description de n'importe quel langage documentaire.

Nous montrerons ensuite que le modèle descriptif qui vient d'être présenté permet:

(1) de comparer entre eux les langages documentaires et de déceler certaines de leurs caractéristiques essentielles: celles-ci reposent principalement sur le dosage, les modalités de

* Cette communication est faite dans le cadre de recherches effectuees pour EURATOM sous la direction du groupe CETIS.

prise en compte, et l'utilisation des rapports paradigmatiques et syntagmatiques;

(2) d'énoncer et même d'aborder quelques uns des problèmes fondamentaux que l'on doit se poser au moment de l'élaboration d'un langage documentaire: sans que cela constitue la solution de ces problèmes, nous indiquerons les facteurs extérieurs—liés à la population documentaire, et aux utilisateurs—sur lesquels doit reposer cette solution.

Nous nous attacherons tout particulièrement aux problèmes que pose la richesse des rapports paradigmatiques et syntagmatiques. Nous voudrions montrer que cette richesse d'expression du langage ne doit pas être fondée sur un souci de traduction fidèle, mais au contraire sur les désirs de la clientèle quant à l'équilibre que doit assurer le système entre documents parasites et documents omis. La richesse respective des deux types de rapports qui permettra d'assurer cet équilibre, dépend avant tout de la population documentaire.

En conclusion nous insisterons sur le fait qu'un langage documentaire élaboré en vue de permettre une traduction fidèle de la pensée, risque d'être inapte aux fins qu'on lui propose. Celles-ci confèrent en effet au langage documentaire des exigences particulières qui ne sont pas celles d'une bonne traduction.

I. MODELE DESCRIPTIF

(a) Les mots

Par définition, nous appellerons *mot* l'unité linguistique du langage documentaire utilisée pour caractériser un concept.

L'ensemble de ces mots sera désigné par M. L'examen de plusieurs langages documentaires nous a montré qu'un facteur important de différenciation provenait de ce que cet ensemble pouvait être *énuméré* ou *engendré*:

1. L'ensemble M est défini par énumération. Il s'agit du cas où les mots sont donnés par une liste exhaustive, celle-ci pouvant éventuellement être mise à jour périodiquement. Les langages SAINT-GOBAIN et SYNTOL sont de ce type.

2. L'ensemble M est défini par génération. Il s'agit du cas où les mots sont construits à partir "d'éléments" que nous appellerons ici des "générateurs", ceux-ci étant donnés a priori. Les langages RULY-ENGLISH et W.R.U. sont de ce type. Dans ces langages, les

mots sont définis par une suite, généralement ordonnée, de certains des générateurs. Nous désignerons dans ce qui suit par G l'ensemble des générateurs. Ceux-ci portent des noms variés: racines et modulants dans RULY-ENGLISH; facteurs sémantiques, infixes et suffixes, dans W.R.U.

(b) Les rapports paradigmatiques

Par définition, nous rangerons sous le nom de *rapports paradigmatiques* les rapports essentiels universellement acceptés—et par conséquent implicites—qui existent entre les mots. Ex.: Paris est une ville de France.

L'étude de ces rapports nous a conduit à les classer en deux catégories: rapports hiérarchiques et rapports latéraux.

1. Rapports hiérarchiques. Ce sont ceux qui introduisent une relation anti-symétrique, telle que celle qui existe entre contenant et contenu, notion générale et notion spécifique. Remarquons que rien n'oblige à considérer cette relation comme transitive. Ces rapports, lorsqu'ils sont binaires, définissent sur M un graphe généralement sans circuit. Dans le langage SYNTOL, ce graphe est une arborescence, dont on trouvera un extrait dans la Fig. 1. Le langage RULY-ENGLISH souhaite, au contraire, prendre en compte un graphe sans circuit du type le plus général.

2. Rapports latéraux. Ce sont ceux qui introduisent une relation symétrique, telle que celle marquant un voisinage entre plusiers éléments, ou leur appartenance à une même classe. Ici encore, rien n'oblige à considérer cette relation comme transitive.

Ces rapports définissent sur M une famille de parties, chacune de ces parties étant constituée par les mots liés par un même rapport latéral. Ex.: Paris, Lyon, Milan, etc. ..., appartiennent à une même classe, qui est celle des villes.

Cette famille de parties peut, en fait, être traduite en conservant à M la structure de graphe que lui confèrent les rapports hiérarchiques (bien que les rapports latéraux soient de nature différente). On peut admettre, en effet, que chacune de ces classes est désignée par un mot de M, l'appartenance d'un mot m quelconque à une classe $\overline{m}$ peut alors être exprimée par un rapport hiérarchique entre $\overline{m}$ et m. Pour distinguer les rapports latéraux des rapports hiérarchiques, il suffit de savoir distinguer, dans l'ensemble M, les mots tels que $\overline{m}$ caractérisant les classes.

Il semble donc que, malgré leur double nature, les rapports

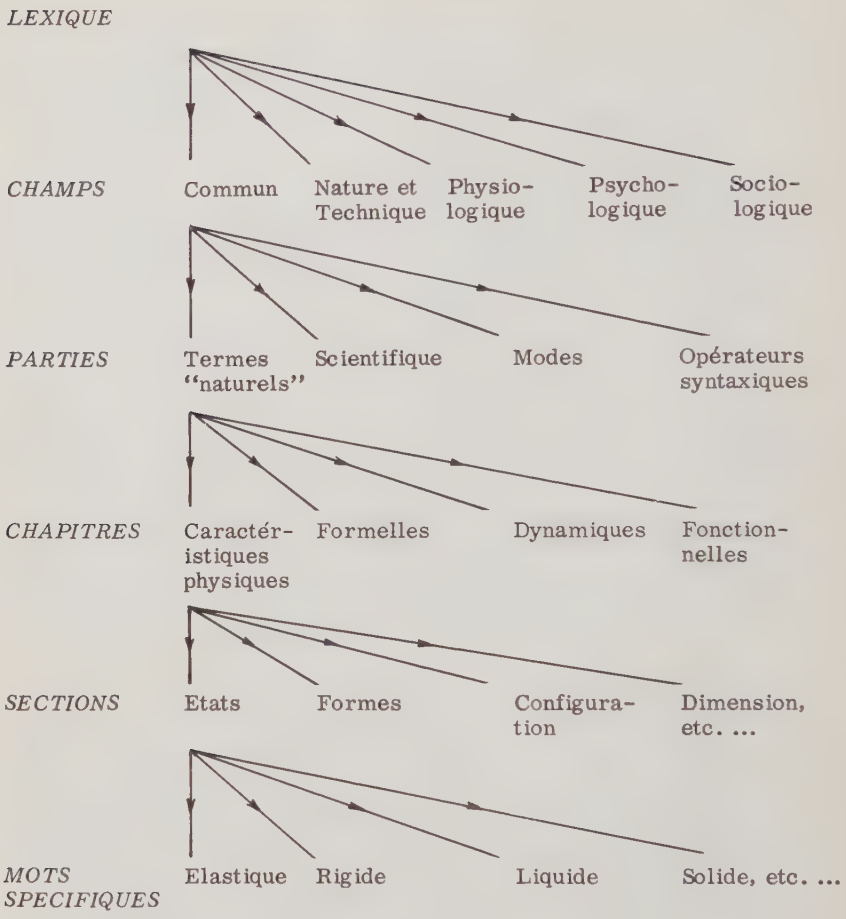


FIG. 1.

paradigmatiques puissent être toujours exprimés à l'aide d'un graphe défini sur M. Dans tout ce qui suit, nous désignerons par P ce graphe paradigmatique. La Fig. 2 en donne l'exemple.

(c) Les rapports syntagmatiques

Par définition, nous rangerons sous le nom de *rapports syntagmatiques* les rapports occasionnels et contingents—par conséquent explicites—qui sont susceptibles d'exister entre les mots. Ex.: dans le titre du livre de J. F. Gravier "Paris et le

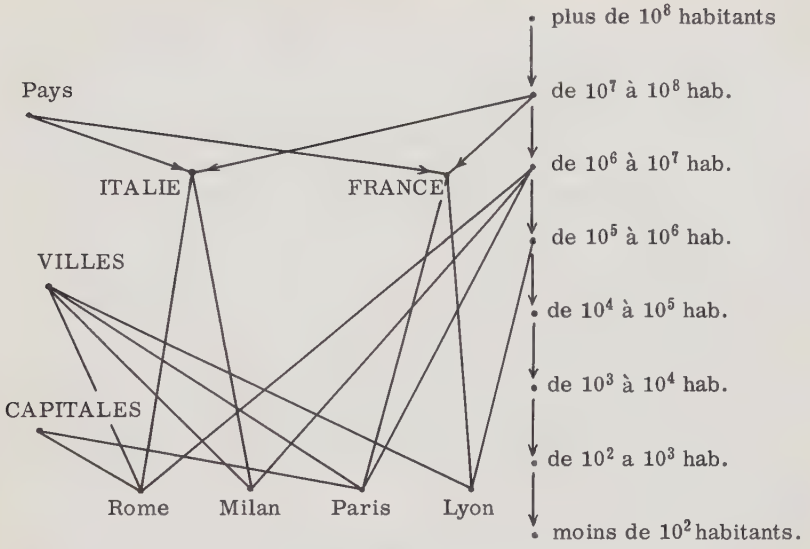


FIG. 2

désert français'', on peut déceler deux rapports syntagmatiques. L'un provient de la juxtaposition des mots ''désert'' et ''français'', l'autre est exprimé par la conjonction ''et''. L'étude de ces rapports nous a conduit à les classer en deux catégories: rapports spontanés et rapports expliqués.

1. **Rapports spontanés.** Ce sont ceux qui naissent de l'ordre ou du regroupement des mots. Ex.: grand homme et homme grand; désert francais dans le titre du livre de J. F. Gravier.

2. **Rapports expliqués.** Ce sont ceux dont la nature est précisée par une relation établie entre des mots ou des groupements de mots. Ex.: ''vers'', dans ''exportation de la France vers l'Italie''; ''et'', dans ''Paris et le désert francais''.

Les raisons profondes qui rendent nécessaire la distinction (en documentation automatique) entre rapports paradigmatiques et rapports syntagmatiques sont exposées en *annexe I*.

(d) La représentation formelle des documents

C'est la manière dont le langage documentaire utilise ses mots, sa paradigmatie, sa syntagmatie, pour représenter de façon abstraite la référence et l'analyse de chaque document. Il s'agit,

autrement dit, de l'intégration des éléments étudiés dans les paragraphes précédents.

Si le langage consiste simplement en une classification (cas du Centre de Documentation Sidérurgique, C.D.S.), cette représentation se réduit à un (ou plusieurs) mot caractérisant la (ou les) classe dans laquelle on range le document. Dans un langage à mots clefs (tel que celui employé par SAINT-GOBAIN), cette représentation n'est autre qu'un ensemble de mots dont l'ordre ne joue aucun rôle.

Dans les langages plus riches, cet ensemble de mots est muni d'une structure plus ou moins complexe, qui permet de représenter de façon plus précise le contenu des documents. C'est ainsi que dans RULY-ENGLISH il existe plusieurs familles de parties définies par la présence des interfixes, le regroupement en items et compositions; dans SYNTOL ou dans GRISA, ce sont des relations binaires de plusieurs types qui confèrent à l'ensemble cette structure.

(e) La représentation formelle des questions

Il s'agit du même problème qu'au paragraphe précédent, mais relatif cette fois aux questions et non plus aux documents. Il est traité de la même manière dans la plupart des langages, tout au plus certains d'entre eux autorisent-ils l'emploi d'opérateurs logiques dans la représentation des questions, alors que cela ne se fait pas pour les documents (cf. W.R.U.).

(f) La règle de pertinence

La règle de pertinence est, par définition, *l'ensemble des conditions* (nécessaires et suffisantes) que doivent vérifier le couple constitué par:

—la représentation dans le langage formel d'un document d (analyse et référence).

—la représentation dans le langage formel d'une question q.

pour que le document d soit *décrété pertinent*

par le système documentaire en réponse à la question q.

Nous symboliserons cette règle par R et traduirons par:

d R q = 1 le fait que d est décrété pertinent pour q,

d R q = 0 le fait que d n'est pas décrété pertinent pour q.

La définition de cette règle est d'autant plus délicate que la représentation formelle des documents et des questions est plus structurée (cf. SAINT-GOBAIN, SYNTOL, W.R.U.).

L'analyse approfondie de chacun de ces six points a été entreprise pour plusieurs langages documentaires.* Cette analyse nous autorise à penser que le plan d'étude qui précède peut conduire à une description complète de n'importe quel langage, cette description se prêtant de plus à des comparaisons fécondes entre langages.

L'application de ce modèle descriptif nous a, en outre, conduit à penser que les options essentielles qu'il s'agit de prendre au moment où l'on conçoit un langage documentaire, se concentrent autour de quatre poles d'attraction ou *caractéristiques* principales.

II. CARACTERISTIQUES PRINCIPALES

Les quatres caractéristiques retenues concernent:
—le dosage des rapports paradigmatiques et syntagmatiques,
—le mode de prise en compte des rapports paradigmatiques,
—le mode de prise en compte des rapports syntagmatiques,
—les fondements de la règle de pertinence.

Dans cette section, nous allons définir, non seulement l'objet précis de chacune de ces caractéristiques, mais aussi l'ensemble des "états" qui leurs sont associés. Ces états sont à la caractéristique ce que les nombres sont à la variable numérique, ils correspondent autrement dit aux grandes lignes des différents types d'options que l'on peut concevoir et distinguer à priori. Toutefois c'est seulement dans la section suivante que nous aborderons les problèmes fondamentaux que posent le choix de ces états.

(a) Le dosage des rapports paradigmatiques et syntagmatiques

L'objet de cette caractéristique est de fixer l'importance du rôle dévolu dans le langage documentaire à chacun des deux types de rapports paradigmatiques et syntagmatiques. C'est essentiellement le problème de leur richesse respective qu'il s'agit de résoudre en optant pour l'un des états possibles.

On pourrait être tenté de croire qu'il y a là deux problèmes distincts, donc deux caractéristiques. En fait, ces deux problèmes ne peuvent être dissociés car leur solution consiste en un *dosage* des deux types de rapports. Ce point de vue sera justifié à la sec-

*Cf. Rapport n° 1 remis au CETIS.

tion III. Bornons-nous à souligner ici que ce dosage doit aboutir à un équilibre entre les deux critères antagonistes que constituent le nombre de documents parasites et le nombre de documents omis. Cet équilibre est défini par les besoins des utilisateurs, alors que le dosage qui permet de l'atteindre dépend avant tout du rôle que jouent l'un et l'autre de ces types de rapports dans la population documentaire elle-même.

Toutefois leur dosage ne s'impose pas pour autant de façon simple et inéluctable. En effet, la richesse des rapports paradigmatiques et syntagmatiques ne doit pas être guidée par un souci de traduction fidèle des analyses des documents, mais par celui de fournir les moyens d'agréger et de séparer ce qui a besoin de pouvoir être agrégé et séparé. (Voir Annexe I.)

Le dosage, dont il vient d'être question, étant par essence assez vague, il paraît bien hasardeux de vouloir préciser les états de cette caractéristique. On peut, de façon très grossière et subjective, schématiser un ensemble d'états possibles en distinguant, pour chacun des deux types de rapports, trois degrés de richesse:

—l'absence: aucun rapport du type considéré n'est explicitement pris en compte dans le langage, ce qui n'interdit pas une prise en compte implicite (grâce à des artifices de vocabulaire par exemple) de certains rapports très spécifiques du domaine étudié:

—la présence faible: quelques rapports assez vagues, ou au contraire nettement définis, mais n'offrant que des possibilités très limitées, sont explicités;

—la présence forte: une gamme très riche de rapports, permettant beaucoup de finesse, est explicitée.

Il en résulte neuf états possibles résumés dans le Tableau I ci-après. Nous avons en outre indiqué dans ce tableau l'état qui nous semble avoir été souhaité à l'origine par les principaux langages que nous avons étudiés.

Il convient de remarquer a ce sujet que:

—GRISA, s'il a été conçu pour être dans l'état 7, semble maintenant vouloir adopter l'état 8, ou peut être même 9.

—W.R.U., s'il offre une grande richesse d'expression paradigmatique, ne s'est guère préoccupé de l'utiliser en tant que telle (ses générateurs servent plus au dosage qu'à l'exploration systématique de voisinages); c'est pourquoi nous pensons que l'état visé était l'état 8.

—RULY ENGLISH, s'il a visé l'état 9, ne semble pas avoir encore surmonté les difficultés permettant de l'atteindre.

TABLEAU I
ENSEMBLE DES ETATS POSSIBLES

Rapport syntagm. / Rapport paradigm.	Absence	Présence faible	Présence forte
Absence	1 Saint Gobain	4	7 Grisa
Présence Faible	2 C.D.S.	5 Syntol	8 W.R.U.
Présence forte	3	6	9 Ruly English

Ces quelques considérations montrent l'importance de cette caractéristique et l'aspect primordial de l'option prise à ce niveau de la conception. C'est d'elle en particulier que dépendent les options relatives au mode de prise en compte, par le langage, de chacun des deux types de rapports (qui font l'objet des deux caractéristiques suivantes).

(b) Le mode de prise en compte des rapports paradigmatiques

L'objet de cette seconde caractéristique est d'indiquer *comment* ces rapports peuvent être pris en compte dans le langage documentaire. On distingue pour cela quatre états possibles dont voici la description.

1. Etat α: Les rapports paradigmatiques ne sont pas explicités par la représentation formelle des mots.* Cela signifie que:

—l'ensemble M des mots est énuméré, et ses éléments sont codés indépendamment des rapports paradigmatiques éventuels qui les concernent; (voir section I, paragraphe a.1)

—les rapports paradigmatiques sont consignés dans une liste

*Rappelons que nous designons sous ce terme les unites linguistiques du langage documentaire utilisees pour representer les concepts.

énumérant les arcs du graphe paradigmatique P ou exprimés grâce aux rapports syntagmatiques.

Notons que cet état est compatible avec l'absence complète de rapport paradigmatiques; de plus, il est le seul à présenter cette particularité.

Cet état a été adopté par les langages SAINT-GOBAIN et GRISA.

2. Etat β: Les rapports paradigmatiques sont complètement explicités par le codage des mots. Cela signifie que:

—l'ensemble M est énuméré;

—les rapports paradigmatiques sont pris en compte par un codage convenable de M, c'est-à-dire en appliquant P sur une autre *structure conçue a priori à partir d'éléments n'ayant aucun contenu sémantique.*

On peut noter dès maintenant que, si cet état est assez favorable à une exploration systématique du voisinage des mots, il se prête mal à l'assimilation des mots nouveaux et conduit à des codages qui risquent, dans certains cas, d'être très encombrants.

Cet état à été adopté par les langages C.D.S. et SYNTOL.

3. Etat γ: Les rapports paradigmatiques sont complètement explicités par l'expression des mots en fonction de générateurs. Cela signifie que:

—l'ensemble M est engendré; (voir section I, paragraphe a.2)

—on construit les mots à partir des générateurs, chaque mot étant représenté par une suite d'éléments de G (l'ordre jouant un rôle éventuellement).

—les rapports paradigmatiques sont totalement explicités par la présence (et éventuellement l'ordre) des générateurs qui servent à représenter les mots.

On notera que ce cas ne peut être confondu avec le précédent, contrairement à ce que l'on pourrait être tenté de croire. En effet, si les générateurs jouent un rôle semblable, par certains aspects, à celui dévolu aux éléments qui servent à fabriquer les codes dans l'état β, il est en général impossible d'attribuer à ces éléments un contenu sémantique comparable à celui des générateurs. De plus, l'état γ confère à M une structure bien particulière qui est celle des parties de G, structure que l'on n'a aucune raison de retrouver lorsque M et P sont définis par énumération.

Cet état a été adopté par le langage W.R.U.

4. Etat δ: Les rapports paradigmatiques ne sont que partielle-

ment explicités par la représentation formelle des mots. On peut distinguer deux possibilités:

—δ': une partie des rapports paradigmatiques est explicitée par le codage des mots, le reste l'est indépendamment de leur formalisation (compromis entre α et β);

—δ'': une partie des rapports paradigmatiques est explicitée par l'expression des mots en fonction de générateurs, le reste l'est indépendamment de cette formalisation et peut concerner, soit les seuls générateurs, soit au contraire les mots eux-mêmes.

On notera que cet état conduit à des systèmes fort complexes et qu'aucun langage, à notre connaissance, n'a retenu la solution δ'. Le langage Ruly English semble s'orienter vers la solution δ'' mais on ignore si les difficultés qu'elle présente ont pu être vaincues.

(c) Le mode de prise en compte des rapports syntagmatiques

L'objet de cette troisième caractéristique est d'indiquer *comment* ces rapports peuvent être pris en compte dans le langage documentaire. Voici une synthèse des principales solutions rencontrées, tant en ce qui concerne les rapports spontanés que les rapports expliqués.

1. Rapports spontanés. On peut distinguer trois voies nullement exclusives:

—Tenir compte de l'ordre des mots: c'est ce qui semble être fait dans W.R.U.

—Segmenter les analyses à un ou plusieurs niveaux. Cette solution a été adoptée dans Ruly English (segmentation à deux niveaux: item et composition), ainsi que dans W.R.U. (segmentation à quatre niveaux; syntagme, proposition, phrase et paragraphe).

—Utiliser des interfixes sans contenu sémantique, servant uniquement à indiquer que les mots affectés du même interfixe se concernent mutuellement. Cette solution a été adoptée dans Ruly English.

2. Rapports expliqués. On peut distinguer trois voies, qui sont cette fois difficilement conciliables entre elles, mais que l'on peut facilement associer avec les précédentes.

Elles consistent à employer respectivement:

—Une famille d'indicateurs de rôle précisant le rôle des mots dans le contexte; ces indicateurs sont très importants dans

W.R.U.; ils jouent, sous le nom d'opérateurs syntaxiques, un rôle secondaire dans SYNTOL.

—Un groupe de quelques relations binaires à contenu sémantique très large: c'est la voie suivie par SYNTOL, qui se limite à un groupe de quatre relations (prédicative, associative, consécutive, coordinative).

—Une série très riche de relations binaires précises, indiquant avec beaucoup de finesse la nature des rapports syntagmatiques. Cette voie a été choisie par le langage GRISA, qui distingue plus de vingt relations différentes lesquelles sont elles-même subdivisées.

Il apparaît ainsi que les voies envisagées offrent des possibilités combinatoires assez considérables. L'ensemble des états de cette caractéristique en découle. Il est infiniment plus nuancé que celui relatif à la précédente et il ne peut être analysé de la même façon.

(d) Les fondements de la règle de pertinence

Cette règle ne peut être définie avec précision qu'une fois l'élaboration du langage complètement achevée quant aux cinq premiers points décrits dans la section I. Toutefois, ses *fondements*, c'est-à-dire la manière dont les rapports paradigmatiques et syntagmatiques vont être utilisés dans la recherche documentaire, doivent être jetés très tôt, dès la conception même du langage, si l'on ne veut pas risquer de s'écarter du but de la documentation automatique. L'objet de cette dernière caractéristique est précisément de fixer ces fondements.

Les états en sont intimement liés à ceux des deux caractéristiques précédentes. C'est en effet chaque fois en fonction de la combinaison d'états retenus pour ces dernières, qu'il faut élaborer les lois qui doivent régir l'utilisation des rapports paradigmatiques et syntagmatiques.

Ce sont finalement ces lois qui définissent l'état cette quatrième caractéristique. Les possibilités combinatoires qui en résultent là encore rendent impossible l'énumération de ces états. Nous nous bornerons ici à illustrer sur quelques exemples simples (qu'il serait peut être souhaitable d'approfondir et de compléter ultérieurement) la nature de ces lois.

Pas de paradigmatie, pas de syntaxe—Ici aucun choix n'est offert, la règle s'impose complètement:

d R q = 1 si et seulement si l'ensemble des mots qui sert à représenter d contient (au sens ensembliste) celui qui sert à représenter q.

Dans ce qui suit, nous désignerons cette règle d'inclusion ensembliste par le symbole $\supset$.

Présence de paradigmatie, pas de syntaxe—On peut songer à des règles du type suivant:

—associer à la question q un ensemble Q d'autres questions déduites de q par substitution de mots conformément à des lois faisant intervenir les rapports paradigmatiques;

—et poser:

d R q = 1 si et seulement si il existe q′ dans Q tel que d $\supset$ q′.

On notera qu'une foule de possibilités s'offrent pour définir l'application q → Q.

Pas de paradigmatie, présence d'une segmentation à un niveau: on peut envisager la règle suivante:

d R q = 1 si et seulement si il existe pour tout segment q′ de q un segment d′ de d vérifiant d′ $\supset$ q′.

Pas de paradigmatie, présence de relations binaires on peut songer à la règle suivante, assez stricte et quelque peu dangereuse:

d R q = 1 si et seulement si d $\supset$ q et, pour tout couple de mots de q liés par une relation, on retrouve entre ces deux mots la même liaison dans d.

On peut assouplir cette règle de multiples façons: par exemple en autorisant des substitutions de relation conformes à certaines lois précises, ou encore en imposant à deux mots liés par une relation dans q d'être seulement reliés dans d par une chaîne (ayant certaines propriétés définies).

Ces quatre caractéristiques nous semblent recouvrir l'essentiel des points sur lesquels il est nécessaire de porter l'attention pour élaborer un langage documentaire,* tout au moins dans ses grandes lignes. Il nous reste maintenant à examiner les problèmes qui se posent au moment de fixer l'état de ces caractér-

*Pour plus de precisions on pourra consulter le rapport n° 3 remis au CETIS.

istiques en fonction des besoins des utilisateurs et des particularités de la population documentaire.

III. PROBLEMES FONDAMENTAUX

Ces problèmes viennent, nous l'avons dit, se nouer autour des caractéristiques qui ont été décrites ci-dessus. C'est pourquoi nous les examinerons en étudiant pour chacun d'elles les facteurs qui, selon nous, doivent entrer en jeu dans le choix des options les concernant.

(a) Problèmes que pose le dosage des rapports paradigmatiques et syntagmatiques

Choisir le dosage de la paradigmatie et de la syntagmatie est probablement l'une des options les plus fondamentales à prendre au niveau de la conception d'un système documentaire; comme nous allons le montrer, toute l'efficacité du système est en effet en jeu.

Considérons un langage documentaire (plus ou moins satisfaisant) et supposons que l'on enrichisse tour à tour les rapports paradigmatiques et syntagmatiques:

- —L'enrichissement des rapports paradigmatiques ne peut, à notre avis, qu'augmenter le nombre de documents décrétés pertinents pour une question donnée. Il en résulte une meilleure efficacité du système dans la mesure où les documents supplémentaires sont, pour la plupart, effectivement pertinents. Cette amélioration s'accompagnera, toutefois, presque inévitablement, d'un accroissement du nombre de documents parasites (voir Fig. 3).
- —L'enrichissement des rapports syntagmatiques ne peut, à notre avis, que diminuer le nombre des documents décrétés pertinents pour une question donnée. Il en résultera une meilleure efficacité du système, dans la mesure où la majeure partie des documents éliminés ne sont pas réellement pertinents. Cette amélioration s'accompagnera toutefois, presque inévitablement, d'un accroissement du nombre des documents omis (voir Fig. 4).

Les rapports paradigmatiques et syntagmatiques ont une influence à la fois antagoniste et complémentaire sur les réponses fournies par le système.

Celle-ci peut être résumée par la Fig. 5.

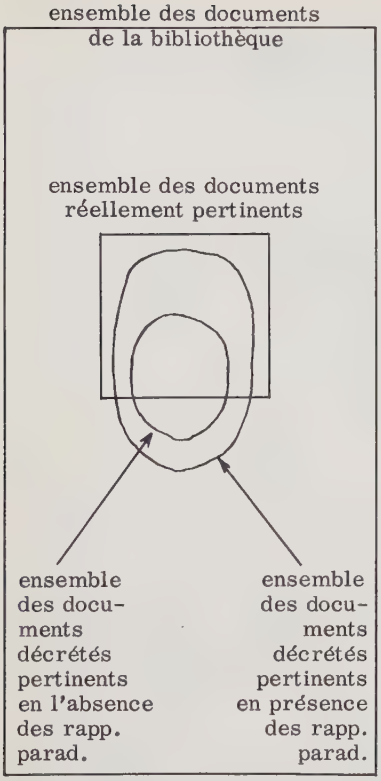


FIG. 3. Schématisation du cas où l'enrichissement de rapports paradigmatiques est efficace.

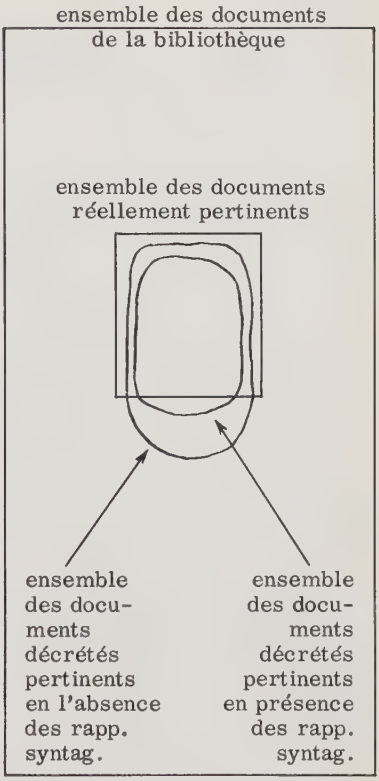


FIG. 4. Schématisation du cas où l'enrichissement de rapports syntagmatiques est efficace.

Cette figure met en lumière l'importance du dosage de chacun des deux types de rapports, dosage qui doit aboutir à un équilibre entre les deux critères antagonistes que constituent le nombre de documents parasites et le nombre de documents omis.

Cet équilibre est défini par les besoins des utilisateurs; de façon quelque peu schématique, on peut penser que:

—plus l'importance qu'ils attachent aux documents omis est grande, et plus on doit se déplacer vers le bas dans le tableau 1;

—plus l'importance qu'ils attachent aux documents parasites est grande, et plus on doit se déplacer vers la droite dans le tableau 1.

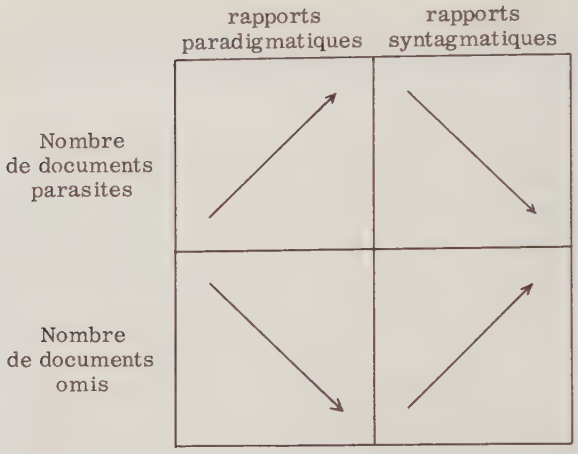


FIG. 5. Les flèches indiquent le sens de variation du nombre de documents qu'elles concernent, en fonction du type de rapports paradigmatiques ou syntagmatiques considéré.

Il est bien clair, d'autre part, que la place occupée par les deux types de rapports dans le domaine traité, doit elle aussi avoir une action non négligeable sur celle qu'il convient de leur réserver dans le langage: le dosage qui assurera l'equilibre souhaité en dépend dans une certaine mesure.

Enfin, le nombre de documents de la population documentaire peut, lui aussi, influencer ce dosage. C'est ainsi par exemple que si le nombre des documents est relativement faible (de l'ordre de quelques dizaines de milliers) et dans la mesure où les exigences des utilisateurs relatives aux documents omis et parasites restent suffisamment modérées, on peut accepter l'absence complète de chacun des deux types de rapports (cf. SAINT-GOBAIN). Cette absence deviendra intolérable si la bibliothèque passe de quelques dizaines à quelques centaines de milliers de documents.

(b) Problemes que pose la prise en compte des rapports paradigmatiques

Deux facteurs semblent jouer un rôle prépondérant dans le choix de la solution $\alpha\beta\gamma$ ou δ qu'il convient de retenir pour prendre en compte les rapports paradigmatiques. Ce sont:

—la nature des rapports existant dans le domaine traité,

—la vitesse plus ou moins grande d'évolution de la langue des documents, ou "démographie conceptuelle" du domaine.

Avant d'aborder les deux problèmes qui leur sont liés, mentionnons que la place dévolue à ces rapports par suite du dosage retenu peut aussi influencer directement ce choix. En effet, les solutions γ ou δ paraissent plus propices que les deux autres au maniement d'une grande densité de rapports paradigmatiques.

1. Influence de la nature des rapports paradigmatiques existants. Cette interaction est évidente: nous avons déjà souligné en présentant l'état γ que les types de structure paradigmatique pris en compte par β et γ étaient très différents en général. Illustrons toutefois cette interaction par un exemple précis.

Supposons que les rapports à prendre en compte soient tels (hypothèse faite dans C.D.S. ou SYNTOL) que l'on puisse se contenter d'associer à chaque mot un précédent ou plus (traduisant un rapport hiérarchique), le nombre de suivants étant quelconque. Il s'ensuit que le graphe P des rapports paradigmatiques (qui est une réunion d'arborescences) peut être pris en compte par un codage lexicographique des mots (état β).

Désignons en effet par n le nombre maximum de suivants que peut avoir un sommet, et choisissons n symboles ordonnés, tels que 1, 2, ..., n. On codera les sommets de rang 1 en affectant à chacun l'un de ces symboles (un même symbole ne pouvant être affecté à deux sommets distincts). Tous les sommets de rang p - 1 étant codés, on codera les sommets de rang p en ajoutant (à droite) au code de leur précédent (lequel est unique) l'un des symboles 1, 2, ..., n (le même symbole ne pouvant être affecté à deux sommets ayant le même précédent)(voir Fig. 6).

Il faut remarquer que cette solution ne permet pas de combiner aisément rapports hiérarchiques et rapports latéraux. En effet, puisqu'un mot ne peut admettre qu'un précédent, on ne peut exprimer simultanément:

—qu'il est suivant d'un autre au sens hiérarchique (Rome est en Italie)

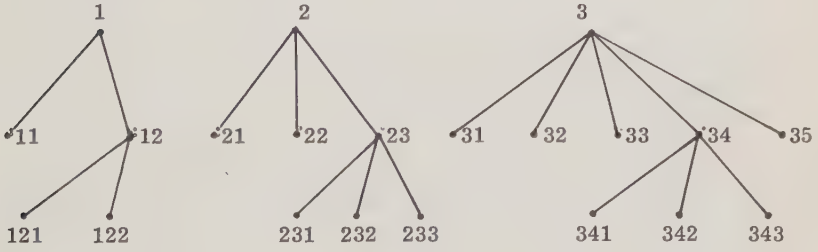


FIG. 6

—qu'il appartient à une classe marquant la présence de rapports latéraux entre les éléments de la classe (Rome est une capitale, ce qui est une manière de la lier à Paris).

Les seuls rapports latéraux que l'on peut utiliser sont ceux qui existent entre deux mots du fait que leur unique précédent est le même. Cette faiblesse du codage lexicographique le rend impropre à l'expression d'une grande richesse paradigmatique. SYNTOL ayant adopté cette solution, risque fort de se trouver gêné dans l'avenir; il devra alors, pour traduire ses rapports latéraux, opter pour l'état δ.

2. Influence de la démographie conceptuelle. Au fur et à mesure que l'on progresse dans le domaine traité, les concepts mis en jeu s'affinent et se multiplient. Cette évolution impose l'introduction de concepts nouveaux dans le langage documentaire, ceux-ci pouvant être entièrement nouveaux, ou, au contraire, dérivés de concepts anciens, par restriction, réunion, voire même déviation (concepts nouveaux empiétant sur plusieurs concepts anciens et ne s'identifiant pas avec eux par exemple). Le problème qui se pose est celui de l'assimilation de ces concepts nouveaux: c'est naturellement leur intégration dans la structure paradigmatique (établissement de leur rapport avec les concepts anciens) qui rend difficile une telle assimilation. Or, les états $\alpha\beta\gamma\delta$ ne presentent pas du tout les mêmes aptitudes à ce sujet:

Etat α: Il n'y a, en général, aucune difficulté à incorporer, dans le langage, des mots nouveaux, puisque cela conduit essentiellement à augmenter deux listes: celle M des mots, celle P des rapports paradigmatiques. Les seuls problèmes pourraient provenir de la manière dont a été enregistré le graphe P dans l'équipement.

Etat β: Si le procédé de codage adopté laisse suffisamment de "trous" pour incorporer des mots nouveaux, n'importe où et n'importe comment dans le graphe P, il n'y aura là encore aucun problème. Par contre, il en sera tout autrement si certaines "zones" du graphe P sont saturées.

Considérons par exemple le cas d'un codage lexicographique faisant intervenir n symboles (voir plus haut), et supposons qu'un mot nouveau doive être inséré comme suivant d'un mot qui en possède déjà n. Cela est impossible à moins de pouvoir introduire un nouveau symbole dans le codage.

Etat γ: Cet état est certainement le plus favorable à l'assimilation des mots nouveaux. En effet, on se trouvera toujours devant l'alternative suivante:
—ou bien les mots nouveaux s'expriment en fonction de générateurs de G, et sont incorporés dans le langage immédiatement;
—ou bien les mots nouveaux nécessitent l'introduction de générateurs supplémentaires, ce qui, en général, ne présentera pas de difficulté majeure.

Etat δ: Cet état, apparaissant comme un mélange en proportion variable des trois précédents, présente vis-à-vis de l'assimilation des mots nouveaux les avantages et inconvénients des précédents à des degrés divers. Toutefois, sa complexité ne peut qu'entraver cette assimilation.

(c) Problèmes que pose la prise en compte des rapports syntagmatiques

Il semble qu'une grande latitude soit offerte dans le mode de prise en compte des rapports syntagmatiques. Cela tient d'une part à la multitude des solutions possibles (cf. section II paragraphe c), et d'autre part, à la nature très lâche des liaisons qui s'exercent entre cette caractéristique est les autres facteurs susceptibles de l'influencer.

Certes, la nature des rapports syntagmatiques existant dans le le domaine étudié ne sera pas en général étrangère à leur mode de prise en compte dans le langage. C'est ainsi par exemple qu'on ne songera à tenir compte de l'ordre des mots que si celui-ci traduit effectivement quelque chose de précis dans des cas suffisamment nombreux. Mentionnons en revanche que la segmentation éventuelle que l'on peut retenir peut en raison des soucis différents auxquels elle répond, n'avoir que des liens très laches avec le découpage en proposition, phrase ou paragraphe.

Le facteur agissant le plus directement sur le mode de prise en compte des rapports syntagmatiques est sans doute la longeur des analyses des documents. Il est clair en effet que, plus le nombre moyen de mots intervenant dans ces analyse est grand, et plus l'introduction d'une interfixation ou celle d'un segmentation devient nécessaire.

(d) Problèmes que posent les fondements de la règle de pertinence

Les fondements de la règle de pertinence reposent en premier

lieu sur le mode de prise en compte de chacun des deux types de rapports et dépendent par conséquent dans une très large mesure des options relatives à ces deux caractéristiques.

Ils dépendent en second lieu du nombre moyen de mots entrant dans une question: plus ce nombre est grand, et plus le choix d'une règle convenable est délicat. En effet, si la question est courte, il est aisé d'énumérer tous les cas possibles quant à la structure syntagmatique de cette question; il est donc relativement facile d'en dégager les conditions de l'adéquation de cette structure avec celle d'une analyse. En revanche, si les questions sont longues, il n'est plus possible d'examiner en détail tous les cas possibles, les conditions d'adéquation que l'on est alors en mesure de formuler ne peuvent pas être très nuancées, elles risquent d'être trop lâches ou au contraire trop strictes.

Ainsi, par exemple, les substitutions de mots ou de relations (cf. section II, d) doivent se faire suivant des lois d'autant plus complexes que la question est plus longue.

Enfin, les aptitudes des utilisateurs à formuler leurs questions en liaison avec le langage jouent un rôle qui, dans certain cas, peut être très important quant aux fondements de la règle de pertinence. Ce rôle concerne avant tout l'exploitation des rapports paradigmatiques à propos desquels on peut imaginer, par exemple, des indications plus ou moins précises sur ceux qui doivent être utilisés dans la recherche des documents répondant à la question que l'utilisateur a posée. Il est donc possible d'ajouter à la question proprement dite certains renseignements qui viendront orienter la règle de pertinence.

Rappelons à titre d'exemple que SYNTOL prévoit des "modulations" dont la nature et le niveau peuvent être précisés par l'utilisateur.

CONCLUSION

Pour terminer, nous voudrions rappeler deux vérités essentielles qui nous paraissent souvent négligées:

- ce n'est pas un souci de traduction fidèle qui doit conditionner la richesse et le mode de prise en compte, des rapports paradigmatiques ou syntagmatiques, c'est seulement la nécessité de sélectionner, autant que possible, tous les documents pertinents et rien que ceux-là;
- ce n'est pas une "intersection" évidente entre document et

question qui permet de décréter la pertinence, mais c'est une règle souvent complexe et difficile à définir qui est étroitement liée aux deux types de rapports.

ANNEXE I. FONDEMENTS DE LA DISTINCTION ENTRE RAPPORTS PARADIGMATIQUES ET SYNTAGMATIQUES

Généralités

Chacun de ces deux types de rapports répond, en documentation automatique, à deux besoins dont l'influence varie selon les cas, mais que l'on ne peut jamais ni confondre ni satisfaire par une même structure. Pour les mettre en évidence, nous allons considérer un langage ne prenant en compte ni rapports paradigmatiques ni rapports syntagmatiques, et examiner en quoi un langage de ce type peut paraître insatisfaisant. Dans un tel langage, les analyses se réduisent à une suite de mots (voir SAINT-GOBAIN) dans laquelle l'ordre ne joue aucun rôle. Il s'ensuit que:

—Une question étant posée, certains documents peuvent ne pas présenter tous les mots requis par la question pour être décrétés pertinents, bien qu'ils le soient en fait: c'est ainsi, par exemple, que si l'on s'intéresse aux exportations françaises de céréales, tout document traitant des exportations françaises de blé ou d'orge sera rejeté, alors qu'il devrait être accepté.

—Une question étant posée, certains documents peuvent posséder tous les mots requis par la question, pour être décrétés pertinents bien qu'ils ne le soient pas en fait: c'est ainsi, par exemple, que si l'on s'intéresse aux exportations de la France vers l'Italie, tout document traitant des exportations de l'Italie vers la France sera accepté, alors qu'il devrait être rejeté.

Ainsi, il peut sembler nécessaire, dans certains cas, de pouvoir, de façon automatique et grâce au langage:

—agréger des documents mettant en jeu des mots distincts, lorsque ces mots présentent entre eux certains rapports essentiels universellement acceptés et par conséquent non explicites dans les documents. C'est à ce besoin d'*agrégation* que doivent répondre les rapports paradigmatiques.

---

*Cela ne veut nullement dire qu'un tel langage soit a rejeter, il peut, au contraire, sous certaines conditions, etre le meilleur.

—séparer des documents mettant en jeu les mêmes mots, lorsque ces documents établissent de façon explicite entre ces mots certains rapports qui varient d'un document à l'autre. C'est à ce besoin de séparation que doivent répondre les rapports syntagmatiques.

Ces besoins, tout comme les moyens qui servent à les satisfaire, s'opposent non seulement par leur nature—l'un discrimine, l'autre rapproche—mais aussi par le niveau auquel ils s'insèrent: l'un est interne aux documents, c'est-à-dire s'applique aux mots appartenant à une même document, l'autre est externe aux documents, c'est-à-dire s'applique aux mots, indépendamment des documents auxquels ils appartiennent.

Remarques importantes: Il convient de remarquer que les rapports paradigmatiques peuvent être exprimés par l'intermédiaire de rapports syntagmatiques établis entre les mots.

Cette remarque ne justifie en rien l'abandon de la distinction entre ces deux types de rapports: en effet, exprimer les rapports paradigmatiques n'est pas le but visé, il s'agit de le faire de manière à satisfaire au besoin d'agrégation, c'est-à-dire de pouvoir les utiliser au moment voulu dans la recherche documentaire. Il est donc indispensable de savoir reconnaître ceux des rapports syntagmatiques qui ont été établis en vue de représenter ces rapports paradigmatiques, de ceux qui ont été établis pour exprimer des rapports occasionnels et contingents au niveau d'un document; il serait très dangereux de confondre ces deux sortes d'information. C'est ainsi que si le titre "Paris et le désert français" était assimilé à une information paradigmatique, on pourrait être amené à considérer la France comme un désert! ...

Toutefois, rien ne s'oppose à ce que les rapports paradigmatiques soient représentés à l'aide des rapports syntagmatiques: la structure P est alors définie par un document spécial. Cette solution (qui est celle envisagée dans GRISA) convient aux cas α et δ. (Cf. mode de prise en compte des rapports paradigmatiques, section II paragraphe 2.)

TRAITEMENT DE L'INFORMATION PAR L'ALGEBRE DE BOOLE

P. Camion

Euratom Cetis,
Ispra, Italy

RESUME

Il est connu que des problèmes de circuits électriques avec contact a deux positions peuvent etre resolus par l'Algebre de Boole. R. Fortet a montre dans (1), comment on peut resoudre par l'Algebre de Boole des problemes tres divers tels que coloriage d'une carte de géographie au moyen de quatre couleurs de manière à ce que deux pays adjacents soient de couleurs différentes, ordonnancement des fabrications dans un atelier etc.

Poursuivant l'extension des applications de l'algèbre de Boole, nous avons montré dans (2) comment on pouvait résoudre des problèmes d'algèbre diophantiennes, puis comment on pouvait aborder au moyen de variables binaires des problèmes relatifs à des relations de préordre sur un ensemble fini.

Nous avons expérimenté avec succès une méthode donnée dans (2) de traitement sur ordinateur d'Algèbre de Boole mettant en jeu 36 variables binaires. L'ensemble des programmes mis au point constitue en quelque sorte un langage de programmation permettant de traiter de nombreux problèmes dits "non numériques".

TRAITEMENT DE L'INFORMATION PAR L'ALGEBRE DE BOOLE

Plan de l'Exposé

Nous allons voir comment, en traitant en machine les fonctions Booléennes, donc en effectuant certaines opérations sur les fonctions Booléennes, on peut résoudre une classe étendue de problèmes combinatoires, c'est à dire des problèmes d'arithmetique et d'algèbre posés sur des ensembles finis.
Cet exposé comportera 4 points essentiels:

1. Description des familles de problèmes susceptibles d'être résolus au moyen des fonctions Booléennes.
2. Types d'opérations rencontrés dans la résolution de ces problèmes.
3. Principes choisis pour le traitement de ces operations sur machine digitale.
4. Résultats obtenus et perspectives.

I. DESCRIPTION DES FAMILLES DE PROBLEMES SUSCEPTIBLES D'ETRE RESOLUS AU MOYEN DES FONCTIONS BOOLEENNES

a. Le Professeur Fortet (1) a montré que l'on pouvait résoudre par l'algèbre de Boole, même à la main, des problèmes de nature combinatoire posé sur des ensembles finis, problème des quatre couleurs, problèmes d'ordonnancement, etc. Il suggère que l'on adapte l'outil aux problèmes ou interviennent des entiers, par exemple les programmes linéaires pour lesquels on désire une solution entière.

b. Nous avons montré (2) comment on pouvait résoudre tout système d'égalité ou d'inégalité Diophantienne par l'algèbre de Boole, pour autant que l'on connaisse une borne pour chaque variable entière. En effet, le principe consiste à écrire une variable entière au moyen de n variables binaires, cette variable ne peut donc prendre que 2^n valeurs distinctes.

Chaque composante en écriture binaire d'une variable indépendante entière est une variable indépendante binaire. Chaque fonction de ces variables indépendantes obtenue en effectuant un nombre fini de fois les lois de compositions addition, soustraction, multiplication, puissance, sur ces variables et des entiers auront pour composantes en écriture binaire des fonctions Booléennes dont les arguments sont les composantes des variables indépendantes.

L'objet essentiel de (2) est de fournir les méthodes de constructions de ces dernières fonctions et des méthodes analytiques de recherche d'extréma de fonctions de variables entières, éventuellement sous contraintes.

Les problèmes combinés d'algèbre et d'arithmétique sur des ensembles finis peuvent être également mis en équation. (Traveling salesman problem, quadratic assignment problem (2) etc...).

Il devient impératif pour la famille b d'effectuer les opérations sur les fonctions Booléennes en machine.

c. Les problèmes de cheminement dans les graphes, si importants dans les problèmes d'ordonnancement, trouvent une mise en équation commode en terme d'Algèbre de Boole(3). Le principe en est que une relation de pré-ordre sur un ensemble fini (les chemins d'un graphe)(6) peut etre represente de maniere biunivoque par une équation Booléenne, l'implication entre variables binaires étant une relation de pré-ordre. A chaque élément de l'ensemble fini correspond une variable binaire, à chaque relation binaire entre deux éléments correspond une implication, la relation de pré-ordre sera donc représentée par la conjonction de toutes ces implications.

Une application, d'ailleurs expérimentée avec succès, consiste en la recherche de tous les circuits hamiltoniens d'un graphe.

II. TYPES D'OPERATIONS RENCONTRES DANS LA RESOLUTION DE CES PROBLEMES

Les lois de compositions sur l'ensemble B de deux éléments notés 0 et 1 sont

$\dot{+}$	0	1
0	0	1
1	1	1

$\dot{+}$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

$\dot{+}$: co-plus
$\dot{+}$: contra-plus
$\cdot$: multiplié par.

Et l'opérateur de négation - : non-zéro égale un
non-un égale zéro.

Les fonctions Booléennes sont définies par l'application des lois de compositions et de l'opérateur un nombre fini de fois sur un nombre fini d'arguments (variables indépendantes), ou de fonctions Booleennes.

Toutefois $\dot{+}$ et $\dot{+}$ ne seront pas utilisés simultanément pour une même fonction ni pour toutes les fonctions intervenant dans la résolution d'un même problème. Si ces deux lois apparaissent lors de la mise en équation on transforme les expressions de façon à ce qu'une seule subsiste.

De plus on fait au départ les transformations nécessaires à ce que l'opérateur de négation ne porte que sur les variables indépendantes.

En résumé les fonctions seront toujours ramenées à une forme normale: co-somme de produits de variables niées et non niées ou contra-somme de produits de variables niées et non niées.

Ceci dit, les opérations à effectuer pour résoudre un problème seront:

a. Produit de fonctions.

b. Co-somme ou contra-somme de fonctions.

c. Négation d'une fonction (cela veut dire trouver la forme normale de la négation d'une fonction donnée sous forme normale).

d. Elimination par produit d'un sous-ensemble de variables. (Pour une fonction $f(x_1, \ldots, x_n)$ éliminer x_1 par produit signifie effectuer le produit $f(1, \ldots, x_n) \cdot f(0, \ldots, x_n)$).

e. Passer d'une forme normale co-plus à une forme normale contra-plus et réciproquement.

Cette liste n'est pas exhaustive, elle définit un programme de travail, la combinaison des opérations a, b, c, d, e permettant de résoudre une classe étendue de problèmes.

III. PRINCIPES CHOISIS POUR LE TRAITEMENT DE CES OPERATIONS SUR MACHINE DIGITALES

Nous n'utilisons pas la méthode de R. Ledley consistant (4) à représenter chaque fonction de n variables par un mot de 2^n bits malgré ses grands avantages. Nous avons en effet traité des fonctions de 36 variables, ce qui n'aurait pas été possible par cette méthode. Notre méthode (3) consiste à représenter chaque monôme d'une forme normale par un mot binaire de 2n bits ou selon la récente suggestion de A. Debroux qui dirige l'éxecution du projet, par un couple de mots binaire, chacun de n bits. Une fonction en forme normale sera donc représentée par un ensemble de couples de mots. Dans cette dernière version, à chaque variable correspond une position du couple de mots de n bits, les deux bits de cette position seront respectivement 01, 10, 11, 00 selon que la variable apparait dans le monôme sous la forme x, $\bar{x}$, 1 (c'est à dire que la variable ne figure pas dans le monôme) ou $x \cdot \bar{x}$, cas qui peut apparaitre lorsque l'on effectue le produit de deux monômes. Dans ce dernier cas le monôme doit être effacé. Le test consistera à faire la co-somme des deux mots du couple et à vérifier si le résultat est différent du mot qui a 1 en chaque

position. Pour effectuer le produit de deux monômes, on voit qu'il suffit d'effectuer le produit bit à bit de leur représentation. Pour effectuer le produit de deux fonctions il suffit de multiplier chaque monôme de la première par chaque monôme de la seconde et grouper le résultat dans un ensemble repéré de mémoires. Le test $x \cdot \bar{x}$ suivi d'effacement éventuel est effectué ensuite. Egalement certaines simplifications, celles du type:

$$x_1 x_2 x_3 \dot{+} x_1 x_2 x_3 x_4 = x_1 x_2 x_3$$

$$x_1 x_2 x_3 \dot{+} x_1 x_2 \bar{x}_3 = x_1 x_2$$

La programmation de cette simplication demande une grande habilité, il faut en effet éviter tant que possible de comparer des monômes qui ne donneront pas lieu à simplification.

Bien que R. Monterosso et A. Debroux aient choisi une methode pour les premières expériences, le projet définitif n'est pas arrêté. Il faudra en fait, selon A. Debroux, différents sous-programmes selon le nombre de variables et de monômes des fonctions. Il ne fait aucun doute que l'efficacite de ce sous-programme déterminera l'efficacité de la méthode dans son ensemble.

La co-somme et contra-somme n'étant pas utilisées simultanément, elles auront la même représentation, c'est à dire réunion logique de deux sous-ensembles de mots-monômes, suivi de simplification.

On imagine aisément comment les opérations c, d, e precitées peuvent faire chacune l'objet d'un sous-programme.

La représentation des fonctions sous forme normale pour le traitement des problèmes combinatoires se justifie par ceci:

Les solutions d'un problème combinatoire sont en général peu nombreuses et l'énoncé s'exprime en peu de mots.

Lorsque ces solutions sont données par le traitement de fonctions Booléennes elles apparaissent sous la forme de mots binaires, la position i de ce mot étant 0 ou 1 selon que la variable x_i vaut 0 ou 1 pour cette solution.

L'expression finale, qui donne l'ensemble des solutions, pourrait être construite en faisant la co-somme ou contra-somme des produits complets relatifs à ces solutions. Donc cette expression finale en forme normale (après simplification) aura generalement moins de termes qui'il n'y a de solutions. Puisque l'enonce s'exprime en peu de mots, de meme que son expression algebrique s'ecrit generalement en peu de symboles (relativement aux possibilites d'une machine electronique) et que l'expression

finale est restreinte, le seul danger est de passer par des fonctions intermédiaires pratiquement impossibles à écrire.

IV. RESULTATS OBTENUS ET PERSPECTIVES

a. Recherche des ensembles intérieurements stables maximaux d'un graphe. Soit X un ensemble fini et R une relation binaire symétrique définie sur X. Si iRj on dit que i et j sont adjacents. On peut définir cette relation par une matrice carrée, à l'ensemble des lignes ou des colonnes de la matrice M correspondent biunivoquement X. Le coefficient à l'intersection de la ligne i et de la colonne j (i, j $\in$ X) vaut 1 si iRj sinon il vaut 0. M est symétrique. Un sous-ensemble Y C X est dit intérieurement stable si la sous-matrice X carrée qui lui correspond est nulle (chaque coefficient vaut 0). K. Maghout (5) a mis ce problème en équation Booléenne de façon très simple. A chaque élément de x on fait correspondre biunivoquement une variable binaire. Chaque sous-ensemble Y est défini par un système de valeurs attribué aux variables par:

$$i \in Y \iff x_i = 0$$

Pour tout couple d'éléments tels que jRi on ne peut avoir

$$x_i = 0 \quad \text{et} \quad x_j = 0$$

On doit donc avoir $x_i \dot{+} x_j = 1$

[1]

donc:

$$\prod_{\{j \mid j\, R\, i\}} (x_i \dot{+} x_j) = 1$$

[2]

ou:

$$x_i \dot{+} \prod_{\{j \mid j\, R\, i\}} x_j = 1.$$

[3]

et

$$\prod_{i \in X} (x_i \dot{+} \prod_{\{j \mid j\, R\, i\}} x_j) = 1.$$

En effectuant le premier membre de [3] on trouvera une forme normale qui après simplification donnera tous les ensembles intérieurement stables maximaux, l'ensemble des variables apparaissant dans chaque terme définissant le complémentaire d'un tel ensemble. Ce complémentaire est minimal à cause des règles de simplification pour la co-somme. Ci-dessous l'énoncé d'un tel problème pour un ensemble de 20 éléments. Il s'agit donc pour la machine d'effectuer le produit de 20 fonctions, à

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}	x_{20}
x_1			1	1	1	1	1	1												
x_2															1	1	1	1	1	1
x_3	1			1				1	1					1						
x_4	1		1		1				1	1										
x_5	1			1		1				1	1									
x_6	1				1		1				1	1								
x_7	1					1		1				1	1							
x_8	1		1				1						1	1						
x_9			1	1						1				1	1	1				
x_{10}				1	1				1		1					1	1			
x_{11}					1	1				1		1					1	1		
x_{12}						1	1				1		1					1	1	
x_{13}							1	1				1		1					1	1
x_{14}			1					1	1				1		1					1
x_{15}		1							1					1		1				1
x_{16}		1							1	1					1		1			
x_{17}		1								1	1					1		1		
x_{18}		1									1	1					1		1	
x_{19}		1										1	1					1		1
x_{20}		1											1	1	1				1	

simplifier au fur et à mesure. Elle effectue ce travail en moins de deux minutes. Les fonctions les plus importantes après simplification ont 167 monômes.

Il y a 160 ensembles intérieurement stables maximaux dont les 7 plus grands, classés à la fin, ont 6 éléments.

Ils sont: 3, 5, 7, 15, 17, 19
5, 8, 9, 12, 17, 20
4, 6, 8, 15, 17, 19
4, 6, 8, 16, 18, 20
4, 7, 11, 14, 16, 19
3, 5, 7, 16, 18, 20
3, 6, 10, 13, 15, 18.

b. Le deuxième problème consiste en la recherche des circuits hamiltoniens du graphe de 12 sommets donné par la matrice d'incidence:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}
x_1		1				1		1	1			
x_2			1	1			1					1
x_3		1			1	1		1				
x_4							1		1	1		1
x_5	1		1			1					1	
x_6		1			1		1				1	
x_7	1	1						1			1	
x_8		1			1					1		1
x_9			1			1				1		1
x_{10}				1			1		1		1	
x_{11}	1			1	1		1					
x_{12}		1					1		1	1		

La methode utilisee est celle decrite en (3).

Le problème général n'a pas pu être traité parce que le programme ne pouvait pas traîter plus de 1.500 monômes.

En introduisant la contrainte supplémentaire que le circuit hamiltonien passe par quatre arcs donnés et pour trois contraintes différentes, la machine donne tous les circuits hamiltoniens pour tous les trois cas en cinq minutes.

Pour ces trois cas, les fonctions les plus importantes comportaient après simplification 509, 307 et 557 monômes.

Il y avait respectivement 3, 0 et 8 solutions. Pour la contrainte:

le circuit doit passer par les arcs (3,6), (4,9), (5,11), (11,4), par exemple les solutions sont:

(1, 8, 2, 3, 6, 5, 11, 4, 9, 12, 10, 7, 1)
(1, 8, 5, 11, 4, 9, 3, 6, 2, 12, 10, 7, 1)
(1, 8, 12, 2, 3, 6, 5, 11, 4, 9, 10, 7, 1)

PERSPECTIVES

A. Debroux a arrêté un plan d'éxecution d'un programme général de traitement d'algèbre de Boole, le nombre de variables et de monômes étant a priori illimité (usage de la bande).

Ce programme consisterait en l'ensemble des sous-programmes exécutant les opérations décrites ci-dessus, cette exécution faisant appel à l'une ou l'autre méthode selon le nombre de variables et de monômes en jeu de façon à ne faire des comparaisons et donc des simplifications entre monômes que lorsque au total il y a un réel gain de temps. Ces sous-programmes pourraient être appelés dans un langage Fortran, de sorte que, pourvu qu'un problème ait été mis en équation Booléenne, la programmation de sa résolution deviendrait extrêmement aisée.

BIBLIOGRAPHIE

1. R. Fortet, "L'algèbre de Boole et ses applications en Recherche Opérationnelle" Cahiers du Centre d'Etudes de Recherche Opérationnelle, No. 4, 1959.
2. P. Camion, "Une méthode de résolution par l'algèbre de Boole des problèmes combinatoires où interviennent les entiers", Cahiers du Centre d'Etudes de Recherche Opérationnelle, Vol. 2, No. 3, 1960.
3. P. Camion, "Une méthode de résolution par l'Algèbre de Boole de problèmes où intervient la relation de pré-ordre d'un graphe", Note Euratom.
4. R. Ledley, "Mathematical Foundations and Computational Methods for a Digital Logic Machine", J.O.R.S.A. Vol. 2, No. 3., 1954.
5. M. K. Maghout, "Sur la détermination des nombres de stabilité et du nombre chromatique d'un graphe", Extrait des "Comptes rendus des séances de l'académie des Sciences" - t. 248, p. 3522-3523, séance du 22 juin 1959.
6. C. Berge, "Theorie des graphes et ses applications," Durnod, 1958.
7. P. Camion, "Quelques propriétés des chemins et circuits hamiltoniens dans la théorie des graphes", Cahiers du Centre d'Etudes de Recherche Opérationnelle Vol. 2, No. 1 1960.

A PROGRAM FOR THE AUTOMATIC SOLUTION OF ORDINARY DIFFERENTIAL EQUATIONS WITH TWO POINT BOUNDARY CONDITIONS

A. Gibbons

University of Manchester
England

SUMMARY

A phrase structure language is described by means of which a compiler reads differential equations and differentiates their right hand sides with respect to the unknowns in the system. This enables the program which contains the compiler to find both single solutions and families of solutions to the equations. Some examples are given of the programs that the compiler will accept.

INTRODUCTION

This paper describes some aspects of a program, written by the author in conjunction with C. B. Haselgrove, for the solution of non-linear differential equations with two-point boundary conditions. In effect, the program is a special-purpose autocode input routine, the input language consisting of the equations, the boundary conditions and other pertinent information. From this data the program constructs several routines which are later used by a Runge-Kutta integration routine, by an initial conditions setting routine and by the control routine. It is the aim of this paper to describe the significant features of the input routine which reads the differential equations. Before we can see what is involved we must first indicate how the equations are to be solved.

We suppose that the boundary conditions at each end of the range of integration depend on some unknowns c_1, c_2, ... These

c_j may also occur directly in the differential equations as parameters or eigenvalues. We write the differential equations as

$$\frac{dy_i}{dx} = f_i(x, y, c) \qquad i = 1, 2, \ldots, n$$

or in vector notation

$$y' = f$$

It is also convenient to have the constants π, k_1, k_2, ... and, since it is not always desirable to define an equation in a single instruction, the auxiliary variables a_1, a_2, ... These themselves may be defined by means of differential equations provided the initial conditions are known at both ends of the range. We also make use of the elementary functions sin, cos, exp, log, and sqrt.

THE METHOD OF SOLUTION

In describing the method of solution, we find it convenient to suppose that the equations are all of first order. This restriction is not imposed on equations which may actually be presented to the program.

The method we use is to choose an appropriate set of values for the c_j and then integrate from both ends of the range in to some fitting point. Unwanted solutions can often be more easily avoided if the fitting point is not at either end of the range. We then change the values of the c_j by a suitable amount and repeat the process until either a solution is obtained or there is no further improvement.

If the values of the vector y are y_1 and y_2 at each side of the fitting point and $\Delta y = y_1 - y_2$ then we wish to solve the set of non-linear equations

$$\Delta y = 0$$

where Δy is a function of the c_j. Given an approximate solution to these equations we could improve it if we could solve the linear equations

$$\sum_j \frac{\partial}{\partial c_j} (\Delta y_i)\, \delta c_j = -\Delta y_i \qquad (1)$$

and add the increment δc, or some fraction of it, to the vector c.

However the program can be made much more powerful if we allow the effective number of y's to be one less than the number of c's. The solution of the equations then is not just a single point in c_j-space but a curve. It is also convenient to allow more equations than are strictly required to define a problem. The result of this is that the matrix of coefficients

$$\frac{\partial}{\partial c_j}(\Delta y_i)$$

is not square and the linear equations (1) cannot be solved. C. B. Haselgrove (1961) has devised methods that do give a value for δc and the methods described in sections 3 and 6 of his paper have been incorporated in the program.

The derivatives $\partial y_i / \partial c_j$ which determine the coefficients of the matrix may be found by integrating further sets of differentiated differential equations

$$\frac{d}{dx}\left(\frac{\partial y_i}{\partial c_j}\right) = \frac{\partial f_i}{\partial y_k}\frac{\partial y_k}{\partial c_j} + \frac{\partial f_i}{\partial c_j} + \frac{\partial f_i}{\partial x}\frac{\partial x}{\partial c_j} \tag{2}$$

If the initial conditions of the equations

$$y' = f$$

are functions of the c_j then the initial conditions of equations (2) may be found by differentiating these functions.

It is not always satisfactory in practice to fit the equations at some particular value of x so we introduce a function $g = g(x, y, c)$ and fit the equations on the surface $g = 0$. To calculate the elements of the matrix now, however, we require the derivatives

$$\left(\frac{\partial y_i}{\partial c_j}\right)_{g=0} = \left(\frac{\partial y_i}{\partial c_j}\right)_{x\ \mathrm{const}} + \frac{dy_i}{dx}\frac{\partial x_{g=0}}{\partial c_j}$$

On the surface $g = 0$, and writing $x = y_0$

$$\delta g = 0 = \frac{\partial g}{\partial y_k}\delta y_k + \frac{\partial g}{\partial c_j}\delta c_j$$

and

$$\delta y_k = f_k \delta x_{g=0} + \frac{\partial y_k}{\partial c_j} \delta c_j$$

where we employ the usual summation conventions and k ranges from 0 to n. We now have

$$\delta g = 0 = \frac{\partial g}{\partial y_k} \left(f_k \delta x_{g=0} + \frac{\partial y_k}{\partial c_j} \delta c_j \right) + \frac{\partial g}{\partial c_j} \delta c_j$$

and hence

$$\frac{\delta x_{g=0}}{\delta c_j} = - \left\{ \frac{\partial g}{\partial y_k} \frac{\partial y_k}{\partial c_j} + \frac{\partial g}{\partial c_j} \right\} \Big/ \frac{\partial g}{\partial y_k} f_k$$

Finally

$$\left(\frac{\partial y_i}{\partial c_j} \right)_{g=0} = \left(\frac{\partial y_i}{\partial c_j} \right)_{x \text{ const}} - \frac{f_i}{\gamma} \left\{ \frac{\partial g}{\partial y_k} \frac{\partial y_k}{\partial c_j} + \frac{\partial g}{\partial c_j} \right\}$$

where

$$\gamma = \frac{\partial g}{\partial y_k} f_k$$

If we fitted the equations on the surface $y_1 = 0$, for example, the derivatives

$$\left(\frac{\partial y_1}{\partial c_j} \right)_{g=0}$$

would all be zero and we should have "lost" a row of the matrix. To compensate for this and to ensure that the final solution we obtain is continuous, we always include the equation

$$\frac{\partial}{\partial c_j} (\Delta y_0) \delta c_j = - \Delta y_0$$

It is of no consequence that this is sometimes superfluous as the method of solution allows for more, consistent, equations than unknowns.

THE PHRASE STRUCTURE LANGUAGE

The problem we have, then, is to integrate both the original equations and the differentiated equations, to form a matrix equation and then to solve this to improve the value of c. It will be evident that finding the differentiated equations and their initial conditions and calculating the matrix is by no means a trivial job, although it is a "mechanical" procedure. It is the purpose of our input routine to perform these tasks for us. The input routine is based on phrase structure principles and the most important aspect of it is that for each expression read two subroutines are compiled, one to calculate the expression and the other to calculate its derivative with respect to c_j. It is sufficient to describe in detail how this has been done for the equations.

We first give our notation. We have a set of class definitions which define the format of our language and a set of procedure definitions, one for each element of each class, which define the meaning of the element. In the text the qualifiers "$\cdot$" and "[ ]" are used to indicate that the item so marked can in the first case appear an arbitrary number of times and in the second need not appear at all. Thus

$\dot{x}yz$ means xyz, xyyz, xyyyz, ...

x[y]z means xz, xyz

and

x[$\dot{y}$]z means xz, xyz, xyyz, ...

A program for Mercury is prepared on a teleprinter and must consist of combinations of the following symbols

a b c d e f g h i j k l m n o p q r s t u v w x y z π ?

. 0 1 2 3 4 5 6 7 8 9 + − ψ / , > $\geq$ = $\neq$ * ( ) → ′ $\approx$

Since only lower case letters are available we can, without risk of confusion, use upper case letters in the text for class identifiers. A class identifier may be used both as the name of the class and as an abbreviation for any element of the class. Thus, if we define D to be the class of all decimal digits we refer to "a decimal digit D".

We define a class by enumerating its elements and an element is simply a string of qualified or unqualified basic teleprinter

symbols or class identifiers, the latter being abbreviations for any elements of those classes. We also have the convention that Z, which will be defined in terms of the other classes, is the class of all possible instructions, or formats, in the input language. Some possible elements of class Z are

y1'' = c3cos(x)sin(y1)

y1'' = ((y1)2 − 1)/a1

a3 = k2/a1sqrt(a1)

y2'' = 0.125a2(1−(1+2x(1+x))(a2)4)

(the lack of teleprinter symbols makes it necessary for us to write $(y_1)^2$ as (y1)2)

THE CLASS DEFINITIONS

The classes are defined so as to distinguish between quantities that are functions of the c_j (variables) and those that are not (constants). This simplifies those procedures which are concerned with differentiation. In the definitions, semi-colons, ";", are used to separate the elements of a class.

We first define a decimal digit

D : 0;1;2;3;4;5;6;7;8;9

an integer

N : $\dot{D}$

and a general numerical constant

K : N[.]; [N] . N

We define a dash, "'", which denotes differentiation

Y : '

When an auxiliary variable has been defined by means of a differential equation, it must be a constant with respect to the c_j; we define the class A to be the class of integers which correspond to these auxiliary variables.

We can now define the left hand side of an equation

L : yN$\dot{Y}$; aA$\dot{Y}$; aN

The following classes need little explanation.

A sign

S : +; −

A function

F : sin; cos; exp; log; sqrt

A constant

C : x; K; kN; aA[$\dot{Y}$]; π

A constant factor

E : C; (G)[N]; F(G)

The class G will be defined to be a constant expression. Thus a permissible constant factor is a constant expression in brackets possibly raised to a positive integral power.

We sometimes find it very convenient to say that some element is a member of a class provided it is not followed by something else. Thus we could define a constant product to be an arbitrary number of constant factors not followed by a variable factor. This "not" principle is used in defining class B.

B : not S; c.r.;); /

The c.r. (carriage return) tape symbol is used to terminate each equation. Thus each element of class B is the negation of one which would terminate a product. We now make use of B to define a constant product

W : $\dot{E}$ not followed by B

A single constant term is

O : C not followed by either / or B

a constant denominator is

I : /W

and a constant term is

P : [S]O; [S]W[I]

A constant expression is now

G : $\dot{P}$ not followed by S

In a similar way we can define a variable factor

V : x; yN[$\dot{Y}$]; aN; cN

x is usually a "constant", but it must be treated as a "variable" if the range of integration is variable. Consequently, for a particular problem, x is deleted from the definition of either V or C.

A single variable is

U : V not followed by either / or B

A factor is

Q : E; V; (H)[N]; F(H)

where H will be defined to be an expression; and a product is

X : $\dot{Q}$

A denominator is

J : /X

a term is

T : P; [S]U; [S]WJ; [S]X[I]; [S]X[J]

and an expression is

H : $\dot{T}$

We can now define the right hand side of an equation to be

R : G; H

and the whole equation to be

Z : [$\dot{c.r.}$] L=R c.r.

For reference purposes we give a list of the class definitions in alphabetical order.

A : class of integers N corresponding to the auxiliary variables defined by differential equations.

B : not S; c.r.;); /

C : x; K; kN; aA[$\dot{Y}$]; π	a constant
D : 0; 1; 2; 3; 4; 5; 6; 7; 8; 9	a decimal digit
E : C; (G)[N]; F(G)	a constant factor
F : sin; cos; exp; log; sqrt	a function
G : $\dot{P}$ not followed by S	a constant expression
H : $\dot{T}$	an expression

I : /W	a constant denominator
J : /X	a denominator
K : N[.]; [N].N	a numerical constant
L : yNẎ; aAẎ; aN	a left hand side
N : Ḋ	an integer
O : C not followed by / or B	a single constant
P : [S]O; [S]W[I]	a constant term
Q : E; V; (H)[N]; F(H)	a factor
R : G; H	a right hand side
S : +; −	a sign
T : P; [S]U; [S]WJ; [S]X[I]; [S]X[J]	a term
U : V not followed by / or B	a single variable
V : x; yN[Ẏ]; aN; cN	a variable
W : Ė not followed by B	a constant product
X : Q̇	a product
Y : '	a dash
Z : [ċ.ṙ.] L=R c.r.	an equation

INTRODUCTION TO THE PROCEDURE DEFINITIONS

Let us now sketch the meaning of our classes, or, in other words, how we interpret them. From each expression we shall compile two routines which we call P and P' which will calculate the expression and its partial derivative respectively. We have defined an equation to be

left hand side = constant or variable expression

and in general the two routines we compile will have the form

P	P'
A = expression	A = (expression)'
l.h.s. = A	(l.h.s.)' = A

where A stands for the accumulator.

An expression consists of a number of terms, for example

$$E = T_1 + T_2 - T_3$$

and the compiled routines would be

P	P'
$A = T_1$	$A = T_1'$
$A = A + T_2$	$A = A + T_2'$
$A = A - T_3$	$A = A - T_3'$

Certain simplifications are possible if a term is constant, as then its derivative, being zero, can be ignored.

A term is a product of factors, thus if

$$T = F_1F_2F_3$$

we should compile

P	P'
$A = F_1$	$A = F_1'$
.....	.....
$W_1 = A$	$A = A \times F_2$
$A = A \times F_2$	$W = A$
.....	$A = W_1$
$W_2 = A$	$A = A \times F_2'$
$A = A \times F_3$	$A = A + W$
.....	.....
	$A = A \times F_3$
	$W = A$
	$A = W_2$
	$A = A \times F_3'$
	$A = A + W$
	.....

The working space locations $W_1, W_2, \ldots$ form a listed set; each one can be used only once throughout the whole program. We refer to them as W_l. We also make use of nested working space for both P and P' which we refer to as W_n and W_n'.

We differentiate functions of expressions

$$F(E)$$

by introducing a function F' to correspond to each F and then constructing the two sequences

P	P'
$A = E$	$A = E'$
$W = A$	$A = A \times W_l$

```
exit to subroutine
.....
A  = F'(A)
W1 = A
A  = W
A  = F(A)
← return
```

(W is a working space location)
Thus to calculate cos(E) the subroutine would be

```
A  = −sin(A)
W1 = A
A  = W
A  = cos(A)
return
```

Much of the complication in the differentiation arises from the fact that constants and variables are dealt with differently. Thus in the case of a denominator,

$1/D$

if it is a constant, we have

P	P'
A = D	–
A = 1/A	

whereas if it is a variable we have instead

A = D	A = D'
A = 1/A	A = A × W_1
W = A	
$W_1 = -A^2$	
A = W	

We describe these processes more precisely by giving the procedure definitions. These, as their name implies, give the procedure to be followed when an element of a class is present. When a class is defined in terms of other classes the procedure may merely call in the procedures for the sub-classes in an appropriate order. This is not sufficient, for at some stage we arrive at a basic level and must actually do something such as compile an instruction. All the different basic operations that are required are written as subroutines (sets of machine instructions) which we call Z-routines. We choose the letter Z

as it never occurs otherwise in a procedure. If Z7, for example, is the subroutine which simply jumps to read the next instruction then we write the procedure for the first, and only, element of class Z as

Z1 : R4; L2; Z7

The integers after the R and L indicate which positions they occupy in the element of Z. The sub-procedures are performed in the order in which they are written.

If the same sequence of Z-routine entries occurs in different procedures it is worthwhile calling it a "π-routine", and then just referring to that. A π-routine may also be defined in terms of other π-routines.

Where a sub-class in an element may occur an arbitrary number of times a corresponding number of procedures is performed.

A Mercury machine instruction consists of a function followed by an address. In locations 0 and 2 are stored the numbers 0 and -1 respectively.

The following function codes are used in the procedure definitions. A stands for the accumulator, S for a store line and ' means the new value of the location.

400	A' = S
410	S' = A
420	A' = A + S
430	A' = A − S
500	A' = A × S

We also use the instructions

400	0	A' = 0
520	2	A' = −A
111	1	A' = 1/A
590	v101	A' = 1/A

A full description of machine coding for the Mercury computer is given in reference 2.

The two sets of instructions that we compile, P and P', make use of working space locations W_n, W_n' and W_1. We associate with each term in an expression a sign (+ or −), a switch (constant or variable), a term function (T) and a term' function (T'); T and T' may be "400", "420" or "430". Associated with each

factor in a product is a factor function (F_f) which may be "400" or "500". For each partial product there is a switch to say whether it contains any variable factors. We have two Z-routines which plant an instruction in P or P' and we find it convenient to have four quantities F, addr, F', addr'. We count the dashes which may occur after a variable in a location d and we also use the locations p, i, j, m and k.

As a refinement, whenever we compile an expression, we deal first with the multiple terms and then with the single terms; similarly with a product.

THE PROCEDURE DEFINITIONS

In the procedures which follow we indicate by Z or π which parts of them consist of Z-routines or π-routines.

```
Z1 : R4; L2
     read next instruction    (Z)

R1 : G1
     P' : 400 0   (Z)
     exit         (Z)

R2 : H1
     exit

G1 : nest local variables     (Z)
     set +, T=T'=400          (Z)
     set "constant" term      (Z)
     set "multiple" terms     (Z)
     P1
     set "single" terms       (Z)
     P1
     restore local variables  (Z)
     exit
```

The "local variables" referred to in this procedure are T, T', F_f, k, ±, constant or variable term, and whether or not variable factors are present in a product. It is necessary to preserve these in a nest, as an expression may be defined in terms of other expressions.

```
H1 : nest local variables
     set +, T = T' = 400
     set "multiple" terms
     T1
     set "single" terms
     T1
     restore local variables
     exit

P1 : ignore if doing "multiple" terms   (Z)
     set +
     S1, 02
     T = 420   (Z)
     exit

P2 : ignore if doing "single" terms   (Z)
     set +, S1
     store A   (π1)
     I3, W2
     restore A   (π2)
     T = 420
     exit
```

We now define some π-routines.

$\pi 3$: P : 410 W_n (Z)
n = n − 1 (Z)

The effect of this is to store the contents of the accumulator in the P program in a nest.

$\pi 4$: P' : 410 W_n' (Z)
n' = n' − 1 (Z)

That is, nest the contents of the accumulator in the P' program.

$\pi 5$: n = n + 1
P : F; W_n

$\pi 6$: n' = n' + 1
P' : F', W_n'

These two π-routines are used to take values out of the nests. The functions F and F' are set before use to suitable values.

In the next π-routine we use a conditional jump instruction. This transfers control if the condition is fulfilled.

```
π1 : F_f = 400
     jump 1, T = 400   (Z)
     jump 2, constant term   (Z)
     jump 2, T' = 400   (Z)
     π4
   2)π3
   1)exit
```

This π-routine nests the contents of the accumulators in P and P' if necessary.

```
π2 : jump 1, +
     jump 2, constant term
     P' : 520 2
   2)P  : 520 2
   1)F = F' = 420
     jump 3, T = 400
     jump 4, constant term
     jump 4, T' = 400
     π6
   4)π5
   3)exit
```

π2 signs the accumulator and adds on the previous terms and their derivatives if any. We now return to procedures.

```
S1 : exit
S2 : set -
     exit

01 : C1
     jump 1, +
     jump 2, T = 420
     P : 400 0
   2)T = 430
   1)F = T   (Z)
     P : F, addr    (π10)
     exit
```

C1 : the procedure for each C is to form the address of the constant and plant it in the locations addr and addr'. In the case of C2, a numerical constant, the number is first calculated in the floating point accumulator and then added to a list of numbers, if it does not already occur. Its absolute address is then recorded in addr and addr'.

```
I1  : W2
      P : 111  1
      P : 590 v101
      exit

W1 : set multiple terms
      E1
      set single terms
      E1
      exit

E1 : ignore if doing multiple terms
      C1
      F = F' = F_f
      P : F, addr    (π10)
      jump 1, constant term
      P' : F', addr'    (π9)
    1)F_f = 500
      exit

E2 : ignore if doing single terms
      jump 1, F_f=400
      π3
    1)G2
      set p = exponent i.e. N4    (Z)
      jump 2, p < 2
      P : A = A^p    (π12)
    2)F = F' = F_f
      jump 3, constant term
      P : 410 W_l
      P' : F, W_l    (π8)
      l = l + 1
    3)jump 4, F_f=400
      π5
    4)F_f = 500
      exit

E3 : ignore if doing single terms
      jump 1, F_f=400
      π3
    1)G3
      P : 111  1 \  these instructions set a link and then
      P : 590  0 /  exit to the selected subroutine.
```

```
        plant in last address,
        address of subroutine.
        jump 2/E2
```

The final jump instruction in this procedure is simply to avoid writing out the last part of E2 again. When we calculate a function of a constant expression we do not need to calculate a function' and this simplifies the procedure.

```
T1 : set constant term
     P1
     exit

T2 : ignore if doing multiple terms
     set +
     S1, U2
     set T = T' = 420
     exit

T3 : ignore if doing single terms
     set +
     S1
     store A    (π1)
     set variable term and no variable factors
     J3, W2
     jump 2/T4

T4 : ignore if doing single terms
     set +
     S1
     store A    (π1)
     set constant term
     I3
     set variable term and no variable factors
     P : 410 W1 }
     P' : 400 W1 }    (π7)
     l = l + 1  }
   1)X2
   2)sign and restore A    (π2)
     set T = T' = 420
     exit

T5 : ignore if doing single terms
     set +
     S1
```

```
      store A    (π1)
      set variable terms and no variable factors
      J3
      jump 1/T4

U1 :  jump 1, +
      jump 2, T' = 420
      jump 3, T = 420
      P : 400  0
    3)P' : 400  0
    2)set T = T' = 430
    1)set F = T, F' = T'
      V1
      P' : F', addr'    (π9)
      P : F, addr       (π10)
      exit
```

The procedures for the different elements of class V form the addresses of the variable and the variable' and plant them in addr and addr' respectively. In the case of V1 and V2, the function F' has a modifier added to it, since P' in calculating $\partial/\partial c_j$ (expression) makes use of the derivatives $\partial x/\partial c_j$ and $\partial y_i/\partial c_j$. The required value of j is set in a B-line before P' is obeyed.

```
J1 :  X2
      P : A = 1/A  }
      P : W = A    }    (Z)
      P : A = −A²  }
      set F = F' = Ff
      P : 410 Wl   }
      P' : F, Wl   }    (π8)
      l = l + 1    }
      P : A = W     (Z)
      exit

X1 :  set multiple terms
      Q1
      set single terms
      Q1
      exit

Q1 :  E1
      exit

Q2 :  ignore if doing multiple terms
      set F = F' = Ff
```

```
    V1
    jump 1, variable factors are present
            and if not set present
    P' : F', addr'      (π9)
    P : F, addr         (π10)
    set F_f = 500
    exit
  1)P' : F, addr        (π11)
    P' : 410 W_n'       (π4)
    n' = n' − 1
    P : 410 W_l
    P' : 400 W_l        (π7)
    l = l + 1
    P' : F', addr'      (π9)
    P : F, addr         (π10)
    set F = F' = 420
    n' = n' + 1  }      (π6)
    P' : F', W_n' }
    exit

Q3 : ignore if doing single terms
    jump 1, variable factors are present
            and if not set present
    jump 2, F_f = 500
    H2
    form and incorporate exponent, if any    (π13)
    set F_f = 500
    exit
  2)P : 410 W_n  }      (π3)
    n = n − 1    }
    P' : 410 W_n'  }    (π4)
    n' = n' − 1    }
    H2
    form and incorporate exponent, if any
  3)set F = F' = F_f
    n = n + 1    }      (π5)
    P : F, W_n   }
    n' = n' + 1  }      (π6)
    P' : F', W_n' }
    exit
  1)set F = F' = F_f
    P : 410 W_l + 1 }
    P' : F, W_l     }   (π8 + Z)
    l = l + 1       }
```

P' : 410 W_n'
n' = n' − 1
k = l − 1
l = l + 1
H2
form and incorporate exponent if any
4) P : 410 W_k
P : 500 W_{k+1}
P' : 500 W_{k+1}
set F = F' = 420
n' = n' + 1
P' : F', W_n'
exit

Q4 : omit if doing single terms
jump 1, variable factors are present
if not set present
jump 2, $F_f = 400$
P : 410 W_n
n = n − 1
P' : 410 W_n'
n' = n' − 1
2) set k = l, l = l + 1
H3
P : plant entry to subroutine for function of a variable ($F'(A) \rightarrow W_k$) ($\pi 14$)
P' : 500 W_k
jump 3/Q3, $F_f = 500$
set F_f = 500
exit
1) set F = F' = F_f
P : 410 W_{l+1}
P' : F, W_l ($\pi 8 + Z$)
l = l + 1
P' : 410 W_n'
n' = n' − 1
set k = l + 1, l = l + 2
H3, $\pi 14$
k = k − 2
jump 4/Q3

We now give the other π-routines which have not been explicitly defined.

```
π7 : P : 410 W1
     P' : 400 W1
     l = l + 1

π8 : P : 410 W1
     P' : F, W1
     l = l + 1

π9 : P' : F', addr'

π10 : P : F, addr

π11 : P' : F, addr

π12 : set j = p, i = n
   2)jump 1, j < 2
     P : 410 Wi
     P : 500 Wi
     i = i − 1
     j = ½j
     jump 2
   1)i = n + 1
   5)jump 3, p even
     p = ½p, i =ii − 1
     jump 4, p=0
     P : 500 Wi
     jump 5
   3)p = ½p
     jump 5
   4)exit

π13 : jump 1, p < 2
      plant p in list of numerical constants
      and its address in addr     (Z)
      p = p − 1
      jump 2, p < 2
      π12
   2)set F = F'= 500
      P : 410 W1
      P' : F, W1
      l = l + 1
      jump 3, p < 2
      P : 500 Wn
      jump 4
```

```
3)P : 500 W1 - 1
4)P' : F, addr
1)exit
```

$\pi 12$ replaces A in P by A^p where $p \geq 0$. $\pi 13$ also does this and in addition replaces A' in P' by $A' \times pA^{p-1}$. In these routines we take $p=0$ to be an indication that an exponent has been omitted, not that a zero exponent has been written. Thus we interpret A^0 as A.

Finally we show how the left hand sides of the equations are interpreted. These procedures effectively resolve the n^{th} order differential equations into sets of first order equations. In the Runge-Kutta integration routine that we use, the variables y_i are planted in locations x1 onwards and the f_i in x2 onwards. We call the difference, x2−x1, D.

```
L1 : set d = 0
     Y3
   1)d = d − 1
     set addr and addr' of yN^d (these
     will be addresses in the x1 locations)
     P : 410 addr + D
     P' : 412 addr' + D
     jump 2, d=0
     P : 400 addr
     P' : 402 addr'
     jump 1
   2)exit

L2 : set d = 0
     Y3
   1)d = d − 1
     set addr of aA^d
     P : 410 addr + D
     jump 2, d=0
     P : 400 addr
     jump 1
   2)exit

L3 : set addr and addr' of aN
     set F = F' = 410
     P' : F', addr'
     P : F, addr
     exit
```

```
Y1 : d = d + 1
     exit
```

CONCLUSION

These definitions give an idea of what is involved in a language of this nature and of what can be done in the way of analytic differentiation. The differentiation techniques could also be usefully applied to programs for solving non-linear equations and non-linear programming problems.

The program has proved very useful for solving equations, although experience has indicated how it may be improved when better programming techniques are available.

Solutions are found more easily when all the quantities involved are 0(1), not only the variables and the unknowns but also the elements of the matrix. It requires considerable ingenuity to scale the variables to achieve this, and if it could be done automatically it would be a great advance.

The other snag encountered was that many of the equations we tried to solve had one set of boundary conditions either at infinity or at a point where some of the factors become infinite. In these cases a suitable asymptotic expansion had to be found, if possible, to begin the solution. If, when this is required, it could be done by the program, we should have a system much nearer the ideal.

It is not possible to give a comprehensive user's description of the program here; instead we give two complete "programs" which are almost self-explanatory.

```
title
boundary layer equation
y1 → 3
equation
y1''' = −y1y1''
fit at x = 10
at x = 0
y1 = y1' = 0
y1'' = c1
at x = 10
y1 = 10 − c2
y1' = 1
y1'' = c3
dummy
```

```
print
x = 1,1   y1 = y1' = 1,8   at x = 1(1)9
and
c1 = c2 = c3 = 8
try
c1 = 0.4696   c2 = 1.22   c3 = 3,−6
end
title
plot periodic solutions
y1 → 2
dimension = 3
equation
y1'' = 10c3cos(πx)sin(πy1)
fit at x = 1
at x = 0
y1 = c1
y1' = c2
at x = 2
y1 = c1
y1' = c2
print
c1 = c2 = c3 = 2,6
try
c1 = 0.5   c2 = 0.5   c3 = 1
end
```

ACKNOWLEDGEMENT

The author would like to record his appreciation of the many helpful discussions he has had with Dr. C. B. Haselgrove throughout the course of this work.

REFERENCES

1. C. B. Haselgrove (1961). "The Solution of Non-Linear Equations and of Differential Equations with Two-Point Boundary Conditions", The Computer Journal, Vol. 4, p. 255.
2. "The Programmers' Handbook for the Ferranti Mercury Computer", Ferranti Publication CS 225 A.

GENERATING AN ANALOG COMPUTER WIRING DIAGRAM FROM THE DIFFERENTIAL EQUATIONS INPUT LANGUAGE

Walter Petry

Kernforschungsanlage
Jülich, Germany

In the following we shall discuss the problem of how to generate formally an analog computer wiring diagram from the differential equations input language for any system of ordinary differential equations. A formal construction of the wiring diagram from the system of differential equations can be performed by a data processing computer.*

The differential equations input language and the wiring diagram output language must be defined to investigate this problem.†

I. THE INPUT LANGUAGE AND THE SYSTEM OF DIFFERENTIAL EQUATIONS

1.1 The Elements of the Input Language

(1) The operators:
  (a) Connectives: +, −, ×, /.
  (b) Singulary operators: D (differential operator), D^{-1} (integral operator)**

*At the Institute of Applied Mathematics of the University of Mainz a program for the Z 22 was worked out, which constructs the corresponding analog computer wiring diagram for any system of ordinary differential equations.

†Compare this to the Thesis of the author based on a more general input language resp. a more general output language instead of the differential equations input language resp. the wiring diagram output lanlanguage.

**By the aid of D^{-1} it is possible to construct systems which are more general than differential equations; but in the following these systems will always be called systems of differential equations.

(2) The objects:
For semantic reasons the objects are divided into different classes.
(a) Class 1 (class of dependent variables): x, y, z.
(b) Class 2 (class of the independent variable): t.
(c) Class 3 (class of constants): a, b, c.
(d) Class 4 (class of functions formed by the use of a function generator): F[x], F[y], F[z], F[t].
(3) The parentheses: (,).
(4) The sign of congruence: =.

1.2 The System of Differential Equations

Definition: If A and B are objects, then the following chains A, (A), AD, AD^{-1}, $A \pm B$, $A \times B$ are expressions. If α and β are expressions, then the chains (α), αD, αD^{-1}, $\alpha \pm \beta$, $\alpha \times \beta$ are expressions, too. If α and β are expressions, then the chain $\alpha = \beta$ is a differential equation. Several differential equations form a system of differential equations.

II. THE WIRING DIAGRAM OUTPUT LANGUAGE

2.1 The Elements of the Wiring Diagram Output Language

(1) S: summer
(2) N: sign inverter (inverter of +)
(3) M: multiplier
(4) V: special amplifier (inverter of $\times$)
(5) P: potentiometer
(6) I: integrator

2.2 Correspondence of the Elements of the Input Language to the Elements of the Wiring Diagram Output Language

A wiring diagram element can be made correspondent to each object of classes 2-4, but not to an object of class 1.

2.3 Construction of the Wiring Diagram Element for any Expression

If α and β are expressions of which the elements in the output language are known or which are objects of class 1, then the corresponding wiring diagram element can be constructed for each expression of the form

$\alpha \pm \beta$, $\alpha \times \beta$, αD^{-1},

but not for an expression of the form αD.*

2.4 The Wiring Diagram Elements for Objects of Class 1

If α is an expression, for which the wiring diagram element is constructed, then the wiring diagram element of an object of class 1, defined by a differential equation of the form $y = \alpha$, is identical to the wiring diagram element of α.

REMARK: The usual analog computer wiring diagram will be received by connecting the corresponding in- and outputs of the wiring diagram elements.

III. SPECIAL SYSTEMS OF DIFFERENTIAL EQUATIONS

THEOREM: If α^i $(i = 1,2,\ldots,k)$ is an expression which is not identical to an object of class 1 and which contains at most the k objects $y^1, y^2, \ldots, y^k$ of class 1 and not the operator D, then the analog computer wiring diagram can be constructed for each system of differential equations of the form

(*) $y^i = \alpha^i \quad (i = 1,2,\ldots,k)$

Proof: At first the wiring diagram elements for the expressions enclosed in the inmost parentheses of α^i $(i = 1,2,\ldots,k)$ are successively constructed by observing the rules of precedence for the operators. Then the surrounding parentheses will be regarded and the process will be repeated until the wiring diagram element for α^i $(i = 1,2,\ldots,k)$ is constructed.

The wiring diagram element of y^i $(i = 1,2,\ldots,k)$ is identical with that of α^i according to 2.4.

IV. GENERAL SYSTEMS OF DIFFERENTIAL EQUATIONS

Now systems of differential equations of the form

(**) $\alpha^i = \beta^i \quad (i = 1,2,\ldots,k)$

which contain exactly the k objects $y^1, y^2, \ldots, y^k$ of class 1 will be regarded.

---

*There exists no wiring diagram element for the operator D because the corresponding element is unstable.

4.1 The Transformation Rules for Systems of Differential Equations

4.11 Algebraic Rules.* If $\mathfrak{S}$ is identified with + resp. $\times$ and $\mathfrak{S}^{-1}$ with the inverse operator, i.e., - resp. /, then the following rules can be stated for any expressions α, β and γ:

(1) $\alpha \mathfrak{S} \beta = \gamma \overset{+}{\longleftrightarrow} \alpha = \gamma \mathfrak{S}^{-1} \beta$ resp. $\beta = \gamma \mathfrak{S}^{-1} \alpha$

(2) $\alpha \mathfrak{S}^{-1} \beta = \gamma \longleftrightarrow \alpha = \gamma \mathfrak{S} \beta$ resp. $\beta = \alpha \mathfrak{S}^{-1} \gamma$

4.12 Transformation Rules for the Singular Operators. If α and β are expressions, then the rule

$$\alpha D = \beta \longleftrightarrow \alpha = \beta D^{-1}$$

is valid.

4.13 Introduction of New Objects of Class 1. If α is an expression, then the new object u of class 1 can be introduced by the aid of the differential equation

$$u = \alpha$$

4.14 The Substitution Process. If α, β, γ and δ are expressions and for instance the two differential equations

$$\alpha \mathfrak{S} \beta D = \gamma \quad \text{and} \quad \beta = \delta$$

hold, then the differential equation $\alpha \mathfrak{S} \delta D = \gamma$ is also valid.

4.2 The Wiring Diagram Elements of Objects of Class 1, defined by a System of Differential Equations of the Form (**)

Definition: The analog computer wiring diagram of a system of differential equations of the form (**) is identical to the analog computer wiring diagram of a system of the form (*) received from the system (**) by the aid of the transformation rules according to 4.1.

4.3 A Class of Systems of the Form (**) which can be transformed by the above Rules into a System of the Form (*)

---

*The law of commutativity holds.

+The sign $\longleftrightarrow$ indicates the equivalence of the two differential equations.

4.31 Specification of the Class of Systems. A system of differential equations of the form

$$\alpha^i = \beta^i \quad (i = 1,2,\ldots,k)$$

containing the k objects $y^1, y^2, \ldots, y^k$ of class 1 is called an admissible system, if there exists an object y^{μ_i} ($\mu_i \in \{1,2,\ldots,k\}$) of class 1 for each differential equation $\alpha^i = \beta^i$ ($i = 1,2,\ldots,k$), and conversely with the following conditions:

(1) In this differential equation $\alpha^i = \beta^i$, y^{μ_i} is included in an expression of the form $y^{\mu_i}\underbrace{DD\ldots D}_{\bar{1}\mu_i}$. In this system there exists no differential equation containing an expression of the form $y^{\mu_i}\underbrace{DD\ldots D}_{1\mu_i}$ with $1\mu_i > \bar{1}\mu_i$.

(2) The operator D works on objects of class 1, and with respect to the differential equation $\alpha^i = \beta^i$ on such parenthesis expressions which enclose the corresponding expression $y^{\mu_i}\underbrace{DD\ldots D}_{\bar{1}\mu_i}$. However, the operator D^{-1} cannot work on such parenthesis expressions.

REMARK: There are systems of differential equations for which no, one or more possibilities exist to relate the k objects of class 1 to the various differential equations of the system, so that the two conditions can be fulfilled.

4.32 Construction of a System of the Form (*) from an Admissible System of Differential Equations of the Form (**) fulfilling the Conditions 4.31. If $\alpha^i = \beta^i$ ($i = 1,2,\ldots,k$) is the system of differential equations, then the new objects $u_1^{(\mu_i)}, u_2^{(\mu_i)}, \ldots, u_{\bar{1}\mu_i}^{(\mu_i)}$ ($\mu_i = 1,2,\ldots,k$) of class 1 will be introduced by:

$$\left.\begin{array}{ll} \text{(a)} & u_1^{(\mu_i)} = y^{\mu_i} D \\ & u_2^{(\mu_i)} = y^{\mu_i} DD \\ & u_3^{(\mu_i)} = y^{\mu_i} DDD \\ & \vdots \\ & u_{\bar{1}\mu_i}^{(\mu_i)} = y^{\mu_i} DD\ldots D \end{array}\right\} \longleftrightarrow \left\{\begin{array}{ll} \text{(b)} & y^{\mu_i} = u_1^{(\mu_i)} D^{-1} \\ & u_1^{(\mu_i)} = u_2^{(\mu_i)} D^{-1} \\ & u_2^{(\mu_i)} = u_3^{(\mu_i)} D^{-1} \\ & \vdots \\ & u_{\bar{1}\mu_i - 1}^{(\mu_i)} = u_{\bar{1}\mu_i}^{(\mu_i)} D^{-1} \end{array}\right.$$

When substituting all expressions of the form $y^{\mu_i}\underbrace{DD\ldots D}_{l\mu_i}$ $(1 \leq l\mu_i \leq \bar{l}\mu_i)$ in the system $\alpha^i = \beta^i$ $(i = 1,2,\ldots,k)$ by the introduced objects of class 1 and when adding system (b), the following system of differential equations will be received:

$$(+)\quad \begin{cases} \tilde{\alpha}^i = \tilde{\beta}^i \ * & (i = 1,2,\ldots,k) \\ y^{\mu_i} = u_1^{(\mu_i)} D^{-1} & (\mu_i \in \{1,2,\ldots,k\}) \\ u_1^{(\mu_i)} = u_2^{(\mu_i)} D^{-1} & \\ u_2^{(\mu_i)} = u_3^{(\mu_i)} D^{-1} & \\ \vdots & \\ u_{\bar{l}\mu_i - 1}^{(\mu_i)} = u_{\bar{l}\mu_i}^{(\mu_i)} D^{-1} & \end{cases}$$

This system contains the objects y^{μ_i}, $u_1^{(\mu_i)}$, $u_2^{(\mu_i)}$, $\ldots$, $u_{\bar{l}\mu_i}^{(\mu_i)}$ $(\mu_i = 1,2,\ldots,k)$ of class 1.

This system of differential equations (+) can be transformed by the aid of the transformation rules of 4.1 into a system of the form

$$(++)\quad \begin{cases} u_{\bar{l}\mu_i}^{(\mu_i)} = \beta_i^* & (i = 1,2,\ldots,k;\ \mu_i \in \{1,2,\ldots,k\}) \\ y^{\mu_i} = u_1^{(\mu_i)} D^{-1} & \\ u_1^{(\mu_i)} = u_2^{(\mu_i)} D^{-1} & \\ \vdots & \\ u_{\bar{l}\mu_i - 1}^{(\mu_i)} = u_{\bar{l}\mu_i}^{(\mu_i)} D^{-1} & \end{cases}$$

where y^{μ_i} is the corresponding object of class 1 of the differential equation $\alpha^i = \beta^i$.

---

*The operator D does not work in the expressions $\tilde{\alpha}^i$ and $\tilde{\beta}^i$ on objects of class 1, but only on parenthesis expressions defined by the condition (2) of 4.31.

Proof: The differential equation $\alpha^i = \beta^i$ $(i = 1,2,\ldots,k)$ can be transformed by the rules of 4.11 and the rule of 4.12 into an equation of the form $u\frac{(\mu_i)}{l\mu_i} = \beta_i^*$ $(\mu_i = 1,2,\ldots,k)$ whereas β_i^* does not contain the operator D but only D^{-1} on account of the condition (2) of 4.31.

REMARK: The system (++) is a system of the form (*). Therefore the analog computer wiring diagram for the system (++) can be constructed by the procedure of III. The analog computer wiring diagram of system (**) of differential equations is identical to the analog computer wiring diagram of the system (++).

PREMIERS ELEMENTS D'UN LANGAGE DE PROGRAMMATION POUR LE TRAITEMENT EN ORDINATEUR DES GRAPHES.

par R. Tabory

IBM-France,
Paris, France

INTRODUCTION

Notre but etait de creer les premiers elements d'un langage de programmation pour le traitement facile en ordinateur des graphes, en utilisant des techniques de "list processing" au sens de Newell-Shaw-Simon (Rand Corp.) et de Gelernter (IBM-Research) dans le programme resultant (apres compilation).

Techniquement, nous avons procede par simulation, autrement dit en evitant d'ecrire un veritable compilateur. Les diverses instructions de notre langage sont representees par des sous-programmes ou fonction FORTRAN, les instructions de ces sous-programmes et fonctions etant elles-memes egalement en FORTRAN et FLPL. Les listes NSS sont automatiquement generees, sans que l'utilisateur du "graph processor" connaisse leur existence. Tout se passe donc comme si apres compilation on avait une "list-processing" machine a sa disposition, l'utilisateur travaillant en langage-probleme, les instructions langage-machine etant les instructions FLPL et FORTRAN.

Nous avons essaye de definir les instructions de notre langage d'une facon "parlante" pour un utilisateur des graphes, c'est-a-dire en nous preoccupant plutot de la semantique de notre langage que de sa syntaxe. (De toutes facons, nous n'ecrivons pas un veritable compilateur).

Pour pouvoir creer un langage veritablement oriente vers les problemes d'un certain type et non pas vers la machine, nous avons eu recours a l'ouvrage de M. Claude Berge: Theorie des graphes et ses applications (Dunod, Paris 1958), en y puisant la

terminologie et quelques algorithmes pour les exercices choisis, exception faite du terme "sommet", au lieu duquel nous disons "nœud". Nous avons egalement introduit le terme "semanteme", equivalent du terme americain "data". Les semantemes sont rattachables aux nœuds; dans ce cas leur nombre est arbitraire et chacun d'eux represente une quantite d'informations de longueur egale a 36 bits et de nature quelconque. On peut egalement attacher a chaque arc un seul semanteme de longueur egale a 18 bits. Les nœuds, les arcs et les semantemes peuvent etre repartis en 7 classes. Des instructions du type "boucle" permettent d'explorer les semantemes et les arcs partant d'un nœud.

Les diverses instructions du langage ont ete choisies a partir des elements et des notions les plus simples de la theorie des graphes, sans penser a un probleme particulier quelconque a traiter eventuellement sur machine. Les applications presentees ont comme but de tirer les conclusions d'une confrontation entre ce langage concu a priori et les problemes effectivement traites comme exercices.

Dans le chapitre suivant nous decrivons le langage et ensuite les applications traitees sur 7090 apparaissent.

LES INSTRUCTIONS DU LANGAGE

I. FONCTIONS DE CONSTRUCTION DE GRAPHES

(a) APPEL GRAPHE

Ce sous-programme rend la memoire libre de FORTRAN adaptable au traitement des graphes. Son appel doit preceder l'emploi de toutes les autres fonctions.

Restriction: la premiere memoire de la region COMMUN doit etre reservee dans le programme principal et on ne doit manipuler son contenu.

Remarque: un deuxieme appel de GRAPHE efface tous les graphes construits et rend la memoire libre de FORTRAN de nouveau adaptee au traitement des graphes.

(b) I = NOEUD (J)

Cette fonction construit l'ossature d'un nœud que d'autres sous-programmes permettront d'enrichir par des arcs et des semantemes. Les nœuds peuvent etre repartis par l'utilisateur en 7 classes distinctes, numerotees de 1 a 7. L'argument J est egal

au numero de la classe designee. La valeur I de la fonction est l'identificatif attribue par FORTRAN au noeud.

(c) APPEL ARC (I2, I2, J, k)

ARC etablit entre les nœuds I1 et I2 l'arc I1→ I2. Les arcs peuvent appartenir a 7 classes distinctes designees par l'utilisateur et numerotees de 1 a 7. L'argument J exprime ce numero. L'argument k designe un semanteme de longueur maximale egale a 15 bits (appele "lien") et attache a l'arc etabli.

(d) APPEL SEMANT (I, D, J)

Le semanteme D est attache au nœud I. Les semantemes d'un nœud peuvent appartenir a 7 classes designees par l'utilisateur et numerotees de 1 a 7. L'argument J exprime ce numero.

II. FONCTIONS DE MODIFICATION DE GRAPHES

(a) I = MODIFN (J, N)

La fonction "modification nœud" attribue la nouvelle classe N au nœud J. La valeur de la fonction I est egale au numero de l'ancienne classe.

(b) I = MODIFS (J, D, N)

La fonction "modification semanteme" attribue la nouvelle classe N au semanteme D du nœud J. La valeur I de la fonction est egale au numero de l'ancienne classe. I = 0 si le semanteme D n'est pas present au-nœud J.

(c) I = MODIFA (I1, I2, J)

La fonction "modification arc" attribue la nouvelle classe J a l'arc I1→ I2. La valeur I de la fonction est egale a l'ancienne classe. I = 0 en cas de non-existence de l'arc.

III. FONCTIONS D'EFFACEMENT

(a) APPEL EFFACE (I, J, K)

1° cas: J et k sont nuls. Le nœud I est alors efface, avec ses arcs partants et ses semantemes. I est rendu egal a 0.

2° cas: J ≠ 0 et k = 0. L'arc I→ J est efface.

3° cas: k ≠ 0. J est considere comme semanteme (dans ce cas

ce n'est pas necessairement un symbole fixe) et il est efface dans le nœud I.

Dans les cas 2 et 3 rien ne se passe en cas de non-existence.

IV. FONCTIONS DE CONTROLE DE NATURE DES GRAPHES

(a) I = NATURE (I1, I2, k)

1° cas: I2 et k sont nuls. I est egale au numero de categorie du nœud I1.

2° cas: I2 ≠ 0 et k = 0. I est egal au numero de categorie de l'arc I1→I2. I = 0 en cas de non existence de l'arc.

3° cas: I est egal au numero de categorie du semanteme I2 du nœud I1. I = 0 en cas de non-existence de I2 dans I1. (k = 0)

(b) I = LIEN (I1, I2, J, k)

Si k = 0 J est ignore et I est egal au lien (semanteme de 15 bits) de l'arc I1→I2.

Si k ≠ 0, le lien I1 → I2 est remplace par J. I est egal a l'ancien lien.

I = 0 en cas de non-existence de l'arc I1 → I2.

V. FONCTIONS D'EXPLORATION

(a) Remarque preliminaire

"l'ordre" des arcs partants et des semantemes d'un nœud est l'ordre chronologique en lequel l'utilisateur a enrichi le nœud par ces arcs et semantemes.

(b) APPEL EXPLOR (I, N, A, k)

Le sous-programme explore, dans l'ordre, les semantemes de classe N du nœud I. Si N = 0 tous les semantemes sont explores. Aux appels consecutifs d'EXPLOR A est rendu egal au semanteme qui suit dans l'ordre. (A n'est pas necessairement un symbole flottant). Lorsque le dernier semanteme de la classe demandee a ete explore, k est rendu egal a zero. Il est negatif lors d'inexistence d'un semanteme de la classe demandee dans le nœud I,—positif dans le restant des cas.

Voici un exemple de traitement:

1° APPEL EXPLOR (I, N, A, k)
traitement de A

si (k) 10, 20, 10
2° instruction suivante

Il est possible d'explorer simultanement le meme nœud avec plusieurs valeurs de N, a condition qu'aucune de ces valeurs ne soit zero. Exemple

1° APPEL EXPLOR (I, 1, A k1)
traitement de A
APPEL EXPLOR (I, 2, B, k2)
traitement de B
si (k1 * k2) 10, 20, 10
2° instruction suivante

Un nouvel APPEL EXPLOR, apres un retour avec k = 0, recommence l'exploration des le debut. Lorsqu'on desire interrompre une exploration (apres un retour avec $k \neq 0$) le sous-programme ABANDN (voir plus bas) permet de re-initialiser le nœud pour une nouvelle serie d'appels d'EXPLOR.

(c) APPEL VOISIN (I, N, J, k)

Le sous-programme explore, dans l'ordre, les nœuds voisins J du nœud I, tels que les arcs $I \rightarrow J$ de classe N existent. Si N = 0, tous les nœuds voisins sont explores. Aux appels consecutifs de VOISIN J est rendu egal au nœud voisin suivant dans l'ordre. Lorsque le dernier nœud voisin de la classe demandee a ete explore, k est egal a zero. L'exploration simultanee avec plusieurs valeurs de N est possible, a condition qu'aucune de ces valeurs ne soit zero. Un APPEL VOISIN apres un retour avec k = 0, recommence l'exploration des le debut. $k < 0$ en cas de non-existence.

(d) APPEL ABANDN (I, N, M)

1° cas: M = 0. L'exploration des semantemes de classe N du nœud I est interrompue. Le nœud est re-initialise pour pouvoir recommencer cette exploration.

2° cas: $M \neq 0$. L'exploration des arcs de classe N du nœud I est interrompue. Le nœud est re-initialise pour pouvoir recommencer cette exploration.

Remarque: on programme ABANDN lorsqu'on desire abandonner une exploration (du type EXPLOR ou VOISIN) apres un retour avec $k \neq 0$.

VI. FONCTIONS DE CHEMINEMENT

(a) APPEL CHEMIN (I, J, k)

1° cas: $J = 0$. L'ossature du chemin I de classe k ($1 \leq k \leq 7$) est creee,—l'ossature etant enrichie, par la suite, par des etapes.

2° cas: $J \neq 0$. Le nœud J est ajoute, comme etape suivante, au chemin I. Le sous-programme rend $k = 0$, si le nœud J figurait deja comme etape, sur le chemin I (cas de circuit ferme). $k > 0$ dans le cas normal.

On observera que k est argument ou resultat suivant qu'on est dans le cas 1 ou 2, ainsi que I.

Pour connaitre
la classe d'un chemin I, on programme M = NATURE (I, 0, 0)

Pour effacer completement
le chemin I, on programme APPEL EFFACE (I, 0, 0)

Pour modifier
la classe d'un chemin I, on programme J = MODIFN (I, N)

voir cans les chapitres precedents la signification de ces fonctions.

(b) I = IENLEV (J, k)

1° cas: $k \neq 0$. La k-ieme etape est enlevee du chemin J. I est egal au nœud enleve. $I = 0$ si k est superieur au nombre d'etapes du chemin.

2° cas: $k = 0$. La derniere etape du chemin J est enlevee du chemin. I est egal au nœud enleve ou a zero si le chemin n'a pas d'etapes.

(c) APPEL ETAPE (I, J, k)

ETAPE est un sous-programme du type EXPLOR ou VOISIN. Aux appels successifs J est rendu egal aux nœuds formant les etapes du chemin I.

Au retour, $k = 0$ lorsque la derniere etape a ete exploree.
$k = -1$ en cas de circuit (nœud deja rencontre).
$k = 1$ derniere etape et circuit.
$k = 2$ dans les autres cas.

Apres le retour avec $k = 0$ le chemin est re-initialise pour une nouvelle exploration des etapes.

On a le resultat $J = 0$ lorsque le chemin n'a pas d'etapes.

APPLICATIONS 7090

1er exercice: Algorithme pour la determination d'un ensemble exterieurement stable minimum.

Etant donne un graphe, on dit qu'un ensemble de nœuds de ce graphe est stable exterieurement, si pour tout nœud n'appartenant pas a l'ensemble il existe un arc partant de ce nœud vers l'un des nœuds de l'ensemble.

Nous avons employe l'algorithme propose dans l'ouvrage de M. Claude Berge (Ref. 1, pages 42-43) pour trouver l'ensemble exterieurement stable minimum du graphe traite comme exemple par M. Berge.

Ci-dessous on trouvera le programme compose d'environ une centaine d'instructions "graph processor" et FORTRAN, ensuite le resultat fourni par la 7090 qui contient d'ailleurs la description du graphe traite.

```
RECHERCHE PARTIE EXT STABLE MIN      0/70/36

    DIMENSION NS (36), N(100,2)
    COMMUN LAVST, NS,N,MEX
    APPEL GRAPHE
    INSCR SORTIE BANDE 6, 104
104 MODELE (1H1)
    FAIRE 1 I = 1,100
    FAIRE 1 J = 1,2
  1 N(I,J) = 0
    FAIRE 2 I=1,100
    LIRE ENTREE BANDE 5,100,(NS(I1),I1=1,36)
100 MODELE (36I2)
    SI (NS(1)) 3,4,3
  3 INSCR SORTIE BANDE 6,101,NS(1),(NS(I), I=2,36)
101 MODELE (7HONOEUD=I2,2X,4HARCS35(1X,I2))
    FAIRE 5 J=1,36
    I1=NS(J)
    SI(I1)6,2,6
  6 SI(N(I1,1)) 8,7,8
  7 N(I1,1)=NOEUD(3)
    N(I1,2)=NOEUD(3)
    APPEL ARC(N(I1,1),N(I1,2),0,0)
    APPEL ARC(N(I1,2),N(I1,1),0,0)
  8 SI(J-1) 9,5,9
  9 APPEL ARC(N(I1,1),N(I,2),0,0)
    APPEL ARC(N(I,2),N(I1,1),0,0)
  5 CONTINUER
  2 CONTINUER
  4 MEX=NOEUD(0)
```

```
 12 APPEL REDUC
    FAIRE 10 I=1,100
    SI(2-NATURE(N(I,1),0,0))I1,10,11
 11 SI(N(I,1))12,13,12
 10 CONTINUER
 13 I=0
 14 I=I+1
    APPEL VOISIN(MEX,0,NS(I),K)
    FAIRE 61 J=1,100
    SI(NS(I)-N(J,1)) 61,62,61
 62 NS(I)=J
    ALLER A 63
 61 CONTINUER
 63 SI(K) 14,15,14
 15 INSCR SORTIE BANDE 6,102
102 MODELE(38HOPARTIE MINIMALE EXTERIEUREMENT STABLE)
    INSCR SORTIE BANDE 6,103,(NS(J),J=1,I)
103 MODELE(36I3)
    APPEL EXIT
    FIN(1,1,0,0,0,0,0,1,0,0,0,0,0,0,0)
**  LISTE 8
    SOUS PROGRAMME REDUC
    DIMENSION N(100,2),NS(36)
    COMMUN LAVST,NS,N,MEX
    FAIRE 10 I=1,100
    SI(N(I,1))11,100,11
 11 SI(NATURE(N(I,1),0,0)-2)110,10,110
110 APPEL VOISIN(N(I,1),0,J,K)
    SI(NATURE(J,0,0)-2)111,112,111
111 J1=MODIFN(J,1)
112 SI(K)110,12,110
 12 FAIRE 20 I1=1,100
    SI(N(I1,1)) 21,30,21
 21 SI(I1-I)220,20,220
220 SI(NATURE(N(I1,1),0,0)-2)22,20,22
 22 APPEL VOISIN(N(I1,1),0,I2,K1)
    SI(NATURE(I2,0,0)-2)24,24,23
 24 SI(K1)22,25,22
 23 APPEL ABANDN(N(I1,1),0,1)
    ALLER A 20
 25 J1=MODIFN(N(I1,1),2)
 20 CONTINUER
 30 APPEL VOISIN(N(I,1),0,J,K)
    SI(NATURE(J,0,0)-2) 300,301,300
300 J1=MODIFN(J,3)
301 SI(K) 30,10,30
 10 CONTINUER
100 IL=0
    FAIRE 40 I=1,100
    SI(N(I,2)) 41,210,41
```

```
41    SI(NATURE(N(I,2),0,0)−2) 410,40,410
410   IK=0
413   APPEL VOISIN(N(I,2),0,J,K)
      SI(NATURE(J,0,0)−2) 411,412,411
411   IK=IK+1
      J1=J
412   SI(K) 413,414,413
414   SI(IK−1) 40,416,40
416   J=MODIFN(J1,2)
      IL=IL+1
      APPEL ARC(MEX,J1,0,0)
417   APPEL VOISIN(J1,0,J,K)
      IK=MODIFN(J,2)
      SI(K) 417,40,417
40    CONTINUER
210   SI(IL) 200,50,200
 50   FAIRE 51 I=1,100
      SI(NATURE(N(I,1),0,0)−2)52,51,52
52    J1=MODIFN(N(I,1),2)
      APPEL ARC(MEX,N(I,1),0,0)
 53   APPEL VOISIN(N(I,1),0,J1,K)
      I1=MODIFN(J1,2)
      SI(K)53,200,53
 51   CONTINUER
200   RETOUR
      FIN
```

2eme exercice: trouver un chemin de longueur minimale entre deux nœuds d'un graphe.

L'algorithme employe etait celui propose par M. Claude Berge (ref. 1, page 66), l'exemple etait le labyrinthe (page 64, figure 7-1a)

Sur les pages suivantes, on trouvera le programme et les resultats fournis par l'ordinateur 7090. Les resultats commencent par la description du probleme: au labyrinthe correspond un graphe dont les nœuds sont numerotes de 1 a 22. Pour chaque nœud on indique les arcs qui partent de ce nœud vers les nœuds dont les numeros sont imprimes. Ensuite, la machine calcule la longueur du plus court chemin entre les nœuds 1 et 2, comptee en arcs et les etapes de ce chemin.

```
NOEUD= 1  ARCS  -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD= 2  ARCS   1 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD= 3  ARCS   1  5 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD= 4  ARCS   2  5 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD= 5  ARCS   2 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD= 6  ARCS   2  7 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD= 7  ARCS   5 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
PARTIE MINIMALE EXTERIEUREMENT STABLE
 1  2  5
```

```
CHEMIN DE LONGUEUR MINIMALE          16/03/62

      DIMENSION N(22),NS(36)
      APPEL GRAPHE
      FAIRE 10 I=1,22
10    N(I)=0
      INSCR SORTIE BANDE 6,101
101   MODELE(1H1)
      FAIRE 1 I=1,22
      LIRE ENTREE BANDE 5,100,(NS(J),J=1,36)
100   MODELE(36I2)
      INSCR SORTIE BANDE 6,102,NS(1),(NS(J),J=2,36
102   MODELE(7HONOEUD=I2,2X,4HARCS35(1X,I2))
      FAIRE 2 J=1,36
      SI(NS(J)3,1,3
3     I1=NS(J)
      SI(N(I1)) 4,5,4
5     N(I1)=NOEUD(0)
4     SI(J-1) 6,2,6
6     APPEL ARC(N(I1),N(I),0,0)
2     CONTINUER
1     CONTINUER
      MEX=NOEUD(0)
      I1=MODIFN(N(1),1)
      APPEL SEMANT(N(1),0,0)
7     APPEL VOISIN(N(1),0,J,K)
      APPEL SEMANT(J,1,0)
      I1=MODIFN(J,1)
      SI(K)7,8,7
   8  FAIRE 20 I=2,22
      SI(NATURE(N(I),0,0))21,20,21
  21  APPEL EXPLOR(N(I),0,I1,K)
      I1=I1+1
  24  APPEL VOISIN(N(I),0,J,K)
      SI(NATURE(J,0,0))22,23,22
  23  APPEL SEMANT(J,I1,0)
      I2=MODIFN(J,1)
22    SI(K) 24,20,24
  20  CONTINUER
      SI(NATURE(N(2),0,0))30,8,30
  30  APPEL EXPLOR(N(2),0,J,K)
      INSCR SORTIE BANDE 6,110,J
 110  MODELE(32HOLONGUEUR MINIMALE ENTRE 1 ET 2=I2)
      L1=N(2)
      I1=1
  32  APPEL VOISIN(L1,0,L2,K)
      SI(NATURE(L2,0,0))31,32,31
  31  APPEL EXPLOR(L2,0,J1,K)
      SI(J-J1-I1) 32,33,32
```

```
 33 APPEL ARC(MEX,L2,0,0)
    L1=L2
    I1=I1+1
    SI(L2-N(1))32,34,32
 34 I=0
 65 I=I+1
    I1=J-I+1
    APPEL VOISIN(MEX,0,NS(I1),K)
    FAIRE 61 J1=1,22
    SI(NS(I1)-N(J1))61,62,61
 62 NS(I1)=J1
    ALLER A 63
 61 CONTINUER
 63 SI(K)65,66,65
 66 NS(J+1)=2
    INSCR SORTIE BANDE 6,120
120 MODELE(18HOPLUS COURT CHEMIN)
    I1=J+1
    INSCR SORTIE BANDE 6,121,(NS(I),I=1,I1)
121 MODELE(36I3)
    APPEL DUMP
    FIN(1,1,0,0,0,0,0,1,0,0,0,0,0,0,0)
```

```
NOEUD= 1  ARCS  22 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD= 2  ARCS  13 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD= 3  ARCS  22 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD= 4  ARCS   5  6  8 22 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD= 5  ARCS   4 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD= 6  ARCS   4  7 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD= 7  ARCS   6 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD= 8  ARCS   4 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD= 9  ARCS  22 10 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD=10  ARCS   9 11 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD=11  ARCS  10 12 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD=12  ARCS  13 14 11 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD=13  ARCS  12 19 20  2 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD=14  ARCS  12 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD=15  ARCS  12 16 17 18 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD=16  ARCS  15 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD=17  ARCS  15 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD=18  ARCS  15 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD=19  ARCS  13 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD=20  ARCS  13 21 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD=21  ARCS  20 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
NOEUD=22  ARCS   1  3  9  4 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0 -0
LONGUEUR MINIMALE ENTRE 1 ET 2= 7
PLUS COURT CHEMIN
  1 22  9 10 11 12 13  2
```

REFERENCES

1. Claude Berge, Theorie des graphes et ses applications, Dunod, Paris, 1958.
2. D. Koenig, Theorie der endlichen und unendlichen Graphen.
3. Gelernter-Hansen-Gerberich, A. FORTRAN-compiled List Processing language (Journal of Assoc. Comp. Mach., 7 (1960), p. 87-101).
4. Hansen, The Use of the FORTRAN-Compiled List Processing language (RC-282, IBM Research Center, Yorktown Heights, N. Y.)

UTILISATION DE F.L.P.L. DANS LA RESOLUTION D'UN PROBLEME D'ORDONNANCEMENT

par Pierre Darnaut et Gerard Sandier

Sema, Paris, France

INTRODUCTION

Le problème d'ordonnancement qui va être exposé, rentre dans une classe plus générale de problèmes qui a été étudiée par M. Bernard ROY (1 et 2). La méthode de résolution qui est décrite ci-dessous, est l'application de l'algorithme général à ce cas particulier. L'algorithme a donc subi quelques simplifications, et a été mis en oeuvre sur ordinateur IBM 7090. Le problème a été formulé dans le langage de la théorie des graphes, et on verra que l'une des opérations essentielles pour la recherche de la solution est le cheminement dans un graphe suivant des règles définies par l'algorithme. Il a paru intéressant, pour programmer ce cheminement, d'utiliser les instructions du Fortran-Compiled List-Processing Language qui ont été établies justement pour traiter des structures de ce type. Les opérations F.L.P.L. utilisées sont principalement celles qui permettent de creér des listes et de stocker ou rechercher des informations dans ces listes.

ENONCE DU PROBLEME ET CARACTERISTIQUES DE LA SOLUTION

L'ordonnancement traité concerne la construction d'un grand ensemble industriel,* en l'occurrence une centrale électrique pour Electricite de France. Cette construction est subdivisée en tâches élémentaires, dont chacune a une durée donnée.

L'ordre de succession de ces taches est soumis à des contraintes que l'on peut exprimer sous deux formes équivalentes:

—soit comme contraintes d'antériorité, une tâche ne pouvant commencer avant que certaines autres, dites précédentes, aient été terminées;

—soit comme contraintes de postériorité, une tâche devant être terminée avant que certaines autres, dites suivantes, puissent commencer.

On cherche à déterminer pour chaque tâche des dates de début et de fin compatibles avec les contraintes, et dont l'ensemble soit tel que la durée totale de la construction soit minimale. Les durées des tâches étant données, l'une des deux dates pour chaque tâche suffit à definir sans ambiguité la solution, et la durée de construction est égale à la plus grande des dates de fin trouvées, si l'on prend arbitrairement zéro comme date initiale de référence.

La solution existe à condition que l'ensemble des contraintes données ne contienne pas de circuit (suite de tâches liées dont la première devrait être suivante de la dernière). Cette condition, qui est évidemment nécessaire, est suffisante. Si elle est satisfaite, il existe un ensemble d'ordonnancements qui conduisent tous à la durée totale de construction minimale. La date de debut de chaque tâche se trouve dans un certain intervalle de temps pour tous ces ordonnancements. Nous appellerons "calage à gauche" l'ordonnancement solution dans lequel chaque date de début est prise égale à son minimum, et "calage à droite" celui ou chacune est égale à son maximum. Pour une suite de tâches constituant ce que nous appellerons la "chaine determinante", l'intervalle a une longueur nulle, et leur date de début est donc impérative et indépendante de l'ordonnancement solution choisi. C'est la suite des tâches constituant cette chaine déterminante qui fixe la duree totale de la construction.

ALGORITHME DE RESOLUTION

A ce problème correspond un graphe dont les sommets représentent les taches. Les arcs joignent les sommets—tâches—liés par une contrainte et sont orientés de la tâche précédente vers la tâche suivante. A chaque arc est affectée une longueur égale à la durée de la tâche représentée par son extrêmité initiale. Les données sont donc constituées par ce graphe (sommets et arcs)

et les longueurs. On cherche à affecter à chaque sommet i une date (de début) d_i telle que, pour tout couple de sommets (j, k) reliés par un arc de longueur t_j (cette longueur est en effet indépendante de k), on ait $d_k - d_j \geq t_j$ et telle que le maximum de $(d_i + t_i)$(qui est la date de fin la plus reculée) soit minimal.

A chaque application de l'algorithme pour le calage à gauche, on marque un sommet i dont tous les précédents j sont marqués (et dont les dates de début ont déja été déterminées), et on lui affecte la date de debut $d_i = \text{Max}_j\ (d_j + t_j)$. Au départ on fixe à zéro la date de debut des sommets—tâches—qui n'ont pas de précédents (il y en a obligatoirement au moins un si le graphe est sans circuit). La détermination de l'ordonnancement "calage à gauche" est terminée lorsque les dates de début de toutes les tâches ont été fixées, et l'on obtient la durée de construction minimale $F = \text{Max}\ (d_i + t_i)$.

Le calage à droite reviendrait à faire les mêmes opérations sur le même graphe où l'on aurait simplement inversé les sens des arcs. Si l'on utilise le graphe initial, on fixe à F la date de fin des tâches qui n'ont pas de suivante, et on fixe successivement les dates de fin des taches i dont toutes les suivantes k ont leur date de fin déja fixée, en prenant pour valeur $F_i = \text{Max}_k\ (F_k - t_k)$. La date de début minimale des taches, $\text{Min}\ (F_i - t_i)$, fournie par cet algorithme est évidemment zéro puisque la durée totale de construction dans le calage à droite est F, comme dans le calage à gauche.

S'il existe un circuit dans le graphe, on ne peut marquer tous les sommets ni dans le calage à gauche, ni dans le calage à droite. Il reste donc un certain nombre de sommets non marqués, parmi lesquels se trouvent ceux constituant le (ou les) circuit. Pour déterminer un des circuits, on prend un sommet non marqué, puis un de ses précédents non marqué (qui existe forcément) et ainsi de suite. On retombera obligatoirement à un certain moment sur un sommet par lequel on est déja passé (puisque le nombre de sommets est fini), et on aura donc décrit un circuit.

En résumé les opérations à effectuer pratiquement dans le calage à gauche sont donc les suivantes à chaque application de l'algorithme:

(1) recherche d'une tâche non encore marquée;
(2) pour une telle tâche i, recherche de ses tâches précédentes pour voir si elles sont toutes marquees;

(3) si toutes ses tâches j précédentes sont marquées, calcul des $d_j + t_j$ et report de la valeur maximale en d_i.

Le processus est identique pour le calage à droite. Quelques calculs auxiliaires sont à faire, en particulier le décompte des tâches marquées pour savoir si le processus s'arrête avant qu'on ait terminé le calage en raison de la présence d'un circuit dans le graphe.

ADAPTATIONS DE L'ALGORITHME EN VUE DU CALCUL SUR ORDINATEUR

A chaque stade, pour l'opération 1) ci-dessus, il n'est pas nécessaire de considérer toutes les tâches i non encore marquées, mais il suffit de prendre celles d'entre elles qui sont suivantes des tâches déja marquées. Chaque fois que l'on aura marqué une tâche i, on essaiera de marquer ses tâches suivantes. Si l'une d'elles, j, ne peut pas encore être marquée, cela signifie qu'elle est suivante d'une autre tâche i_1 non encore marquée et qu'on l'essaiera à nouveau au moment du marquage de i_1. Cette procédure, susceptible de réduire notablement le nombre des essais de marquage, nécessite, pour être efficace en ordinateur, la constitution pour chaque tâche i, non seulement de la liste de ses tâches précédentes, mais encore de la liste de ses tâches suivantes. L'ensemble de ces dernières listes est redondant avec l'ensemble des listes de tâches précédentes. Il faut de plus tenir à jour une liste des tâches marquées dont les suivantes n'ont pas encore été essayées.

Une seconde simplification concerne le calage à droite. Lors du calage à gauche, les tâches ont été marquées dans un certain ordre. On peut voir que, lors du calage à droite, on pourra marquer chaque tâche du premier coup à condition de les prendre dans l'ordre inverse. Les opérations (1) et (2) de l'algorithme seront donc supprimees à condition que l'on constitue au cours du calage à gauche une liste des tâches que l'on marque. Cette liste pourra d'ailleurs, avec quelques aménagements, être la mème que celle mentionnée à la fin du paragraphe précédent.

L'opération (3) de l'algorithme dans le calage à gauche nécessite, si la tache i a n_i précédentes, n_i additions et $n_i - 1$ comparaisons. Si au lieu de raisonner sur les dates de début, nous avions raisonné sur les dates de fin f_j, il aurait suffi de $n_i - 1$

comparaisons sur les f_j et d'une addition pour déterminer $f_i = (\text{Max } f_j) + d_i$. De même dans le calage à droite, on travaillera sur les dates de début D_i, au lieu des dates de fin F_i. Cette dernière modification diminue le temps de calcul à chaque application de l'algorithme.

Les deux modifications précédentes diminuent le nombre d'applications de l'algorithme.

CALCUL SUR ORDINATEUR IBM 7090

Les données du calcul fournies par Electricité de France sont:

—les tâches (code et duree)

—pour chaque tâche, les tâches précédentes et les tâches suivantes que l'on structurera en listes.

Le nombre maximal de tâches étudiées est de 3000. Le nombre maximal de contraintes antérieures est de 5000.

L'organisation des données en mémoire est schématisé en Fig. 1.

A chaque tâche est réservé un groupe de cinq mémoires contenant:

—le code E.d.F.

—l'adresse de la tâche qui a été marquée immédiatement avant dans le calage à gauche;

—la durée de la tâche

—l'adresse de la liste des tâches précédentes

—la date de fin minimale

—l'adresse de la liste des tâches suivantes

—la date de début maximale.

Les listes des tâches précédentes ou suivantes sont constituées en affectant une mémoire à chaque tâche précédente ou suivante. Cette mémoire contient en partie décrément, l'adresse du mot suivant de la liste, en partie adresse, l'adresse du mot de la tâche précédente ou suivante contenant la date de fin minimale.

On constate que l'ensemble des données peut être stocke en mémoire centrale.

L'organisation des calculs est schématisée en Fig. 2 où seul le calage à gauche à été détaillé.

Le traitement proprement dit est précéde d'un certain nombre de vérifications. Les tâches sont triées suivant le code alphanumérique E.d.F. avant mise en mémoire afin de faciliter l'impression des résultats. Pendant la lecture des tâches précédentes et suivantes, on vérifie qu'elles ont toutes eu leur durée définie.

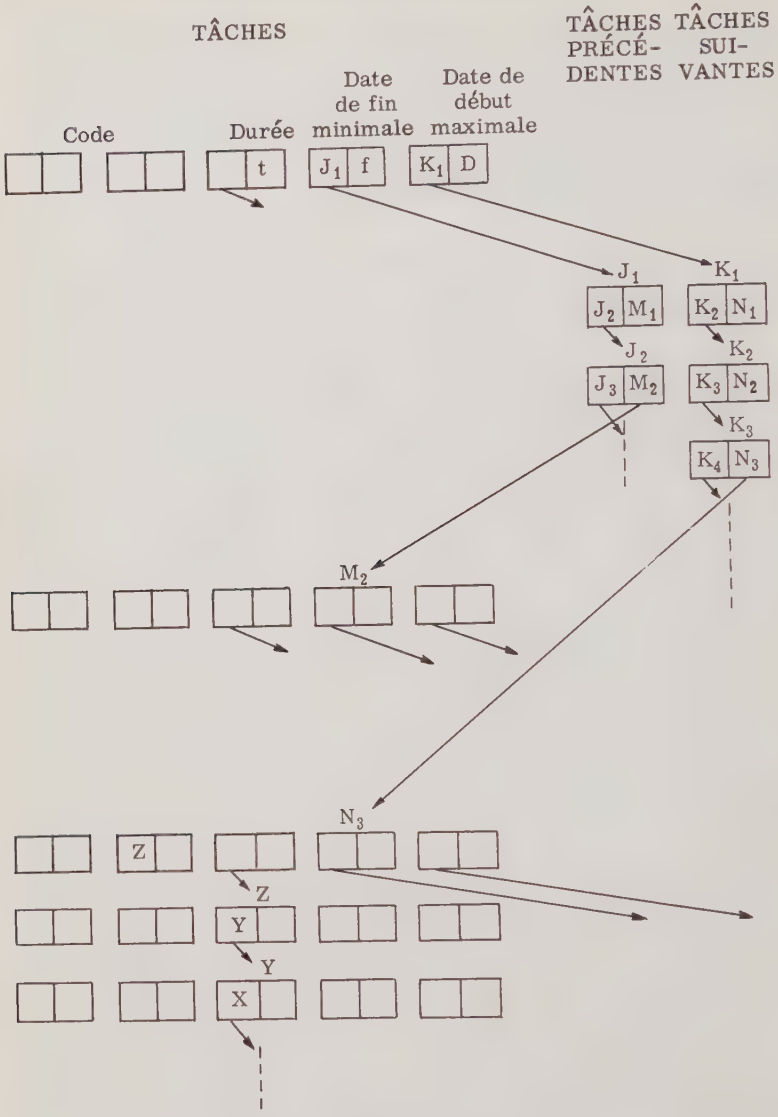


FIG. 1.

Pendant la lecture des tâches suivantes on vérifie que les contraintes postérieures ont déja été données comme contraintes an-

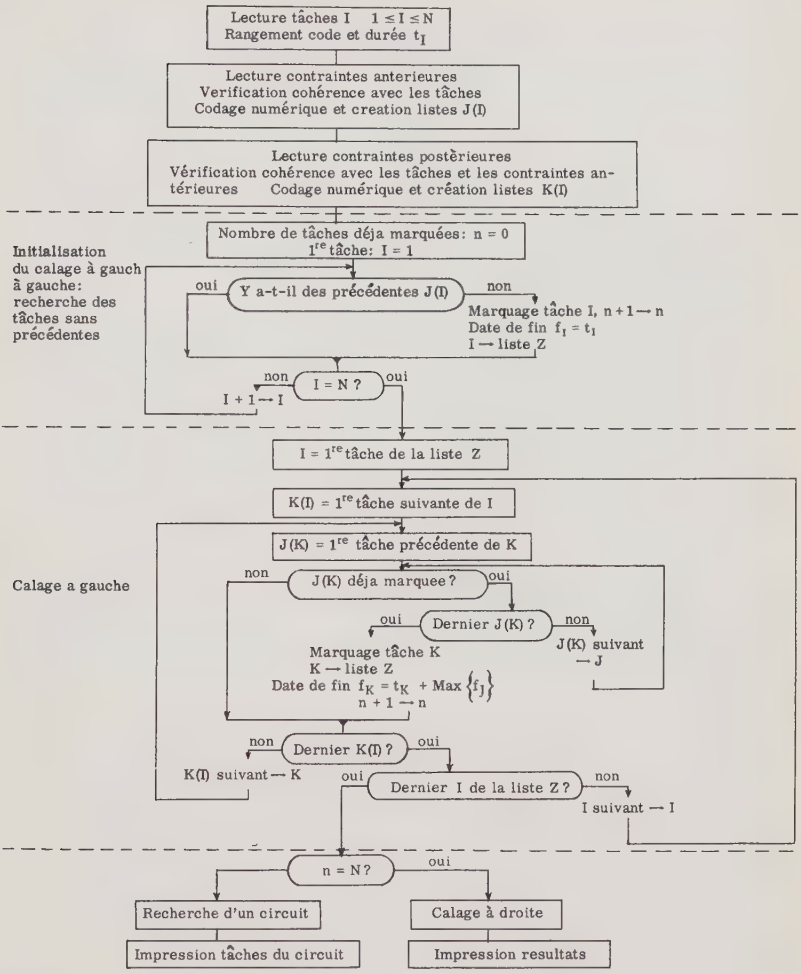


FIG. 2.

térieures. Ceci assure la cohérence des données, exception faite de l'éventuelle existence d'un circuit.

Le calage à gauche est réalisé en appliquant l'algorithme exposé dans les pages précédentes. F.L.P.L. permet une programmation simple de cet algorithme. En effet, la structure de liste permet de solidariser simplement des éléments ayant une carac-

téristique commune et ceci quel que soit leur nombre (exemple: tâches suivantes d'une tâche déterminée).

La constitution de ces listes est facile à réaliser. Au debut du programme, les mémoires sont solidarisées à l'aide d'une seule instruction en une liste de mémoires libres dans laquelle, au cours du programme, on prélève la mémoire dont on a besoin pour prolonger une liste. Une seule instruction réalise la suppression de cette mémoire de la liste des mémoires libres, l'adjonction de cette mémoire à la liste à prolonger ainsi que l'inscription des informations à y ajouter.

L'utilisation de ces listes comporte deux étapes.

La première consiste à suivre la chaine des adresses indiquées dans les parties décrément ou adresse et qui constituent "la liste" pour trouver le mot adéquat.

Le deuxième consiste à prélever dans ce mot l'information recherchée. Cette information peut être constituée:

—par la partie décrément: adresse du mot suivant de la liste;

—par la partie adresse: adresse d'une autre liste ou d'un mot;

—par la partie index: valeur variant de 0 à 4 et pouvant servir à indiquer un marquage.

Par exemple la recherche d'un circuit, qui n'est pas détaillée dans l'organigramme a été programmée comme suit (les tâches déja marquées lors du calage à gauche possédent une partie index valant 1 dans le quatrième mot qui leur est affecté).

Il faut prendre une tâche i non marquée (index = 0). La partie décrément du quatrième mot de cette tâche (J_1 sur la Fig. 1) indique le premier mot J_1 de la liste des tâches précédentes de cette tâche. La partie adresse de J_1 indique M_1: quatrième mot de la première tâche précédent i. Si la partie index de M_1 est nulle, cette tâche est la tâche non marquée cherchée; on met 4 dans sa partie index de manière à la reconnaitre lorsque l'on aura parcouru le circuit. On recommence sur cette tâche les opérations réalisées sur la tâche i.

Si la partie index de M_1 vaut 1, cette tâche a été marquée lors du calage à gauche et il faut étudier une autre tâche précédente de i.

A cet effet, la partie décrément du mot J_1 contient l'adresse J_2 du deuxième mot de la liste des tâches précédentes de i. La partie adresse de ce mot J_2 indique M_2 dont on étudie la partie index comme on l'a fait pour M_1.

Si la partie index de M_1 vaut 4, cela signifie que l'on avait mar-

qué M_1 au cours de l'une des applications de l'algorithme précédent, donc que l'on a décrit un circuit.

Ceci exige 13 instructions FORTRAN dont 6 instructions FLPL.

Le calage à gauche a été programmé a l'aide de 56 instructions FORTRAN dont 23 instructions FLPL.

CONCLUSION

On constate que l'utilisation de FLPL facilite beaucoup la programmation des problèmes non numériques à structure de graphe. Ce type de problème exige des opérations analogues à celles realisées par les instructions FLPL, dont l'existence réduit donc la longueur des programmes et le temps de programmation. Toutefois la simulation de mémoires associatives sur une mémoire classique utilise une partie importante de la mémoire. De plus l'utilisation des instructions FLPL est nettement plus délicate que celle des instructions FORTRAN classiques.

REFERENCES

1. Bernard ROY, These de Doctorat 1960 (disponsible a la S.E.M.A.)
2. id., Contribution de la theorie des graphes a l'etude des problemes d'ordonnancement (2eme Conference Internationale de Recherche Operationnelle—Aix en Provence 1960—English Universities Press Ltd.)

ON THE IMPLEMENTATION AND USAGE OF A LANGUAGE FOR CONTRACT BRIDGE BIDDING

A. L. Bastian, J. P. Foley, and S. R. Petrick

Air Force Cambridge Research Laboratories
Bedford, Massachusetts, USA.

INTRODUCTION

It must be initially noted that the language whose machine implementation and usage are described in this paper is not a specialization suited only for bridge bidding. It is instead a conveniently used dialect of the LISP meta-language. This language, in conjunction with a large number of specialized bridge functions defined in terms of it, does permit the specification of the mass of complex rules which constitute a bidding system. The work described should, therefore, still be of interest to those concerned with the bidding application as well as to the users of such symbol manipulation languages as LISP. A third set of readers to whom this paper is directed includes the designers and users of phrase structure language analyzer—translators. Such a program was developed and used by the authors of this paper as will be described.

A few words are required as to what were the goals of this effort, and why bridge bidding was chosen as a research vehicle. One of the authors' interests is the specification, machine realization, and usage of languages suitable for the description of complex logical processes. Other interests include machine 'learning' and pattern recognition, particularly as applied to speech recognition. Consequently, the complete mechanization of contract bridge was selected as a project because it was felt to embody some of the salient features of all of the above areas. Control of the computer by means of spoken bids, appropriate bidding according to one or more recognized bidding systems, and increased performance in the play of a bridge hand as more examples of approved play are available, were all to be considered. In addition, it was felt that

their integration into a single system should be interesting as the complete realization of a complex set of abilities which are commonly regarded as requiring a high degree of human intelligence.

The only aspect of the overall project with which this paper will be concerned is the problem of specifying a bidding system to a computer. It has been pointed out that this is not a good application for pattern recognition (1, 2). It is instead more reasonable to treat the problem as one of enumeration, specifying proper bids as a function of the sequence of previous bids and of the hand itself or properties derived from it, such as pointcount and distribution of suits. The outstanding difficulty, of course, is that even a rudimentary definition of a bidding system requires a huge amount of specification. A cursory investigation shows that tabular storage of acceptable bids as a function of a set of derived properties is prohibitive. There are just too many basic attributes of a bridge hand which are required to accurately discriminate between certain alternative bidding responses to permit the enumeration of all of their combinations, most of which are mutually incompatible.

There is, however, at least one way of specifying a bidding system as is evidenced by the numerous books which at least purport to do so. As anyone who attempts to learn bidding by means of such a book will testify, they abound with inconsistencies, ambiguities, and completely unspecified situations. Nevertheless, it is possible to learn a bidding system in some sense from a bridge textbook. It was decided, therefore, that a reasonable approach to mechanical bridge bidding was to replace a bidding description expressed in natural language by one in an artificial language understandable by some computer.

In this respect, the bridge application of this paper is similar to many practical problems requiring the specification of the proper responses to a number of possible situations. The analogy is, of course, limited to those applications for which a precise set of instructions exists, but these may be extremely complicated and voluminous. It is hoped that the experience which was gained in conducting this exercies for bridge bidding will be of interest and utility to others concerned with such applications.

USE OF THE LISP S-LANGUAGE

An examination of conventional bidding descriptions discloses two pertinent facts. First, there are a large number of bridge

expressions whose meanings are either assumed or defined in more primitive terms. Ultimately, these terms may be expressed as a function of the cards which make up a bridge hand. Second, bidding instructions are customarily expressed by means of a sequential decision tree. Branching on this tree makes use of propositions which are functions of the various bridge terms. Corresponding to every terminal node there is an appropriate bid.

A number of existing programming languages might have been chosen because they permit convenient definition of bridge terms (functions) and because they also are suited for specifying the required decision trees. The MIT LISP Programming System was selected, however, primarily because its essentially machine-independent nature eliminates many bothersome programming details. It is not possible to describe the LISP programming language here, but several references to it include a programming manual (3) and papers by its originator, McCarthy (4) and by Woodward and Jenkins (5).

The LISP language makes use of a so-called conditional expression which is conveniently used to specify the necessary sequence of logical decisions to produce the desired bid. This is analogous to the ALGOL conditional statement and is written $[p_1 \rightarrow e_1; p_2 \rightarrow e_2; \ldots; p_n \rightarrow e_n]$ where the p_i's are Boolean functions. The value of the conditional is the e_i corresponding to the first true p_i.

LISP functions are customarily written in a language called the M-language (metalanguage). Unfortunately, input to the IBM 7090 LISP System must be expressed in a related language called the S-language (symbolic language). This has been described as "a Cambridge-Polish prefix language which is writeable and somewhat readable". Contrary to what the casual observer might conclude on first acquaintance, this is more or less true for functions of up to moderate length and complexity. Because the magnitude of the bidding project only gradually became evident, however, the difficulties associated with the use of the LISP S-language were not immediately apparent. Therefore, LISP programming proceeded smoothly for some time. Small functions were often written directly in S-language and larger functions were either also so treated, or else they were first written in M-language and then translated by hand to the corresponding S-language representation.

Initially, the bidding system on which mechanization proceeded was a simplified natural bidding system devised by Mr. Daniel Rotman, who is a life master and past winner of a U. S. open pairs championship. The general procedure followed was to avoid the confusion of excessive nesting of conditional expressions by defining new auxiliary functions. The opening bid procedure was the following:

Define the characteristic triplet E of a bridge hand to be a list of three numbers, the first component being the number of doubletons (two card suits), the second component being the number of singletons (one card suits), and the last component being the number of voids (no card suits). Let P be the number of high card points in the hand (A = 4, K = 3, Q = 2, J = 1). Let HRLS of a hand be the higher ranking longest suit of that hand (length having priority over rank), and similarly let LRLS be the lower ranking longest suit. If P = 16, 17, or 18 and if E = (0, 0, 0) or (1, 0, 0), bid one no trump. If P < 12, pass. If P = 12, pass, unless you have three aces, two aces and a king, two suits, one of which has seven or more cards and the other three or more cards, or two suits, one of which has six cards and the other at least four cards.

If you neither pass nor bid one no trump, bid one club if E = (0, 0, 0). If E = (1, 0, 0), bid one spade if HRLS = spades and LRLS = hearts, bid 1 LRLS if HRLS = spades and LRLS ≠ hearts, bid one heart if HRLS = hearts and LRLS = diamonds; bid one LRLS if HRLS = hearts and LRLS ≠ diamonds, and otherwise bid one HRLS. If E = (1, 0, 2), bid seven HRLS if you have two aces and two kings, six HRLS if you have two aces, and one HRLS otherwise. If E is anything else, bid one HRLS.

It must be emphasized that the opening bid presented above is admittedly crude and was given merely because it is less space consuming and easier to follow than a more complicated later version which added such features as opening bids above the one level, checks to insure all suits have a "stopper" for no trump play, more refined treatment of potential slam bidding, etc.

That portion of the LISP M-language which pertains to the bidding of one no trump is given below. This will not be completely meaningful to readers unfamiliar with LISP, but it gives an idea of the complexity of required programming. It also illustrates that a user who wishes to define his own bidding system using LISP must spend a substantial period of time learning the language. In the set of functions which follows, e

and p correspond to the previously defined E and P, A and K are the number of aces and kings, and ℓ is the hand itself. Indentation is used merely for ease of reading and is not a syntactic feature of the language.

bid1 [ℓ] =
    bid11 [ℓ; distntuple [ℓ]; ptsinsuits [ℓ]]

bid11 [ℓ; m; n] =
    bida [ℓ; m; n; chartrip [m]; ptcount [n]; noofaces [ℓ];
    noofkings [ℓ]]

bida [ℓ; m; n; e; p; a; k] =
    [greaterp [p; 15] $\wedge$ lessp [p; 19] $\wedge$ [equal [list [0; 0; 0];
        e] $\vee$ equal [list [1; 0; 0]; e]]] $\rightarrow$ list [1; NO,
        TRUMP]; T $\rightarrow$ bid12 [m; n; e; p; a; k]]

The function bid 12, the definition of which will be omitted, covers the rest of the opening bid.

The corresponding S-language translation is given by:

(BID1, (LAMBDA, (L), (BID 11, L, (DISTNTUPLE, L),
(PTSINSUITS, L))))

(BID11, (LAMBDA, (L, M, N), (BID A, L, M, N, (CHARTRIP, M),
(PTCOUNT, N), (NOOFACES, L), (NOOFKINGS, L))))

(BIDA, (LAMBDA, (L, M, N, E, P, A, K), (COND, ((AND,
(GREATERP, P, 15), (LESSP, P, 19), (OR, (EQUAL, (LIST 0, 0, 0),
E), (EQUAL, (LIST, 1, 0, 0), E))), (LIST, 1, (QUOTE, NO),
(QUOTE, TRUMP))), (T, BID 12, L, M, N, E, P, A, K)))))

DIFFICULTIES INHERENT TO BRIDGE

Unfortunately, Mr. Rotman left AFCRL before the project was very far along, leaving a group of rank bridge amateurs to grapple with the problems unique to contract bridge. Work progressed to responses and rebids, and the mushrooming complexity and volume of specification required more and more bid-defining decisions. It was necessary to face such problems as what constitutes sufficient interest in further information about major suits to indicate use of the Stayman Convention. Recourse to various bridge books did very little to resolve these problems, which were merely painful stumbling blocks of no direct interest in themselves.

After some consideration it was decided to switch from a natural bidding system to a hopefully better specified artifical system. The Roman Club system which has been used so successfully by Giorgio Belladonna and Walter Avarelli of the world championship Italian Bridge team was chosen primarily because it seemed well documented (6, 7). It was not hoped that the system would meet a book jacket promise (6), "when a player . . . has learned the Roman Club, he can hardly make anything but a perfect bid. Because there is almost no bidding situation possible in bridge that does not, under the Roman Club System, call for one and only one clear, correct, perfect bid." Nor was there any worry wasted on the Italian bridge team captain, Carl' Alberto Perroux's answer to the question of whether a book (6) on the Roman Club System was complete. He answered, "It is complete. I'd say even too complete." Captain Perroux also described this book as 'clear as any explanation of an artificial system can be'. While the authors would dispute this fact, it is nevertheless true that the system is relatively well defined. If, as is likely, the present authors' interpretation of this system is faulty, the error is either due to their unwillingness to treat a situation in exhaustive detail or else is due to a misunderstanding of the system.

No attempt was made to make the program nondeterministic by permitting probabilistic alternate responses. The factors considered in many situations are admittedly insufficient, but it is felt that although the resulting performance is inferior to that of many bridge players, it is probably better than that achieved by most players. Complete agreement is expressed by the present authors with the following statement by Alfred Sheinwold (8), "In short you never really *know* what your hand is worth. The best you can do is to come pretty close most of the time.

"This uncertainty is one of the charms of the game. If anybody ever succeeds in making bridge really cut and dried, so that you can bid or play automatically, people will desert the game and take up something more adventurous."

Mr. Sheinwold need lose no sleep over the number of bridge players who will desert the game as a result of this paper.

PROGRAMMING DIFFICULTIES ENCOUNTERED

As the complexity and volume of bidding specification expanded alarmingly, so did the mechanical problems of coding and de-

bugging the bidding procedures. It was initially believed that once a basic set of bridge functions was available, description of the bidding procedure would be quite simple. In practice it was found that sheer volume demanded changes in procedure. For one thing it was decided that the more liberal use of mnemonic function names would contribute to both ease of reading and of remembering for subsequent writing. Accordingly, even such LISP functions as car and cons were renamed levelofbid and bid, respectively. Second, rather than trying to make extensive direct use of previously defined bridge terms, new functions were defined as needed. This led to a large number of functions (about 125 at last count) but was found to foster ease of reading as well as writing. Third, principal bidding procedures such as 'rebid' were defined as one long function written in a modified LISP metalanguage. Nested conditional expressions were found to be confusing, and so they were replaced by a paragraphing system using level indicators, usually indented for ease of reading. Nesting to a depth of as many as eight levels made it very attractive to eliminate the necessity of closing off conditional statements with the proper number of parentheses. An example of this language is the opening bid definition for the Roman Club System which is given in Figure I.

Most of the bridge terms defined by means of special functions are self-explanatory. 'Preempt' can be either a preemptive bid or a pass. 'Greaterp [x; y]' and 'lessp [x; y]' are predicates which are true if and only if $x > y$ and $x < y$, respectively. 'Balanced' is a predicate true only for suit distributions 4, 4, 3, 2 or 4, 3, 3, 3. A 'threesuithand' is one of distribution 5, 4, 4, 0 or 4, 4, 4, 1. 'Length' might well have been written 'noofbiddablesuits' (four or more card suits). A 'goodbutnotgreathand' is one of 15 or more points which contains no suit with more than four cards and more than four high card points. An 'lr3csuit' is the lowest ranking three card suit. 'Firstsuit' is the highest ranking biddable suit. 'Lengthsuit1' and 'lengthsuit2' are the number of cards in the highest ranking and next highest ranking biddable suits, respectively. The 'lrshorter' (lower ranking shorter) of two suits is the suit containing fewer cards. If the suits are of equal length, it is the lower ranking suit.

Use of this modified M-language proved satisfactory except in one respect. It could not be accepted directly by any computer, and so translation to the LISP S-language was necessary. This was done begrudgingly by hand at no faster a pace than was

```
bidrc1 [ℓ] = bidrc11 [ℓ; ptcount [ptsinsuits [ℓ]]; distntuple [ℓ];
             biddablesuits [ℓ]]
bidrc11 [ℓ; p; d; b] =
   A1, lessp [p; 12] → preempt [ℓ; ℓ; p; d]
   B1, lessp [nooflosers [ℓ]; 4.5] Λ greaterp [p; 21] → bid [1; CLUB]
   C1, balanced [d; ℓ] →
       A2, lessp [p; 17] V greaterp [p; 20] → bid [1; CLUB]
       B2, T → bid [1; NOTRUMP]
   D1, threesuithand [ℓ] →
       A2, lessp [p; 17] → bid [2; CLUBS]
       B2, lessp [p; 21] → bid [2; DIAMONDS]
       C2, T → bid [1; CLUB]
   E1, equal [length [b]; 1] →
       A2, equal [firstsuit [b]; CLUBS] V goodbutnotgreathand [ℓ; p; b] →
           bid [1; lr3csuit [d]]
       B2, T → bid [1; firstsuit [b]]
   F1, equal [secondsuit [b]; CLUBS] →
       A2, greaterp [lengthsuit1 [b]; 4] Λ greaterp [lengthsuit1 [b];
           lengthsuit2 [b]] →
           A3, equal [firstsuit [b]; DIAMONDS] → bid [2; NOTRUMP]
           B3, T → bid [2; firstsuit [b]]
       B2, T → bid [1; firstsuit [b]]
   G1, equal [lrshorter [b]; DIAMONDS] Λ lessp [p; 15] → bid [1;
       firstsuit [b]]
   H1, T → bid [1; lrshorter [b]]
```

FIG. 1.

necessary to keep up with debugging. Use of cooperative students was tried with varying success and the use of one private data processing company was tried with no success. Although the translation rules were few and simple, it was definitely established that the numerous errors introduced during translation by human clerical help caused the authors to expend more subsequent time in debugging than would have been required for them to both translate and debug. Even when personally translated, however, errors were introduced.

It was further discovered that LISP programming errors are not always easy to find. Parentheses failing to count out are, of course, diagnosed, but no information is given as to why not. The use of S-expression functions of a packed 7090 page or more is impractical for this reason, and large functions must be temporarily separated into smaller, more manageable functions. Many other syntactic errors are not diagnosed during machine definition and may only be found by observing their operational effect. Tracing (printing the values of subsidiary functions) can

be used, but this usually results in finding a single error at a moderate cost of time and effort. It soon became apparent that a computer translator was necessary if a comprehensive bidding system was to be successfully defined.

THE PHRASE STRUCTURE LANGUAGE TRANSLATOR

At about the same time that a need was seen for machine translation from the metalanguage to S-language, the authors became interested in reports on general purpose compilers for translating phrase structure languages. It was decided to write such a translator, not only for its application to the bidding problem and for several other foreseeable requirements, but also to investigate more closely the construction of this type of compiler. The work of Brooker and Morris (9) was studied as was that of Warshall (10), but the translator actually written was more closely related to one of Irons (11). This paper is more concerned with its usage than of its internal construction, and a separate report is now in preparation on the latter. We therefore limit ourselves here chiefly to describing the performance and benefits obtained from the use of the compiler.

About two man months were required to write and perhaps ten runs to debug the compiler. Subsequently, it was operational, but new features were added daily (they are still being added at almost that rate), resulting in an overall speed-up of more than 100 to 1 and storage savings of perhaps 40% over the initial performance.

At present the longest input function which can be translated is about 14,000 characters long. Pushdown and output storage have been allocated for the present application to roughly run out concurrently. As an indication of operating speed, the rebid function for an initial bid of one club was about 2,500 input characters in length and required six seconds for translation. The computing time required is roughly linearly related to the length of the input string.

The limitations of the IBM character set necessitated some compromise with the previously defined metalanguage. As an example of the version actually punched, we have:

```
TWOSUITRESPONSE (B, D) =
  $1, EQ (SECONDSUIT (B), *HEARTS*)/BID (1,
      LRSHORTER (B)
```

```
$1, EQ (FIRSTSUIT (B), *DIAMONDS*)/
      $2, GREATERP (LENGTHLONGESTSUIT (D), 4)/
          BID (2, HRLS (D))
      $2, T/BID (1, LR3CSUIT (D))
$1, T/BID (1, FIRSTSUIT (B))
```

To precisely define the syntax of this language we give the following phrase structure grammar with initial symbol <dfnst>, and terminal vocabulary consisting of the IBM character set.

<dfnst> : : = <symbol> <lparen> <arglst>
<rparen> <equals> <def> | <comma> <dfnst>

* <symbol> : : = <ltrgrp> | <symbol> <ltrgrp> | <symbol>
<integr>

* <ltrgrp> : : = <letter> | <ltrgrp> <ltrgrp>

* <letter> : : = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
Q | R | S | T | U | V | W | X | Y | Z

* <integr> : : = <digit> | <integr> <integr>

* <lparen> : : = (| <comma> <lparen>

* <rparen> : : =) | <comma> <rparen>

<equals> : : = = | <comma> <equals>

* <comma> : : = <blanks> | <blanks> <comma> | <comma>
<blanks> | ,

* <blanks> : : = <blank> | <blanks> <blanks>

<blank> : : =

* <arglst> : : = <symbol> | <arglst> <comma> <symbol>

<def> : : = <premis> <end> | <pair> <end>

* <premis> : : = <fnctn> | <orgrp> <rparen> | <andgrp>
<rparen> | T

<fnctn> : : = <symbol> <lparen> <args> <rparen>

* <args> : : = <integr> | <integr> . <integr> | <symbol> | <args>
<comma> <args> | <qtdsym> | <fnctn>

<qtdsym>: : = *<symbol>*

<orgrp>: : = <lparen> <premise> <or> <premis>|<orgrp>
<or> <premise>

<andgrp>: : = <lparen> <premise> <and> <premise>|
<andgrp> <and> <premise>

<pair>: : = <lvlind> <premis> <arrow> <concl>|
<pair> <comma> <pair>

<arrow>: : = /|/ <comma>|<comma> <arrow>

<lvlind>: : = $ <integr> <comma>|<comma> <lvlind>

<concl>: : = F|NIL|T|<integr>|<fnctn>|
<integr>. <integr>|<qtdsym>|<pair>

<and>: : =$ AND

<or>: : = $ OR

<end>: : = $ END|<comma> <end>

A set of additions which permit use of the standard LISP M-language conditional are:

<def>: : = <cond> <end>

<cond>: : = <lparen> <pclist> <rparen>

<concl>: : = <cond>

<pclist>: : = <premis> <arrow> <concl>|<pclist> <comma>
<pclist>

The order of applying the above rules is immaterial.

The source language definition actually given to the computer differed from that presented above only in format and by the inclusion of several definitions implied by the others purely for the purpose of speeding up the translation. Along with each rule of the grammer given, of course, there is also given a string of symbols (taken from the target language alphabet and from the immediate constituents appearing in that rule) which are to be substituted for the constituent defined by that rule for potential output purposes.

The syntax compiler speeded up the project by eliminating the necessity of laborious, error-prone hand translation. Debugging of bidding proceeded at perhaps ten times the previous rate per unit of human effort expended, using about one fourth of the formerly required machine time. This was made possible in large measure because the syntax definition was changed to attempt to continue the diagramming procedure when it is discovered that a given function is not well-formed. In this case several plausible hypotheses are examined regarding the nature of the error in syntax. Each such assumption is reflected by an appropriate rule of formation involving a very general constituent <nsx>. Whenever one of these rules of formation is applicable, a special function causes an informative error printout, and further output is withheld, although the search for a structural description continues. It is, of course, necessary to order the application of the rules so that those reflecting errors in syntax are never used unless such errors actually exist.

One assumption made is that all instances of illegal syntax are local to one 'premis, conclusion pair'. Hence, when the compiler establishes that no legal structural description exists, it assumes an error in syntax in the premis, conclusion pair on which it failed. This pair is preceded by some level indicator and is terminated by the first identical level indicator (or by the end of the function). Having found a 'legal error', the search continues for the required terminal derivation.

Used in this manner for diagnosing a class of deviations from allowable syntax, the compiler is more than a decision procedure. If there exists a structural description that can be produced by the grammer, it is found, and thus the compiler is a perceptual device which is capable of understanding every LISP function in the manner of the grammer previously given. If there is no structural description corresponding to some given LISP function, diagnostic information is supplied about the number, location, and nature of the deviations from allowable syntax. This is similar to the corresponding desirable property of a natural language grammar to be able to tell the degree of grammaticalness of a string which is not a sentence. For example, the phrase 'a grief ago' could appear in no English sentence, but its deviation is small enough that its meaning is perfectly clear to the reader.

The use of such a program as has been described appears of considerable value for diagnostic pupposes in determining

whether a given computer program is well-formed, and, if not, to what extent. As an illustration of this, the authors also wrote a context free grammer for LISP S-language, and appended to the definition of each constituent an equivalent string of immediate constituents and M-language characters in order to permit the reverse S to M-language translation. This was done primarily to obtain a readable description of many complex functions which existed only in S-language and whose undocumented nature had long since been forgotten. For this intended purpose it proved successful, but unexpectedly in addition, several LISP users who already were debugging sizeable LISP S-language programs made use of the reverse translation procedure just to find syntactic errors in their programs more easily than was possible by means of the diagnostic aids directly available in the LISP 7090 System. This procedure looks especially attractive because the specification of syntax by means of a context free grammar takes no more than an hour or two and the class of artificial languages which can be generated by such grammars is rich enough to include at least the present generation of problem-oriented languages. It has been shown in this section that the LISP M-language is a context free language, and the same is true for Algol and Cobol. Formal demonstration for the latter has been carried out by researchers at the Case Institute of Technology in writing a Cobol compiler.

In connection with the ease of specifying syntax it should be remarked that a large number of cards puched by a private company contained consistent errors in the use of blanks adjacent to the characters denoting 'or' and 'and'. Rather than repunching these cards, a simple change in syntax made the compiler conform instead to the syntax of the existing cards.

The grammer needed to define the LISP S-language consists of those previously given rules of formation which are starred * and augmented with the additional rules below. The initial symbol is again <defnst> and the terminal vocabulary is as before.

<defnst> : : = DEFINE <lparen> <lparen> <lparen> <symbol>
<lparen> LAMBDA <lparen> <arglst> <rparen>
<def> <rparen> <rparen> <rparen> <rparen>

<pair> : : = <lparen> COND <lparen> <premis> <concl>

<rparen> | <pair> <lparen> <premis> <concl>
<rparen>

<qtdsym> : : = <lparen> QUOTE, <symbol> <rparen>

<fnctn> : : = <lparen> <symbol> <args> <rparen>

<andgrp> : : = <lparen> AND <premis>

<orgrp> : : = <lparen> OR <premis>

<def> : : = <fnctn> | <pair> <rparen>

<concl> : : = <def> | F | NIL | T | <integr> | < integr>
< integr> | < qtdsym>

As in the case of the rules of formation relevant to error diagnosis, the order of application of certain rules is important.

It is noted that some of the rules of formation specify the concatenation of many immediate constituents. In such circumstances one feature of the present compiler which apparently was not used previously is of special importance. This is a so-called look-ahead feature. In attempting to determine a structural description of a given sentence, the compiler matches the input string with the immediate constituents of all relevant rules of formation. If it is thus successful in making up a constituent, it consults a table to determine whether this constituent can lead to a structural description of the required type. This may be the ultimate derivation sought, or it may be a different type of structural description which is a necessary subgoal in order to attain that ultimate goal. Thus we have a structure of the following type:

Starting at point A we proceed toward our goal by matching the input string to the various paths along which we may traverse. At certain points we are given information as to whether all paths going on do not lead to the goal. Rather than arriving at such a point only to find it should not be pursued further, the compiler of this report looks ahead to preclude such a possibility before dealing with the input string to actually reach that point. The inclusion of this facility was found to speed up

the LISP M to S-language translation by a factor of from 100 to 200 times, making it the difference between practicality and impracticality.

EXTENT OF BRIDGE SYSTEM MECHANIZED

The volume of coding necessary, however easy it is to write and debug, is appallingly large. It was hoped that at least one path corresponding to all variations of a single, relatively simple opening bid could be presented here. All continuations arising from an initial Roman Club bid of one no trump, for example, have been coded, but even this requires six typewritten pages of specification exclusive of adverse bidding considerations. An AFCRL station report soon to be released will include the actual coding which defines the bidding systems considered.

At this time the coding completed includes the opening bid and response for a natural bidding system; opening bid, response, rebid, and portions of the second response for the Roman Club System; and the previously mentioned continuations of one no trump, including possible bidding by the opponents. Rather than continuing further in bidding it was decided to first integrate the existing bidding capability into a real time voice input system and a bridge playing 'learning' model. This is currently in progress, and very little additional effort will be placed on completing the Roman Club bidding system definition. Some thought has been given to finishing the specification on a relatively crude basis.

The portion of the Roman Club System presently defined appears quite respectable although admittedly not of championship caliber. As an indication of its performance, a few bidding sequences are included below.

(COMPUTER II)

♠ Q, 9, 7, 4

♡ 7, 6, 5, 2

◇ Q, J

♣ A, 6, 5

(Petrick)

♠ J
♡ Q, 9
♢ 9, 8, 7, 4, 2
♣ Q, 9, 8, 7, 4

(Foley)

♠ 10, 6, 5, 3, 2
♡ A, K, J, 10
♢ 5, 3
♣ 3, 2

N
W E
S

(COMPUTER D

♠ A, K, 8
♡ 8, 4, 3
♢ A, K, 10, 6
♣ K, J, 10

North	East	South	West
—	Pass	1 N. T.	Pass
2 ♢	Pass	2 N. T.	Pass
3 N. T.	Pass	Pass	Pass

(COMPUTER)

♠ A, J, 10, 4
♡ J, 10, 9
♢ A, K, J
♣ Q, J, 4

(Foley)

♠ K, 6, 5, 2
♡ 6, 3, 2
♢ 8, 5, 2
♣ 8, 6, 3

N
W E
S

(Urbano)

♠ Q, 8, 7, 3
♡ 5, 4
♢ Q, 7, 6, 4
♣ 7, 5, 2

(Petrick)

♠ 9
♡ A, K, Q, 8, 7
♢ 10, 9, 3
♣ A, K, 10, 9

North	East	South	West
1 N. T.	Pass	4 N. T.	Pass
5 ♠	Pass	6 ♣	Pass
6 N. T.	Pass	Pass	Pass

(COMPUTER II)

♠ K, 10, 8, 6
♡ K, 7, 4
◇ 10, 6, 3
♣ 9, 7, 2

(Petrick)

♠ Q, J, 4
♡ Q, J, 9
◇ J, 7, 2
♣ K, 10, 4, 3

N
W E
S

(Foley)

♠ 5, 3, 2
♡ 10, 8, 5
◇ K, 9, 5, 4
♣ A, Q, J

(COMPUTER I)

♠ A, 9, 7
♡ A, 6, 3, 2
◇ A, Q, 8
♣ 8, 6, 5

North	East	South	West
--	--	1 ♣	Pass
1 ◇	Pass	1 ♡	Pass
1 ♠	Pass	Pass	Pass

CONCLUSIONS

The bidding project described in this paper is primarily of interest as a study of the problems involved in specifying a mass of complex decision making rules to a computer. Mechanization of this specification necessitated the use of a compiler, but the phrase structure translator produced for this purpose could have many other applications. In particular, the utility of such a

program for detecting errors in the syntax of problem-oriented language sentences was emphasized. The authors believe the compiler used in this project is extremely efficient with respect to both computing time and storage requirements.

The Roman Club bidding system was only partially specified due more to the difficulty in precisely defining any bidding system than to the practical difficulties associated with coding and debugging a completely defined system. The bidding performance obtained was quite good, and it is currently being incorporated with real time voice input capabilities and with a bridge-hand-playing 'learning' program. The resulting artificial bridge player should be interesting as a system operating within the framework of a complex set of human rules while at the same time able to act upon oral input and to improve certain aspects of its performance.

REFERENCES

1. Samuel, A., 'Programming Computers to Play Games', Advances in Computers, Vol. I, 1960.
2. Lefkovitz, D., 'LEX, a Bridge Bidder for the IBM 650', Moore School of Electrical Engineering, University of Pennsylvania.
3. McCarthy, J., et al., LISP 1.5 Programmer's Manual, Computation Center and Research Laboratory of Electronics, MIT, Cambridge, Mass. July 1961.
4. McCarthy, J., 'Recursive Functions of Symbolic Expressions and Their Computation by Machine', Comm. ACM 3, 1960.
5. Woodward, P. M., and Jenkins, D. P., 'Atoms and Lists', The Computer Journal, Vol. 4, p. 47.
6. Belladonna, Giorgio, and Avarelli, Walter, The Roman Club System of Distributional Bidding, Simon and Schuster, 1959. (Translated from 'Bridge. Il Sistema Fiori Romano'.)
7. Kaplan, Edgar, The Complete Italian System of Winning Bridge, Ives Washburn, Inc., N.Y., 1959.
8. Sheinwold, Alfred, 5 Weeks to Winning Bridge, Pocket Books, Inc., New York, 1959.
9. Brooker, R. A. and Morris, D., 'A General Translation Program for Phrase Structure Languages', J. ACM 9, 1962.
10. Warshall, S., 'A Syntax Directed Generator', Proceedings of the 1961 EJCC.
11. Irons, E. T., 'A Syntax Directed Compiler for Algol 60', Comm. ACM 4, 1961.

NOTE ON SOME LEXICAL AND PHILOSOPHICAL IMPLICATIONS OF A COMPUTER SYMBOLIC LANGUAGE

R. Busa, S.J.

Centro Automazione Analisi Linguistica
Gallarate-Varese, Italy

I. WHAT PLANT IS

A group of programmers (four at the present, March,'62) in our Center are preparing for the IBM 7090 a Program for Linguistic Analysis of Natural Texts (PLANT). We adopted the FAP for it. The program is conceived as a utility one.

We want it to be able to process natural texts of any language and any subject, with the following limits: no languages having ideographic writing will be processed unless they have been transcribed into phonetic writing; nor languages which do not write separation between the "words," unless they are transcribed or pre-edited with something indicating those separations.

Consequently the PLANT must process also languages which are not Indo-European or which are written from right to left. There are no limits in the number of letters, punctuations, diacritical marks. It provides a coding of the letters of the words so as to arrange them in any wanted sequence, even when this does not fit with the symbol sequence built in the machine. It provides also features for automatic phonetic transcription, i.e., into symbols keeping fully a bi-univocal correspondence phoneme/symbol.

The target is a tape of a given text, which can list or reproduce census and inventories of its vocabulary: word indices, concordances; indices by frequency, by ambience; lists and frequencies of initials and endings, stems, digrams, trigrams, syllables and so on: both in graphic writing and in phonetic transcription and for use both in lexical and in structural or generally statistic analysis.

Our aim is, in fact, to prepare a tool which will make practi-

cally and economically accessible to every scholar the processing of texts in millions of words, for pure or applied linguistic research.

II. PLANT'S LANGUAGE

Using the PLANT will evidently demand control cards: for managing the input of the texts, being so different one from another, and their outputs, i.e., those outputs wanted by the user from among all the possible program-outputs.

We are now working at the language of those control cards. It is not only a symbolic one, in the restricted meaning of that expression, but an interpretative one. It has of course many close analogies and correlations with strictly symbolic languages.

III. ITS PHILOSOPHICAL AND LEXICAL PROBLEMS

Naturally it is exactly in establishing its symbolism, more than in designing its interpretative functions, that we have to face linguistic and philosophical problems, which to our mind can be summarized as follows.

Categories of data and operations must be detected. Their basic elementary unit must be found, as well as their grouping into progressively larger functional units.

Then those categories or classes or types of language components must be expressed by symbols facilitating both the programmer in his job and the scholar in writing down the information necessary for the control cards.

How to find out, how to define, how to classify, where to fix the borders of the unity, from the elementary one up to the most functionally organized one, are exquisite philosophical problems.

But they are even more philosophically pregnant when applied to the linguistic data, to the operations of researching on linguistic data and to the results of those searches. In fact the "materia circa quam" are not exactly the things, nor exactly the concepts, but those strange realities which we call words, i.e. symbols i.e. things expressing the concepts of other things.

On that we try to get some help from comparing and combining together the aristotelic philosophy with contemporary operational resources.

Those units of individuals or classes of data and operations must then be expressed by symbols acting as a communication system between the scholar, the PLANT, and the user of the PLANT. It is not necessary to add that those symbols must be easy to learn and easy to memorize.

Therefore we try to cut those symbols off from the specific glossary in use in philology and linguistics.

This also involves discovering what can be taken as a standard terminology, if there is any in linguistic sciences.

And that is the lexical problem we meet creating the symbolic language for the control cards of our PLANT.

Any suggestion and help will be highly appreciated.

Du FLEC AU C.P.A.S.

J. Legras

Université de Nancy,
France

INTRODUCTION

Cette note est le résumé des recherches en programmation faites au Centre de Calcul de l'Université de Nancy; des recherches analogues ont été conduites dans d'autres Centres de Recherches et nos propres résultats rappelleront fréquemment des résultats cités dans d'autres rapports. Il a paru cependant utile de faire état de ces recherches, bien que le matériel auquel correspondent ces méthodes soit un matériel techniquement dépassé et dont la construction est maintenant abandonnée, et d'apporter ainsi notre contribution à l'édification de methodes modernes de programmation.

Il s'agissait de donner aux chercheurs scientifiques qui travaillent au Centre de Calcul ou qui ont l'intention d'utiliser l'ordinateur du Centre, une méthode de programmation simple, et d'éviter autant que possible le cumul des difficultés propres à toute recherche scientifique ou technique et des difficultés de la programmation. Cette commodité d'emploi doit exister, non seulement lors de l'écriture du programme, mais également lors de sa mise au point ou de ses modifications éventuelles. Elle se paie évidemment par un alourdissement du temps de calcul, inconvénient secondaire pour un Centre de recherches.

Le point de départ de nos recherches fut le FLAIR, logique extérieure mise au point par la Compagnie I.B.M. pour l'ordinateur I.B.M. 650. Le FLAIR comporte essentiellement les opérations en virgule flottante, les fonctions élémentaires et quelques ordres logiques. L'instruction est numérique, à dix chiffres.

I. Le F. L. E. C.

Une première extension fut le FLEC, logique extérieure, où les instructions sont exécutées au fur et à mesure du décodage, destinée aux calculs en arithmétique complexe et plus particu-

lièrement aux calculs d'impedances de dipôles, d'affaiblissement de quadripôles, etc...., intervenant dans l'étude de réseaux électriques.

L'ordre du FLEC est un ordre numérique à 10 chiffres, précédé d'un signe, à 2 ou 3 adresses. Tous les nombres sont écrits en virgule flottante, le nombre complexe $a' + i\,a''$ mémorisé en A, occupe en fait les mémoires A (partie réelle) et A + 1 (partie imaginaire). Nous trouvons des instructions telles que:

- 1 A. B. C.

addition des nombres réels places en A et B; résultats en C; A. B. C. sont des adresses à 3 chiffres.

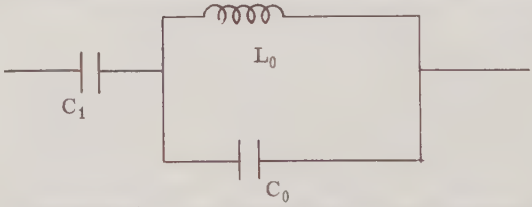
+ 1. A. B. C.

addition de $(a' + i\,a'')$ et de $(b' + i\,b'')$. a' est en A, a'' en A + 1; b' en B, b'' en B + 1, le résultat sera placé en C et C + 1.

Le FLEC comporte également des instructions logiques (saut, lecture, perforation); mais l'intérêt du FLEC réside en particulier dans les ordres speciaux destinés au calcul des réseaux électriques, tels que l'ordre:

0. 019. BBB. CCC

qui calcule l'impédance du dipôle schématisé ci-dessous:



supposant la capacité C_1 en mémoire B, la capacité C_0 en B + 1, la valeur ω_0 de résonnance de la self en B + 2 et qui placera l'impédance (nombre complexe) en C et C + 1.

Il existe des ordres permettant ainsi le calcul des impédances de tout circuit formé de 4 éléments au plus. Pour associer les impedances de circuits en parallèle nous disposons de l'ordre:

5. A. B. C.

qui effectue l'opération

$$\frac{1}{a' + i\,a''} + \frac{1}{b' + i\,b''} = \frac{1}{c' + i\,c''}$$

et place $c' + i\,c''$ en C, C + 1. Il existe de plus des ordres spécialement adaptés aux calculs de quadripôles.

Nous n'insisterons pas plus sur le FLEC, décrit dans une notice I.B.M. et nous nous contenterons de signaler l'existence d'une analyse au niveau FLEC, par simple affichage d'un signe moins au pupitre.

II. Les CODES de PROGRAMMATION

Le FLEC fut suivi des Codes de Programmation, destinés aux problèmes scientifiques généraux.

Choix des Sous-programmes

Pour obtenir un outil de travail valable dans des cas très généraux, il nous a fallu prévoir:

(1) Les calculs en virgule fixe permettant un gain de temps considérable par rapport aux calculs en virgule flottante, donc des ordres concernant en particulier les calculs de fonctions élémentaires.
(2) Les calculs en virgule flottante, simple précision.
(3) Les calculs én precision multiple.
(4) Les calculs matriciels.

De plus, il fallut prévoir des sous-programmes logiques, simulation d'index, ordres de bouclage, test, ...

La plupart des sous-programmes existaient déjà, il a suffi de les adapter. Il a cependant été necessaire de creer les sous-programmes correspondant aux principaux ordres logiques ainsi qu'à la multiple précision: citons en particulier les opérations en double et triple précision, avec une caractéristique à 2, 3 ou 4 chiffres au gré de l'opérateur. On peut ainsi travailler sur des nombres de 26 chiffres, compris entre 10^{-5000} et 10^{5000}.

Structure des Instructions

Les instructions du FLAIR ou de FLEC, limitées à 10 chiffres décimaux, ne contenaient pas une "quantité d'information" suffisante pour créer les ordres à 3 adresses dont nous avons besoin. Comme les mémoires de l'I.B.M. 650 contiennent 10 chiffres décimaux, il fallait associer 2 mémoires par instruction, qui peuvent alors comporter 20 chiffres. L'instruction normale à la structure suivante:

xx	xxx	x	xxxx	x	xxxx	x	xxxx
OP	N	j	A	j	B	k	C

Le premier groupe de 2 chiffres correspond au code opération, le second groupe donne l'adresse de l'instruction suivante (qui occupera les mémoires N et N + 1) adresse nécessairement inférieure à 1000.

Les groupes suivants de 5 chiffres sont des adresses qui peuvent être indéxées: si le premier chiffre i, j, k est nul, l'adresse n'est pas indexée, si non, i, j ou k représente le numéro du registre d'index. On a pu simuler 9 index.

Il existe de plus 2 instructions à *3 mémoires*, correspondant aux opérations de multiplication et division en virgule fixe avec décalage, ainsi que des instructions de service à une mémoire, telles que l'instruction d'initialisation (remise à zéro des registres d'index, ...) de lecture, de perforation simple, d'arrêt, etc. ...

Parmi les instructions logiques, nous trouvons par exemple:

Le Test à 3 Branches

07 . N . 0 . A . 0 . B . k . C .

qui choisit comme "instruction suivante" l'instruction placée en

N et N + 1	si le contenu de C (indexé)		est négatif
A et A + 1	"	"	est nul
B et B + 1	"	"	est positif

c'est l'équivallent de l'ordre SI du FORTRAN.

L'Ordre de Bouclage

9 i . N . α . H . 0 . B . 0 ; C .

qui commande executer (ou FAIRE) H fois ($H \geq 1$) le programme qui commence en B en ajoutant à chaque fois C (et non son contenu) au contenu du registre d'index n° i et en utilisant le compteur de boucle n° α. A la sortie de boucle, aller en N. Ordre analogue à l'ordre FAIRE du FORTRAN.

Le saut conditionnel par test de 8 ou de 9 (test imposé par la technologie de l'I.B.M. 650).

08 . N . 0 A . 0 B . k C

Cette instruction amene le contenu de la mémoire C (indexée) dans le distributeur puis fait le test SD. B sur la position B de

ce distributeur. S'il y a un 8, le programme est aiguillé en N; S'il y a un 9, le programme est aiguillé en A; S'il n'y a ni 8, ni 9, il y a arrêt.

Procédé d'Emploi

Dans les langages symboliques, nous trouvons deux familles: les "logiques extérieures" et les "auto-programmations".

Dans le premier mode, le langage symbolique est traduit, puis exécuté; le programme symbolique n'est pas détruit, c'est le programme machine qui est perdu après exécution.

En "auto-programmation", la machine traduit le langage symbolique et crée le programme machine équivalent. Le langage symbolique sera perdu et le programme machine enregistré sur bande ou carte en vue d'une exécution ultérieure. L'auto-programmation correspond au travail d'un programmeur pour qui le langage symbolique est un organigramme détaillé.

Le procédé de logique extérieure est d'une grande commodité pour la mise au point et pour des modifications éventuelles de programme car il permet à chaque instant de faire l'analyse d'une partie du programme, sans avoir eu besoin de prévoir cette analyse lors de l'écriture du dit programme. Ce procédé paie sa commodité par une certaine perte de temps lors de l'exécution des calculs.

Le procédé d'auto-programmation permet une exécution plus rapide des calculs, mais rend toute correction et toute modification au programme plus délicates.

Ces deux procédés sont complémentaires, et les codes de programmation ont été prévus pour que le même programme puisse être au gré de l'opérateur, utilisé en logique extérieure lors de la période de mise au point, puis transformé en programme machine par auto-programmation.

Organisation Générale des Codes de Programmation

L'ensemble des sous-programmes nécessaires au calcul scientifique est trop important pour être groupé en une seule logique extérieure et placé sur les mémoires du tambour; il a fallu prévoir plusieurs versions des codes de programmation. Les codes de programmation comportent alors:

(1) *Un tronc commun* comportant:

—la phrase interprétative;

—les ordres d'opération en virgule fixe;

—les ordres de routines;
—quelques ordres logiques (tests, bouclage, perforation).

(2) A ce tronc commun on associe selon les besoins les sous-programmes necessaires et les codes correspondants.
Nous disposons des variantes suivantes:
—Code de Programmation en virgule fixe;
—Code de Programmation en virgule flottante, simple précision;
—Code de Programmation en double précision } à caractéristique variable
—Code de Programmation en triple précision }
—Code de Programmation matriciel.
On peut bien entendu:

(1) Complèter un code de programmation par d'autres sous-programmes, par exemple ajouter l'ordre d'inversion de matrice au code de programmation en virgule flottante.

(2) Créer un code de programmation correspondant à une famille de problèmes bien définis (Création prochaine d'un C.D.P. orienté vers les calculs de résistance des matériaux).

(3) Adapter le code de programmation à des I.B.M. 650 non standard, à virgule flottante câblée et à index.

Remarques sur l'Emploi des Codes de Programmation

Les codes de programmation sont utilisés au Centre de Nancy depuis 2 ans, tant par les étudiants du 3ème Cycle de Maths que par les chercheurs qui viennent travailler en self-service. On peut tirer de cette expérience les conclusions suivantes:

(1) Le procédé en "logique extérieure" est pratiquement le seul utilisé; il est très rare qu'un étudiant utilise le C.D.P. en auto-programmation.

(2) Le gain de temps en écriture et mise au point par analyse est considérable. Il y a en particulier gain important de temps machine car cette analyse évite toute mise au point au pupitre.

La perte de temps lors de l'exécution d'un calcul, due à l'emploi d'un langage symbolique, dépend du problème traité. Pour le Code de Programmation, les ordres de grandeurs sont les suivants:

TEMPS D'EXECUTION D'UN CALCUL

	Programme écrit directement au niveau machine	Programme en C.D.P. auto-programmé	Programme en C.D.P. (logique extérieure)
Problèmes en double précision. Problèmes matriciels.	1	1	1
En virgule flottante simple précision.	1	1.5	2
En virgule fixe			
Peu d'adresses indexées	1	2	4
Nombreuses adresses indexées	1	4	8

Sur le plan enseignement, on peut considérer le C. de P. comme première prise de contact avec la programmation, qui dégage l'étudiant des problèmes mineurs et technologiques de la programmation pour ne lui laisser que les difficultés essentielles: conception et organisation du programme. Il est prévu pour cet enseignement une durée de 2 mois à raison d' 1 heure de cours et de 2 séances de travaux pratiques par semaine. On peut également se proposer d'enseigner le C. de P. à des programmeurs de metier; l'expérience faite à Nancy a montré qu'il suffit de 2 conférences et d'une séance de mise au point sur machine.

III. Le C. P. A. S.

Le C. P. A. S. ou Code de Programmation à Adresses Symboliques est une nouvelle écriture des codes de programmation, utilisant des codes opérations et des adresses symboliques qui, d'une part se présentent sous une écriture alpha numérique plus commode à lire et à retenir et dont la machine elle-meme, lors d'une phase d'assemblage, assure la correspondance avec un code-opération ou une adresse numérique.

Les ordres ont alors la structure suivante:

où COD:OP est le code opération.

A:LPHA
B:ETA
G:AMMA
S:UITE
} sont des adresses, éventuellement "blanches"

i. j. k. sont les index écrits sous forme numérique.

Tout code ou toute adresse est formé de cinq signes alphanumériques (chiffres, lettres ou "blanc") suivant les conventions du PASØ.

Une table interne du programme d'assemblage établit une correspondance bi-univoque entre les codes opérations alphanumeriques et les codes opérations numériques du C. de P.

Une table qui se construit au fur-et-à-mesure de l'assemblage établit, lors de la première rencontre d'une adresse symbolique, une correspondance entre cette adresse symbolique et une adresse absolue, correspondance qui sera conservée tout au long du problème.

Nous trouverons des ordres en C.P.A.S. tels que les suivants:

Opération à 3 Adresses

exemple:

qui se lit: multiplication en virgule flottante simple précision du contenu de la mémoire RX4 par le contenu de la mémoire SINR2; placer le résultat en mémoire R S2 indexee par le registre d'index numéro 1; construction suivante en ORD 7.

Ordre à 2 Adresses

qui s'interprète: calcul du cosinus, en virgule flottante, du nombre placé en X (supposé en virgule flottante), placer le résultat en mémoire COS X; instruction suivante en ORD8.

Supposons que nous rencontrions le premier de ces deux ordres, lors de l'assemblage, et que R: X4 ait été attribué à l'adresse 102, que SINR 2, RS2 et ØRD 7 n'aient pas encore été rencontrés, que la première mémoire disponible soit 200. Le programme d'assemblage fera correspondre à MUL L1 le code correspondant 11, il remplacera RX4 par 0102; attribuera à SINR 2, l'adresse 200 qui lui sera réservée, a RS2 l'adresse 201 et les suivantes, selon le nombre qui aura été indiqué dans

```
          RESAB    0277     0286
          EQISB    SIGMA    0277
          RESRG    I1967    1977
          RESRG    L0001    0231
          RESRG    A0232    0252
          RESRG    B0253    0273

                                                                  287 69 028 80289 0000000290 2418841800
DEBUT     INITL                                                   290 02 0000 0291-
          ZERO2    SIGMA                                          291 11 0277 0292-
          MODIF    MP1      BC1                                   292 00 297 01937 0029400295
          MODIF    NP1      BC2                          BCL1     297 00 302 01937 0029900300
BCL1      MULFX    I0001    MP1      I0004                        302 03 304 01968 0029401971
          ZERO2    SJ                                    BCL2     304 11 0305 0307-
BCL2      ADDFX    I0004    I0002    I0003                        307 01 309 01971 0196901970
          MULL2    2B0001   3L0001                                309 23 311 20253 3000100274
          ADDL2             SJ       SJ          BC2             311 21 295 00274 0030500305
BC2       FAIR2    0        BCL2      0002                        295 92 313 00000 0030700002
          MULL2     SJ      1A0001                               313 23 315 00305 1023200274
          ADDL2     SIGMA             SIGMA       BC1            315 21 300 00277 0027400277
BC1       FAIR1    1        BCL1      0002                        300 91 317 10000 0030200002
          PERFO     SIGMA                          9876          317 06 0277 9876-
MP1       C  00                 4                                294 00 000      4
NP1       C  00                 3                                299 00 000      3
```

le pseudo-code, 7 par exemple, de réservation de memoires indéxées et attribuera a ORD 7 l'adresse 208 et réservera 208 et 209. L'ordre C.P.A.S. donnera naissance à l'ordre:

01	208	0	0102	0	0200	1	0201

Le mécanisme d'assemblage est un peu différent pour un code à 2 adresses (calcul d'une fonction) le programme d'assemblage fait correspondre au code COS N L

(1) Un code 00 en OP

(2) Un code 1004 en position "A" du code de programmation. Si les mémoires X, COS X et ØRD 8 correspondent respectivement à 103, 175, 210, l'ordre du C.P.A.S. sera transformé en:

00	210	0	1004	0	0103	0	0175

L'usage d'adresses en "blanc", c'est-à-dire dont les colonnes correspondantes de la carte ne contiennent aucune perforation allège l'écriture et le lecteur trouvera en annexe de ce rapport l'exemple d'un problème écrit en C.P.A.S. ainsi que le programme en C. de P. obtenu par assemblage.

Il faut rappeler que le C.P.A.S. a été prévu pour un ordinateur I.B.M. 650 standard, sans dispositifs "spéciaux", qui, en particulier, n'accepte pas les signes tels que +, −, :, ×, ( ), il est impossible d'espèrer une écriture aussi évoluée et aussi proche de l'écriture mathématique que des écritures telles que FORTRAN ou AP3.

Moins spectaculaire que ces derniers, le C.P.A.S. possède peut être plus de souplesse dans le choix des adresses symboliques et des codes opérations. C'est ainsi que nous avons 6 codes de multiplication, alors qu'il n'existe qu'un (ou deux) signes de multiplication.

MUL FX: multiplication en virgule fixe sans décalage;
MUL $\lambda\mu$: multiplication en virgule fixe avec décalage (le décalage est caractérisé par les chiffres μ et λ)
MUL L1 : multiplication virgule flottante simple précision;
MUL L2 : " " double précision;
MUL L3 : " " triple précision;
MUL MA: " de 2 matrices.

et s'il faut pour calculer une expension telle que:

$$\text{RAC } (B^{XX} - 4\,AC) \quad : \quad \text{SIN } (1 : E)$$

un grand nombre d'instructions, cet inconvénient est peut être compensé par la simplicité de chaque instruction, l'absence de règles strictes sur la position des parenthèses et des signes de division successifs, ainsi que le choix d'addresses plus "parlantes".

$$\text{CALCUL de } \sum_{i=0}^{n} \sum_{j=0}^{m} a_i\, b_j\, l_{ij}$$

Programme Écrit en C.P.A.S., Virgule Flottante Simple Précision

Les valeurs a_i, b_j et l_{ij} sont appelées à l'aide des registres d'index 1, 2, et 3, d'adresses régionales I0001, I0002, I0003, l'adresse de l_{ij} étant: $i(m + 1) + j$.

L'ordre INITL: remet à zéro tous les registres d'index et les compteurs de boucle avant le calcul.

L'ordre ZERØ 2: réserve et remet à zéro deux mémoires consécutives qui sont les équivalents de SIGMA ou SJ.

Les ordres FAIR I et FAIR 2: sont des ordres de bouclage. Ils demandent faire (m + 1) ou (n + 1) fois la séquence de calcul qui commence en BCL 2 ou BCL1, tout en ajoutant 0002 aux registres d'index 2 ou 1. Après exécution de ces boucles, les registres sont automatiquement remis à zéro. Le décompte du nombre de boucles s'effectue dans les compteurs n° 0 et n° 1.

Les nombres de boucles à effectuer peuvent être variables, alors les ordres MØDIF, modifient les ordres de bouclage donnés sans indications spéciales, en leur indiquant (m + 1) ou (n + 1).

Les opérations sur les registres d'index se font en virgule fixe (ADDFX). Celles sur les valeurs numériques données en virgule flottante double précision (MULL2).

Les adresses régionales et absolues suivent les mêmes conventions qu'en écriture PASØ, d'où les pseudo-codes qui affectent à chaque région une zone du tambour. En particulier, le C. de P. place les registres d'index dans la séquence 1967-1977. Un équivalent en zone de sortie est imposé à SIGMA pour per-

mettre un ordre de perforation simple: PERFØ. L'arrêt calcul est provoqué par l'adresse impossible 9876.

Le programme commence seulement en 0287, en effet, les trois premières mémoires libres: 0274.5.6, sont réservées au départ comme mémoires de travail affectées aux adresses facteurs "en blanc". Les ordres d'entrée en code (0287.288.289) ont été ajoutés à l'assemblage, alors qu'ils n'existent pas sous forme symbolique.

Il serait facile de transformer ce programme pour utilisation en triple précision, il suffit de remplacer les ordres MULL2, ZERO2, par MULL3, ZERO 3, des réservations de 3 mémoires par symbole seront faites automatiquement à l'assemblage.

BIBLIOGRAPHIE

Le F.L.E.C.

1. These de 3ème cycle de Mr. F. Thomas—Juin 1960: Etude Detaillée du "FLEC"; Application de la Méthode de Modelage au Calcul d'un Correcteur.
2. Notice I.B.M.

Les CODES de PROGRAMMATION

1. Thèse de 3ème cycle de Mme. Crehange, assistante à la Faculté des Sciences, Mars 1961: Structure du Code de Programmation.
2. Thèse de 3ème cycle de Mr. C. Colas—Juin 1961: Opération en Triple Précision - Code de Programmation - Moments Centres, Courbe de K. Pearson.
3. Groupement des Utilisateurs Scientifiques des Ordinateurs I.B.M. 650 Cahier n° 1.

Le C.P.A.S.

1. Thèse de 3ème cycle de Melle. Leonard, assistant à la Faculté des Sciences, Juin 1961: Calculateur à Arithmétique Binaires—C.P.A.S. —Affectation des Mémoires.

PROBLEMS OF PROGRAMMING SYSTEMS

PROBLEMS IN PROGRAM INTERCHANGEABILITY

J. H. Gunn

The University of Birmingham, England.*

1. INTRODUCTION

With the ever increasing numbers of computers coming into general use, it is becoming increasingly important to find means whereby existing programs can be adapted for use on the new machines. It is obviously advantageous to avoid wasting valuable programming time in making this change-over so as to leave the programmer free to tackle new problems. One of the ways in which programs can be made 'interchangeable' in the sense that they can be run on succeeding generations of machines, is to write them in one of the existing 'Autocodes', such as 'Mercury Autocode', 'Fortran' or 'Algol'.

But not all programs are written in autocodes, and not all programs that are, keep to the rules that are given. Many programs are written in basic machine code or in autocode plus some machine code and as a result depend very much on the machine structure of the computer in question. One short example suffices to illustrate this point:

Mercury possesses two 'test-registers' as distinct from 'B-registers' in which integer arithmetic is performed. These test registers are used by some conditional jump instructions, for which purpose the B-register cannot be so used. This does not occur in Orion basic machine code: in this case there are no separate B-registers or test registers, tests being performed on the ordinary registers. Thus to achieve the same logical outcome, i.e., that of the conditional jump, two different techniques are employed.

Hence in order for a program written in one basic computer to be translated into a second computer code, the logical content must be extracted from the program and re-coded in the second language. We call this 'program-interchangeability'; the problems that arise in doing this are considered here. To demon-

*Now at Nordisk Institut for Teoretisk Atomfysik, Copenhagen, Denmark

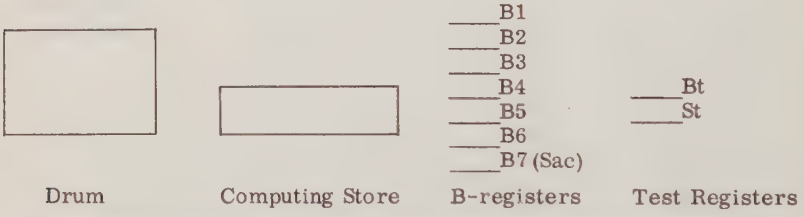
strate the kind of problems that arise in such work, the particular case of converting a Mercury machine language into one that will run on Orion is considered in some detail.

2. THE TWO COMPUTERS

Mercury was chosen as the 'source' computer, partly because of its familiarity but principally because of its ready availability for testing the ideas expressed here. Orion was chosen as its partner because of its dissimilar structure to Mercury and for the fact that there is some practical importance in converting some of the existing Mercury programs into ones that will run on Orion. From the point of view of more clearly demonstrating some of the points stated here it is advantageous that the computers be as different as possible, as in this case.

2.1 The Mercury Computer

The storage of Mercury can be represented by the following diagram:



There are three different word lengths which are used in Mercury. The first is of 40 bits for the representation of floating point numbers, the second of 20 bits for the storage of instructions and the third of 10 bits for the representation of integers. Arithmetical operations on 40 bit words must take place through the accumulator and must specify the address of some register in the computing store. This latter address may be modified ('B-modified') by the contents of one of the B-registers if desired. In addition arithmetical and logical operations may be performed on 10-bit words through the B-registers for the purposes of counting and modification etc. In this case a B-register and an address in the computing store are specified. The seventh B-register (or 'Sac' as it is more commonly called)

is different from the other six B-registers in that a special set of instructions are available which allow modification of the specified addresses.

Two test registers are provided—the 'B-test register' (Bt) and the 'Sac-test register' (St); which are set when certain B-register or Sac instructions are obeyed. These are used by the conditional jump instructions since these do not test the contents of the B-registers directly.

The following are examples of Mercury instructions:

(a) 400 10
"transfer the contents of register 10 to the accumulator".

(b) 406 12
"transfer the contents of register 12 modified by the contents of B-register 6 to the accumulator".

(c) 430 18
"subtract the contents of register 18 from the contents of the accumulator (floating point) and leave the result in the accumulator".

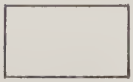
(d) 026 13+
"add the contents of the (10 bit) register 13+ to B-register 6 and leave the result in B6 and Bt".

(e) 213 11
"transfer the contents of sac to the (10 bit) register 11 modified by the contents of B3".

(f) 280 1v16
"if St $\neq$ 0 jump to the instruction after the one labelled 16, otherwise jump to the next instruction".

2.2 The Orion Computer

The Orion Computer can be represented in an equivalent manner by the following diagram:


Drum


Computing Store

In this case there are no separate accumulators, B-registers or test registers. The word length for Orion is 48 bits which can be split into eight 6-bit 'characters'. Apart from certain character manipulation instructions there are only arithmetic

operations on the full word-length of 48 bits in fixed or floating point modes. An Orion instruction has the following written form:

INSTRUCTION CODE NUMBER, X-ADDRESS, Y-ADDRESS, Z-ADDRESS

If a quantity is present in the Z-address position, the instruction is said to be in the 3 address form, if it is absent in the 2 address form. In Orion arithmetical or logical operations do not go through an accumulator but instead the result of operating on X and Y may be placed in Z in the three address form and in X in the two address form. Modification of the X and/or Y address by the contents of Z is permissible. There are no test registers, conditional jumps being made by comparing the contents of two registers.

The following are examples of Orion instructions:

(a) 04 A10 A17
"transfer the contents of register A17 to register A10".

(b) 00 A11 A14 A16
"add the contents of register A11 to register A14 and leave the result in register A16". (fixed point addition)

(c) 90 A15 A19
"add the contents of A19 to the contents of register A15 and leave the result in register A15". (floating point addition)

(d) 60 6+ A19 A20
"jump to the instruction 6 further on if register A19 contains the same as register A20, otherwise carry on with the next instruction".

3. METHOD OF TRANSLATION

3.1 Dummy Registers

It is convenient to set aside seven Orion registers to imitate the properties of the Mercury B-registers, these are designated A1-A7 and correspond to B1-B7. In addition, A8 represents the Mercury accumulator and is designated ACC in the examples; A9 represents Bt (designated BTEST); A10 represents St (STEST) and A11 is used as additional working space (WKSP).

3.2 Translation of the Individual Function Codes

In the translation program each Mercury instruction is considered separately and the equivalent Orion instruction (or instructions) constructed.

3.3 Instructions Representable by One Orion Instruction

40 Instruction.

400 L "transfer the contents of register L to the accumulator" $A' = L$

becomes quite simply:

04 ACC L

the B-modified form of this:

40B L "transfer the contents of register L modified by the contents of the B-register to the accumulator" $- A' = L_b$.

becomes:

04Y ACC L B

50 Instruction.

500 L "multiply the contents of the accumulator by the contents of register L and leave the result result in the accumulator" $A' = A \times L$

translates to:

94 ACC L

with a corresponding form for the modified instruction.

3.4 Instructions Not Representable by One Orion Instruction

The above example shows that it is possible to make a one-to-one translation of certain of the Mercury instructions into Orion instructions. Unfortunately, it is not always possible to do this because of the different natures of the two instruction codes. The simplest example of a Mercury instruction that requires two Orion instructions to represent it, is the 51 instruction (which has the effect of multiplying the accumulator negatively by the contents of some register)—this has no counterpart in

the Orion order code; instead this must be translated as follows:

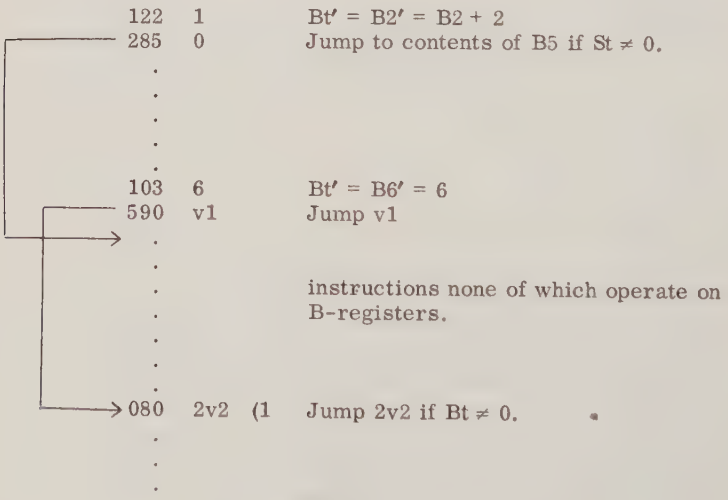
```
510  L          A' = A × L
```

becomes:

```
94  ACC  L      A' = A × L
93  ACC  ACC    A' = −A
```

where the '94' instruction is a floating point multiplication and '93' a simple negation instruction.

Other examples which require two or more instructions, are those instructions which perform arithmetic on the B-registers. Because the B-registers have a test register associated with them, the contents of which is in general the contents of a B-register after a particular instruction has been obeyed, an extra Orion instruction is necessary to set the dummy test register that is being used instead. It might be argued that it would be preferable to avoid dummy test registers altogether and use the dummy B-registers directly for performing the conditional jump instructions—but this gives rise to certain difficulties, the solution of which other than by retaining dummy test registers is not clear; for example consider the following piece of program:



If one tried to compile the equivalent Orion program without using test registers it would be necessary to relate the '08' instruction to the B-register it was testing. It is not possible without actually running the program, to know that the '08' instruction is testing the contents of B2 and not B3. Although 'statically' (that is in the order of the instructions on the program tape) the instruction altering the value of B3 occurs after the instruction altering the value of B2; 'dynamically' (that is in the order that the instructions are obeyed) the instruction altering the value of B3 is not obeyed in the branch of the program that is being considered. Thus since the program is being translated statically and not being compiled as the instructions are being obeyed, it would be difficult to translate conditional jump instructions such as '08' without using a test register.

The problem of dealing with program quantities, the dynamic values of which determine some course of action, and which can only be allowed for statically, is the one that gives rise to most of the difficulties of making up a compiler; this will become clearer when problems of 'addressing' are dealt with.

A typical B-register instruction requiring two Orion instructions to represent it is:

12B n	"add the integer n to the B-register and reset the B-test register to this value"
	$Bt' = B' = B + n$.

this becomes:

10 B n	$B' = B + n$
05 B MASK BTEST	$Bt' = B \,\&\, MASK$

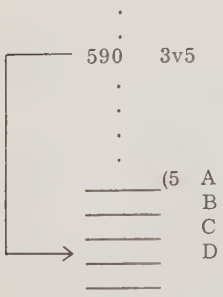
Note: the logical instruction 'collate' is used to ensure that the contents of the dummy B-test register are only 10 bits in length—this is one of the devices necessary to ensure that there is no trouble due to the different word lengths in the two machines.

4. ADDRESSES

4.1 Relative and Symbolic Addresses

Each routine of a Mercury program has a labelling system of its own, and this can be carried over into the equivalent Orion program. Addresses which consist of a pure label may be left

as such and references to this label will be translated in the obvious way. But suppose an instruction of the following form was encountered:



Here 3v5 means 'the third instruction after the one labelled 5'—this is the instruction D. On translation, instruction A will still be labelled 5 but instruction D will not necessarily be the third after the one labelled 5 in the translated program because some of the instructions may require more than one instruction to represent them. This difficulty is solved by adding to the integer 3 the number of extra instructions introduced by the translation. As the Mercury source program is read in by the translator and each instruction translated and stored, sufficient information is tabulated so that the addresses can be filled in when all the program has been read in. Obviously the addresses can not be filled in as the program is translated instruction by instruction because of references to parts of the program not yet encountered.

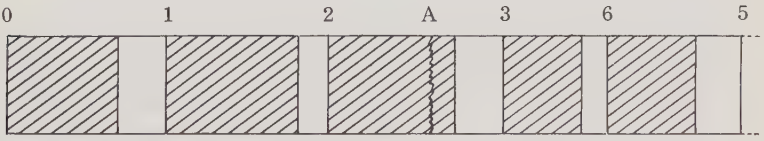
4.2 Absolute Addresses

Absolute addresses (i.e. addresses that refer to fixed locations in the store irrespective of the program) can refer to instructions or data in the first half of the computing store and to data only in the second half. (This restriction arises because of ten bits only available for addressing on Mercury.) Thus addresses in the first half will be constructed by a similar technique to that employed for symbolic addresses; but addresses in the second half will merely have a constant added since there will only be data which translates one-to-one in terms of registers.

5. PROGRAM STRUCTURE

A Mercury program is split into 'Chapters', one of which is held in the computing store at any one time. A Chapter may, if desired, be broken down into 'Routines'—each having its own independent labelling scheme. Both Chapters and Routines have their equivalents in Orion programs which perform the same duties as in Mercury programs. Unfortunately, one has to deal with the possibility of part of a Chapter being transferred from the drum to the computing without destroying the remaining part of the computing store. Mercury drum transfer instructions transfer blocks of 64 instructions called 'sectors', into 'pages' of the computing store. Since a block of 64 Mercury instructions will translate into considerably more than 64 Orion instructions, the number dependent on the nature of the original Mercury instructions, it is necessary to make special provision in the compiler if this facility is to be allowed.

This difficulty can be avoided if a block of Orion instructions of a definite length is allocated to each translated page of instructions. This will mean that there will be spare space at the ends of most of the Orion blocks and the addresses will have to be adjusted to take this into account. We can represent the situation in the following way:



The shaded areas represent the presence of instruction.
Unshaded areas represent spare registers.

Suppose we wish to correct the address A, which has already been calculated assuming that there are no gaps: we first compare A with a table showing in which block A lies (in this case 2) and add in the correction from the table.

The advantage of this procedure is that the compiler can detect the minimum block length required to fit the largest block of translated program. This cuts down the space required for the program and also shows when this procedure is not required: i.e. when no drum transfer of instructions take place.

6. INSTRUCTIONS REFERRING TO HALF REGISTERS

A particular half register in a Mercury program will be represented in a variety of ways depending on the function of that half register in the original program. Consider a block of ten bit registers containing data being processed in some way, then this will have a corresponding block of (12 bit) registers in the Orion program.

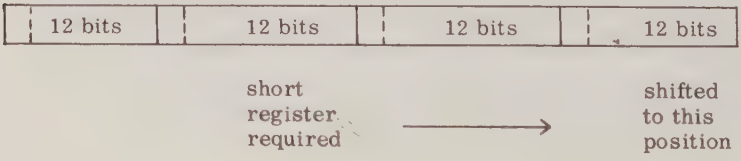
A half register in a Mercury program is, in fact, one quarter of the standard word length of 40 bits (i.e. 10 bits); the corresponding length in Orion is 48 bits. Hence it is convenient to use one quarter of this to represent a half register. The two extra bits will be redundant and great care must be exercised so that the extra bits have no effect on the running of the program. This statement is made because it is possible, in some cases, from the way the translator is constructed to have the two most significant bits non-zero. Thus if this number was transferred directly to a test register and tested for zero it could happen that a wrong course of action was taken. For this reason the content of a half register or B-register are placed in the appropriate test register by collating with a mask.

Since there are no analogous instructions to the B-register instructions of Mercury, arithmetic on 10 bits presents special difficulties, an outline of which will be given below.

Trouble will again arise when the half register under consideration is in the address part of an instruction. On translation this address is likely to be in the 'X' position or the 'Y' position of an Orion instruction and hence it is necessary to ensure that the correct position is operated on.

In the translated form of most short register instructions an Orion 'shift' instruction must be included to bring the correct portion of the word containing the short register into line with the 10 least significant bits of the B-register concerned.

Suppose the short register required was in the Orion word in the position indicated—it is necessary to transfer the contents to some B-register.



Thus a shift to the right 24 places is required to bring the short register to the correct position.

To perform the reverse operation the contents of the B-register must be placed in a temporary register, shifted to the correct position and then inserted into the Orion word.

In the event of there being no shift required, the shift instruction is omitted by the compiler.

Example.

```
OOB  H      'copy the contents of the half register H into
            the B-register and the B-test register'—
            Bt' = B' = H.
```

this becomes:

```
53  H  12i   WKSP      shift
04  B  WKSP            B' = H
05  B  MASK  WKSP      Bt' = H
```

the shift is 12i places to the right, where i depends on the section of the word (i.e. on H)—this is omitted if not necessary.

7. THE TRANSLATION PROGRAM

The translation program is being developed to run on Mercury but it is hoped that the program may itself be used as data to produce a version that will run on Orion. This will also be a practical test of the scheme outlined in this paper.

For convenience, the program has been broken down into sections. The first is the input program which is a much modified version of the Ferranti Input scheme which reads in one instruction at a time and stores in a convenient form. The second section of the program converts the instruction into the equivalent Orion instructions, stores these on the drum and updates the various tables used for filling in the addresses. After all the source program has been dealt with in this way the third section deals with the addresses, a fourth section prints out the completed Orion program.

7.1 Removal of Redundant Test Register Instructions

Splitting up the translation program has the additional advantage that refinement can be made at any stage of the transla-

tion procedure. For example, consideration is being given to the elimination of unnecessary test register instructions which arise in the following way:

The piece of program

220 H	St′ = S′ = S + H
320 7	St′ = S′ = S + 7

translates to

53 H 12i WKSP	shift
00 SAC WKSP	S′ = S + H
05 SAC MASK STEST	St′ = S
10 SAC 7	S′ = S + 7
05 SAC MASK STEST	St′ = S

The first test register instruction is redundant in both the cases where the Mercury instruction pair is entered at either the first or the second instruction and can be eliminated before the addresses are filled in by making the necessary corrections to the address tables. The first test register instruction is also redundant if there are other instructions between the two instructions shown, provided that these instructions do not alter the test register or jump to another branch of the program.

7.2 Removal of Unnecessary Shift Instructions

It was stated in section 6 that the Orion shift instructions arising because of operations on short registers could sometimes be omitted if they were not required. This must be done after the addresses have been constructed because the amount of 'shift' required depends on the half register concerned. If the instruction can be omitted, the necessary alterations to the program and to the address tables are made.

7.3 Condensation of Arithmetic Instructions

Consider the following sequence of Mercury instructions:

400 A	ACC′ = A
500 B	ACC′ = ACC × B
410 C	C′ = ACC

Using the translation scheme outline, this would require three Orion instructions to perform the same operations:

04 ACC A	ACC′ = A
94 ACC B	ACC′ = ACC × B
04 C ACC	C′ = ACC

If the sequence of instructions is never entered at either the second or third instruction, it would be possible to condense the three Orion instructions into two, in the following manner:

94 A B ACC	ACC′ = A × B
04 C ACC	C′ = ACC

Note: one cannot use the single instruction

94 A B C

because this does not leave the result of the multiplication in the accumulator, which may be required by later instructions.

Sequences of this type are sufficiently common to make it worth while condensing them into a more compact form. The major difficulty that must be overcome is that one can never be completely sure that the above sequence is not entered at the second or third instruction. Work is at present proceeding on the problem of formulating a set of rules which will enable the translator to discriminate between those parts of the program that can be condensed and those which cannot.

Programs in which a significant amount of condensation takes place will make a much more efficient use of the Orion order code which will partially compensate for the additional Orion instructions required to deal with short registers.

8. CONCLUSIONS

By the methods outlined in this paper, most of the facilities available for a Mercury program are convertable into a form acceptable to an Orion computer, with the exception of one or two tricks: for example altering the contents of some register known to contain an instruction into another instruction during the running of the program—a trick that requires detailed knowledge of the function codes internal to the machine. Fortunately this device is rarely used and can be excluded

without seriously limiting the use of the translator. However it must be pointed out that in principle no restriction on the program has been found that could not be removed if sufficient programming effort was undertaken and a lengthy interpreter subroutine included in the Orion program to deal with the facility.

It was hoped that when this project was started that a mechanical translator of this nature would be of practical importance but it must be admitted that because of the considerable time losses due to the necessary inclusion of additional instructions, it is doubtful that it will be very useful in this way.

However, this project has pointed out some of the difficulties involved in converting programs written for one computer into those acceptable to another, and has shown that an order structure of the Orion type is too unwieldy for performing efficient translations of the type outlined. If ready conversion of programs from one machine to another is envisaged as a practical proposition, then consideration must be given to the order codes of computers. Programs continue to be written in basic machine codes because even now the efficiency of any autocode compiler is not so great as to preclude it; and while this remains so it would seem sensible to consider problems of the type outlined in this paper.

ACKNOWLEDGMENTS

I would like to thank Mr. K. A. Redish for the many helpful discussions, comments and criticisms during the course of this work and to D.S.I.R. for financial support.

REFERENCES

1. Programming manual for Ferrnati Mercury Computer (Ferranti CS 158, July 1957).
2. Programmers Handbook for Ferranti Mercury Computer (Ferranti CS 225, Nov. 1958).
3. Annotated Input Scheme for Ferranti Mercury Computer (Ferranti CS 240, July 1959).
4. Provisional Description of Ferranti Orion Computer System (Ferranti CS 264D January 1961).
5. Ferranti Orion Computer - Primitive Input Scheme.
6. Ferranti Orion Computer - The Built In Programs (Ferranti CS 279 January 1961).
7. Ferranti Orion Computer - Summarized Programming Information (Ferranti CS 299).

A LANGUAGE DESIGNED FOR COMMUNICATION BETWEEN COMPUTERS OF DIFFERENT TYPES.

by E. Nuding

Siemens-Schuckertwerke, Erlangen, Germany

If we try to establish direct communication between computers of different types in a system, in which man has to be included (Fig. 1), then the main difficulty lies in the differences in the lists of instructions. If we face the situation, where a computer has to transmit information to another computer during the calculation, then this difficulty clearly cannot be solved by a common *external* language, but has to be approached by some sort of common *internal* language.

The most natural choice would be to choose as a common language the kernel of the different primary languages involved (Fig. 2). By this we mean the set of all the instructions which are common to the computers A, B, C in Fig. 1 in the sense that they are included in the list of instructions, but need not be represented in the same manner. We shall call this correspondence "syntactical equivalence". We may expect to find in the kernel such instructions as "add", "go to" and "store".

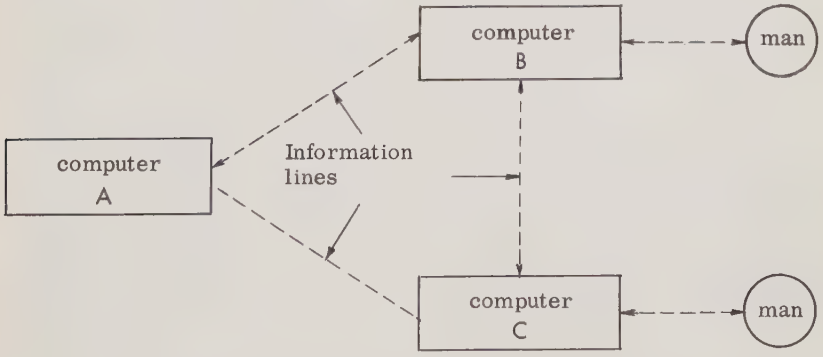


FIG. 1. Collaboration of several computers.

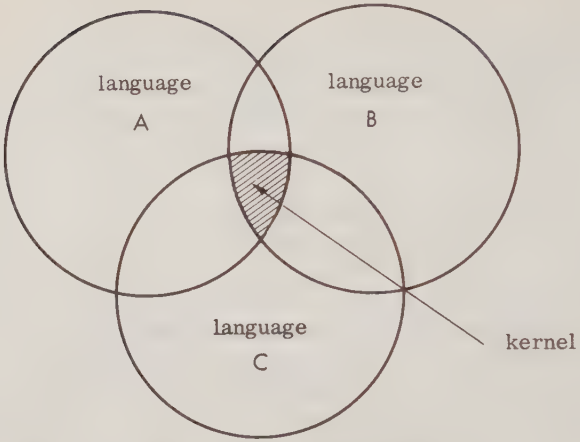


FIG. 2. Common kernel of languages A, B, C.

We may imagine that such a choice would not be a very neat one, as we would hereby use the most powerful computer in just the same manner as the least powerful one. Therefore, we have extended the language beyond the kernel by adding pseudo-instructions to the kernel language. These pseudo-instructions may be represented by machine-instructions in some of the

Instructions		Pseudo-Instructions
CLEAR ADD (z)	(z) → m, o → a	LABEL
		BEGIN
ADD (z)	(a) + (m) → a	END
	(z) → m	LOCAL
MULTIPLY (z)	(m). (z) → m	FOR
STORE (z)	(a) + (m) → z, m	pseudo-GO TO
EXECUTE		
GO TO		
etc.		

FIG. 3. Some typical instructions of the language.

computers but not in all of them. At this point we arrive at a *partially interpretive* representation of the language in the stores of the computers.

By a partially interpretive technique we understand that the computer may follow an internal-coded instruction without reference to an interpretive routine. Every occurrence of a pseudo-instruction leads to an exit out of the program, to an interpretive routine, where the pseudo-instructions are decoded and executed. Afterwards control is again transferred back to the main routine. Such a technique may be applied to a variety of computers provided they are equipped with facilities which permit control to be transferred to a fixed location on every occurrence of a "marked" instruction. Of course, the contents of the instruction counter, the instruction register and the accumulator must not be destroyed during this process.

In this technique are combined the benefits of using an interpretive system (flexibility, any variety of instructions) with the efficiency of direct coding. Indeed, a prototype system showed an efficiency of only few percent below optimal coding.

Fig. 3 shows some typical instructions of the language. In the case of imperative statements we have extended the principles of the LEHMANN-language, employed on the D1-computer.[1]

Here, a second arithmetic register (M-register) is required, in addition to the accumulator, for storing partial products. This device enables us to write down in one statement such algebraic formulas, in which all opening brackets are clustered on the left. Therefore, storing of intermediate results is automatic in most practical applications. Unary operations are performed with the contents of the M-register as operands.

Of course, this special treatment of algebraic expressions is not characteristic of the language. It is equally possible to use a more restricted system with only one register (accumulator) in the usual sense. Equally possible is a more elaborated system with a push down list of the accumulator, suitable for handling *any* algebraic expression in one statement. It might well be interesting as an intermediate language for an ALGOL-processor.

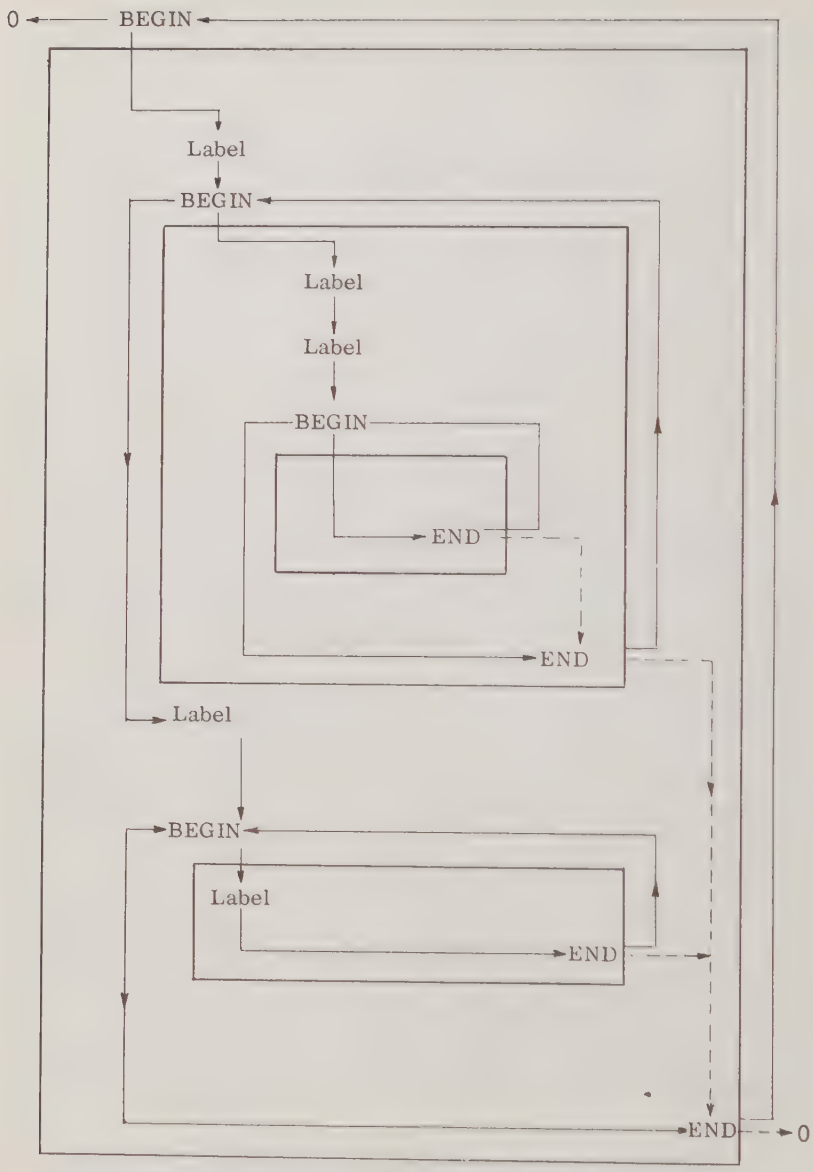


FIG. 4. Threaded list structure of "structure elements". The dashed lines point to "END" of the next higher level.

Of rather more theoretical interest should be the inclusion of some structure-elements, such as label, begin, end, local, in the language. These structure-elements are stored as (pseudo-) no-operations at the correct position within the program. They form a threaded-list-structure, establishing the logical structure of the program (Fig. 4). The opening brackets (begin, local) are the branching points in this tree. It may be noticed, that a supervisory program may reach any desired label by searching through the structure. Also the whole structure of the program may be enrolled totally from inward, beginning at any label. This latter possibility might well be considered in the event of an unexpected situation arising during the run of a program. In addition, a list of identifiers (variable identifiers, procedure identifiers) is attached to the pseudo-instruction "local". These identifiers are defined in the part of the program which begins with "local" and terminates with a corresponding "end". This device reflects the definition of "block" in ALGOL. A significant advantage in our system is that there is no unnecessary restriction in the number of identifiers.

The terminus "FOR" is included in the language as a pseudo-instruction, since for-statements may eventually serve as restart-points in case of machine's malfunctions.

In concluding we should add that we are of the opinion that the scope of the general principles mentioned above may well be extended to quite more elaborated systems than formula processors. The philosophy of viewing interpretive systems as being far inferior in efficiency to compilers, has been overcome by quite a simple technique. We therefore conclude, that we may well pay more attention to interpretive techniques in the future than has hitherto been the case.

REFERENCES

1. Lehmann, "Bericht über den Entwurf eines kleinen Rechenautomaten an der Technischen Hochschule Dresden (Bericht Math. Tagung, Berlin S. 262 (1953).

COMMUNICATION BETWEEN INDEPENDENTLY TRANSLATED BLOCKS*

Peter Wegner

L. S. E., London, England

I. INTRODUCTION

It is becoming increasingly clear that the problem of communication between independently translated blocks is a key problem in the design of an overall programming system. The facilities for communication between blocks largely determine the characteristics of the intermediate language into which source language blocks are translated for subsequent loading at execution time. Furthermore, the characterisation of the intermediate language forms a logical starting point in designing a complete programming system, so that the problem of communication between blocks should be one of the first to be considered by the system designer.

The present paper presents a solution to the problem of communication between independently translated blocks that is largely derived from the following two existing programming systems:

1. The FORTRAN-FAP system for the IBM 7090 (1)
2. The GENIE system for the Rice Institute Computer (2).

The facilities for communication between blocks provided by the Burroughs B5000 (3) are also relevant to this discussion.

Although the techniques here described are, in a sense, well known, it is hoped that the present discussion might contribute to a better understanding of the principles of communication between blocks in an intermediate language. The related problem of dynamic storage allocation for fixed and variable length blocks is also considered. Some final remarks, indicating a general approach to the subject of indirect addressing, were added as an afterthought.

*This paper has been published in substantially the same form in Communications of the ACM, July 1962.

In Section VII the block structure of FORTRAN and Algol are compared. It is suggested that Algol blocks are too heavily dependent on lexicographical context. Introduction of the declarations *'common'*, *'public'* and *'external'* is suggested, to permit greater lexicographical independence of Algol blocks.

II. COMPATIBILITY BETWEEN SOURCE LANGUAGES

An intermediate language may be regarded as a Universal Computer-Oriented Language (UNCOL) for a specific programming system. Thus a programming system which accepts as input source language blocks in symbolic machine code, FORTRAN, Algol, or COBOL (see figure 1), should translate source language blocks to a common intermediate language, so that blocks written in different source languages are compatible in the sense that they can be used as component blocks of a single object program. If the intermediate language is designed to permit compatibility between different source languages, the problem of communication between blocks written in different source languages reduces in principle to that of communication between blocks written in the same source language.

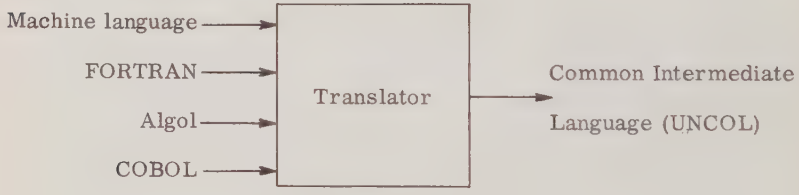


Fig. 1.

III. BLOCKS

For the purposes of this discussion, a program is assumed to consist of a number of blocks, each of which will occupy a compact region in memory during program execution. More specifically, a block may be defined as a unit of program or data which is translated independently of other blocks. It will be seen later that the problem of independent translation of intercommunicating compact (relocatable) segments is closely related to that of independent storage allocation at load time, and even to independent storage allocation during execution.

Blocks may contain data elements and instructions. Reference to data elements involve data transmission typified by a mnemonic 'FETCH NAME' where NAME is the name of the data element.* References to instructions involve transfer of control typified by the mnemonic 'GO TO NAME' where NAME is the name of the instruction or instruction sequence to which control is to be transferred.

In connection with the subsequent discussion of indirect addressing, it should be noted that indirect transfers of control do not require explicit indirect addressing, since a chain of transfers of control will do just as well. On the other hand, indirect data references require explicit indirect addressing.

When considering storage allocation, blocks may be divided into the following three categories:

1. Blocks whose storage requirements are completely determined at translation time (fixed blocks).
2. Blocks whose storage requirements cannot be determined at translation time, but can be completely determined at load time (variable length blocks).
3. Blocks whose storage requirements vary during execution (dynamically varying blocks).

Initially only fixed blocks will be considered. Subsequently, variable length blocks and dynamically varying blocks will be seen to fit easily into the storage allocation scheme developed for fixed length blocks.

IV. SOME EXAMPLES INVOLVING COMMUNICATION

1. Single Block

Consider first the internal organisation of a program consisting of a single block. Assume that the translated program is relocatable. Internal cross-references within the program can be evaluated during the translation process with the aid of a specially constructed symbol table. At load time, only a single quantity—the relocation constant—is required in order to establish the program in executable form within the computer memory.

*A one-adress instruction format is assumed for the sake of concreteness. The discussion would not be materially different for machines with other instruction formats.

2. Single Data Block

Next consider a program consisting of a single data block and a number of independent program blocks which operate on the data block but do not interact with each other. Further assume that the relative locations of all data names with respect to the beginning of the data block are known to each of the program blocks at translation time.* In this case all data block references within a program block can be specified as relative to the data block origin. The loader then requires merely a knowledge of two relocation constants when loading any given program block (i.e. the relocation constant for that program block, and the relocation constant for the data block). When loading the data block, only a knowledge of the data block relocation constant is required.

3. Single Symbolic Cross-Reference

Now consider a symbolic cross-reference involving a transfer of control from a location LOC in block A to the location NAME in block B. Assume that the location NAME is undefined when block A is translated, so that the reference in block A to the location NAME in block B remains symbolic. The symbol NAME is said to be an external symbol.

The symbol NAME is defined within B relative to the origin of block B as B is translated. Since the value of the symbol NAME is required by block A at load time, both the symbol and its value must be available in the intermediate language. This is most conveniently done by means of a private symbol table for external symbol definitions associated with each block.

On loading, the symbol table containing the value of the symbol NAME will be available in memory. When the reference to NAME occurring in block A is encountered, the corresponding value can be obtained from the symbol table.

4. Auxiliary Information—Program Parameters

It should be noted that a single symbolic cross-reference between blocks can give rise to the transmission of an indefinite amount of auxiliary information. Auxiliary information is normally specified in three principal ways:

1. In conventionally used communication registers of the computer.

*This is essentially the FORTRAN COMMON facility.

2. In a set of locations fixed relative to the point of call (program parameters).
3. In a set of locations fixed relative to the point of destination.

For instance, if the transfer of control is a subroutine call requiring a subsequent return to the point of call, the return link is normally transmitted in a conventionally specified register. If the transfer of control requires several parameters to be transmitted to the point of transfer, this is frequently accomplished by setting aside a number of locations immediately following the point of call for this purpose. The information in these locations can be retrieved using the return link parameter. The third alternative is rarely used, since it is more wasteful of space than the second alternative, and has no compensating advantages.

However, in spite of the fact that a single symbolic cross-reference can give rise to the transmission of a large volume of auxiliary information, it may still be necessary to set up a large number of symbolic cross-references which cannot be defined during translation. The general problem of symbolic communication between independently translated blocks will now be considered.

V. SYMBOLIC COMMUNICATION BETWEEN BLOCKS

In general, a source language block may contain symbols belonging to the following three categories:

1. Internal symbols defined and used only within the block.
2. External symbol definitions, defined within the given block but used within some other block.
3. External symbols, used within the block, but defined within some other block.

An example of a source language permitting definition and use of external symbols is given in section VI.

Internal symbols may be dealt with by standard assembly techniques, i.e. construction of a symbol table of symbols and their values, and substitution of values for the symbols during the translation process.

External symbols *defined* within a given block will have their values relative to the beginning of the block defined during the translation of that block. A private symbol table of external symbol definitions is compiled for each block during translation and is output with the translated program. On loading, the loader collects the private symbol tables associated with each block

into a single symbol table, so that the values of all external symbols are available for substitution in all blocks.

External symbols *used* within a given block remain undefined during the translation process, and are defined only during the loading process. On translation, external symbols are stored in their symbolic form, so that they can be identified with the corresponding definitions during the loading process. During the loading process, the values of external symbols are substituted for all uses either directly or by means of indirect address references. The use of indirect address references for this purpose is discussed in section VIII.

VI. EXTERNAL SYMBOLS IN FAP

External symbol definitions and uses must be explicitly indicated in the source language to distinguish them from internal symbols. In the FAP language for the 7090, external symbol definitions are indicated by means of the ENTRY control operation. Thus a block for computing the sine or cosine of an argument would be preceded by the two control operations:

```
ENTRY SIN
ENTRY COS
```

When SIN and COS are defined in the body of the block, they are treated as external symbols and their relative values are entered in a special external symbol table which is preserved in the intermediate language. Although ENTRY is a mnemonic indicating transfer of control to an instruction sequence defined by the symbol, the control operation can also be used to define data elements as external symbols. Thus,

```
ENTRY A
```

where A is a symbol specifying the address of a data element, defines the symbol A as an external symbol.

Transfers of control to external symbols are indicated in FAP by the control operation CALL. Thus,

```
CALL SIN
```

would designate the symbol SIN as an external symbol defined in some other block, and would cause a subroutine jump to an external symbol to be inserted in the program.

External symbols can also be indicated in FAP by preceding them with a $ sign.

Thus,

```
ADD * $A
```

would designate the symbol A as an external symbol. The * denotes indirect addressing and is required for internal organisation of the program.

Subroutine jump to the external symbol SIN can be accomplished with the use of the subroutine jump instruction TSX by means of the instruction:

```
TSX $SIN, 4
```

where the 4 designates the index register used to transmit the link.

This latter instruction is exactly equivalent to the control operation

```
CALL SIN
```

Thus, in FAP, external symbols are identified for purposes of definition by the control operation ENTRY, and are identified during use by the control operation CALL or by the $ sign.

VII. BLOCKS IN FORTRAN AND ALGOL

In FORTRAN the subroutine* is the unit which is independently translated into intermediate language. The subroutine is therefore the basic block of FORTRAN. A FORTRAN subroutine may have only a single symbolic name, corresponding to a single entry point in FAP. No symbolic reference to data in other subroutines is permitted. However, an unlimited amount of auxiliary information may be transmitted by means of program parameters as described in example IV.4. Reference to a single COMMON data area with characteristics described in example IV.2 is permitted.

It is evident that symbolic cross-referencing in FORTRAN is very restrictive. Some improvements are being made, such as the inclusion of an arbitrary number of COMMON data blocks (1). However, a general symbolic cross-referencing facility of the kind advocated here is not currently being planned.

In Algol (5), there is no counterpart to the block as defined in section 3. The Algol block may be nested within other blocks and

*For the purpose of this discussion the term subroutine is taken to include the Main program.

bears no resemblance to the block defined here. Symbolic names in Algol are called 'identifiers' and are defined by being 'declared'. The range over which a given identifier declaration is valid is referred to in Algol as its 'scope', and precise rules regarding the scope of an identifier declaration are given. Algol permits declarations to be valid over non-compact ranges by local redefinition of the identifier. However, no facility for designation of a locally declared identifier as external to a given block is provided.

The lack of flexibility in defining external program and data symbols is a fundamental flaw of Algol. The rules which govern the scope of Algol symbols seem to have been chosen in a somewhat arbitrary manner, as is evidenced by the anomalous and *ad hoc own* facility. Furthermore, little thought has been devoted to the structure of a Universal Computer-Oriented Language into which Algol might be translated (6). The specification of such a language would be eminently worthwhile and would probably give rise to modifications of Algol permitting richer and more flexible problem statement.

Algol block structure is based on the idea that the context of a block is governed by lexicographically enclosing blocks. Every block which is not a complete program must be physically embedded in an outer block which defines its context. Although it is possible to *define* (declare) a procedure which is independent of its context by channelling all communication through its formal parameters, the possibility of defining context-independent procedures with a view to independent translation is probably accidental rather than due to deliberate intention. However, even when procedures are defined in a context-independent manner, there is still the requirement that all *uses* of a procedure must occur in a single outer lexicographically enclosing block in which it is declared.

FORTRAN subroutines are deliberately designed to be independent of lexicographical context, so that they can be independently translated. This has been accomplished by channelling all external references either through program parameters or through a COMMON storage area of the kind described in IV.2.

In order to make Algol block structure more flexible, I should like to suggest the introduction of a *common* declaration for data along the lines of FORTRAN II or FORTRAN IV. Furthermore a declaration should be introduced to indicate that an identifier *defined* within a block is used outside the block, and a corre-

sponding declaration should be introduced to indicate that an identifier used within a given block is defined outside that block. The declaration for external identifier definition and external identifier use could be '*public*' and '*external*' respectively.

Introduction of the declarations '*common*', '*public*' and '*external*' would make Algol block structure far more flexible than is the case at present, and would permit independent translation of Algol blocks. Since these declarations would have repercussions on other features of the Algol language, the semantics of such a change would have to be worked out very carefully. While this is being done, the whole question of scope could well be re-examined, and the possibility of introducing sophisticated facilities for grouping of blocks and for symbol renaming, as in the MIT LSS loader (4), could be considered.

Although the above suggestions for generalising the scope facilities of Algol block structure have arisen from a machine-oriented approach, it should be emphasized that the resulting facilities are machine-independent. The suggestions facilitate more flexible subdivision of a complex problem into a number of interrelated but independently manipulable component subproblems. The 'scope' rules of a language specify the manner in which information can be transmitted between interrelated subproblems and are quite independent of any specific machine.

VIII. INDIRECT ADDRESS LINKAGE AND DYNAMIC STORAGE ALLOCATION

The scheme for symbolic communication between blocks outlined in section V requires the intermediate language block to contain a private symbol definition table of external symbols *defined* within the block, and a list of external symbols *used* within the block together with an indication of where they are used (symbol use table). At load time, the private symbol tables of each block are merged into a common symbol table, and values of symbols are substituted for all uses of external symbols.

The number of physical substitutions of external symbol values may be substantially reduced by the use of indirect addressing. Consider a block with numerous uses of the same external symbol. It would be unduly laborious to note all occurrences of the external symbol and to perform substitutions for all occurrences of that external symbol when it is subsequently defined during the loading process. Direct substitution may be avoided by set-

ting aside an additional location within the block for that symbol, and by replacing all references to the symbol by indirect references to the single location. Substitution of the value of the symbol in the single location at load time causes all the indirect references to that location to have an effect exactly the same as that arising from corresponding direct substitution for the original symbols.*

Indirect addressing has been taken one stage further by Iliffe (2). Thus, direct substitution of the value of each external symbol in the indirectly addressed location of blocks which use that symbol, can be replaced by a second indirect address referencing the actual value of the symbol in the symbol table.

This latter solution has the advantage that all references to a given external symbol are now directly linked to the value of that symbol in the symbol table. If it is desired to change the value of that symbol, it is necessary to change only the single stored quantity representing the value of that symbol. In particular, if the symbol table is in memory at execution time it is possible to modify external symbols during execution of the problem merely by changing the corresponding value in the symbol table.

The ability to modify externally defined symbols during execution is particularly useful in a time-shared mode of operation in which it is desired to assign space to blocks independently and to reallocate space to blocks during execution of the program.

Flexible reallocation of space to dynamically varying blocks is also straightforward, since both the location and the length of a block may be regarded as externally defined symbols. Thus, if the size of an array varies during execution, it is necessary merely to adjust the value of the external symbol corresponding to the size of the array.

Thus, the simple expedient of a symbol table whose values are linked by indirect address chains to all instructions using the symbol provides a flexible solution both to the problem of communication between independently translated subroutines, and to the problem of dynamic storage allocation.

In computers like the Burroughs B5000(3) all referencing of data and transfer locations is indirect, so that the indirect

*This form of referencing has in fact been adopted for FAP and FORTRAN subroutines.

mode of referencing symbol tables is automatically built into the hardware of the computer. This leaves no alternative to the system programmer but that of designing a flexible system of communication between blocks based on indirect addressing.

IX. MULTIPLE-VALUED SYMBOLS AND MULTIPLE NAMED SYMBOLS

Symbols with multiple values or multiple names can be dealt with by extending indirect addressing still a stage further into the symbol table itself.

Assume that entries in the symbol table contain indirect address references to the value of the symbol in a table of symbol values. When a symbol of given value has a number of names depending on its context, we can link all symbols having a given value to the appropriate location in the value table.

The case of a symbol of given name having a number of values depending on the context is a little more tricky. However, this problem can be solved by assigning a different name in the intermediate language to each context in which the symbol occurs. This could be accomplished by associating a context indicator with each symbolic name in the symbol table.

Multiple-named symbols and multiple valued symbols occur in control languages associated with linking loader type of intermediate languages (4). When the rules for defining context are given, the symbol redefinition facilities can be obtained within the framework of the kind of symbol table outlined above.

A special kind of multiple-valued symbol occurs in the case of symbols in a recursive procedure which require fresh storage assignment at successive levels of the recursion. In this case, the changes of context have a push-down list structure, and multiple values are efficiently dealt with by means of a vector of values associated with each such symbol. Reference to this vector would involve an indirect address indexed by the level of recursion. This technique could be applied for instance in the case of OWN variables associated with Algol procedures.

X. LOADING AND DYNAMIC STORAGE ALLOCATION AS A COMPLETION OF THE TRANSLATION PROCESS

Translation between languages of similar structure (e.g. from symbolic machine code to binary) is essentially a matter of table

look-up and of substitution of the value of a symbol for all its occurrences. During translation, table look-up followed by substitution is used to translate all operation codes and internal symbols. The same operation is used at load time to translate external symbols, and during execution to allocate storage to dynamically varying quantities.

Indirect addressing chains permit effective substitution for all occurrences of a given symbol to be accomplished by a single substitution. Effectively, indirect addressing chains superimpose a non-sequential structure on the basic sequential structure of computer memory. The non-sequential structure is a many-one linkage reflecting the fact that many symbol uses refer by table look-up to a single symbol definition.

The economy of control over symbolic communication permitted by indirect addressing makes dynamic substitution (translation) of symbol values a practical proposition.

Symbolic communication between blocks has in the past been complicated by the fact that symbol definition and redefinition required physical substitution of symbol values. However, by means of indirect addressing the operation of symbol definition can be made independent of substitution. This separation between the operations of definition and substitution is of general logical interest, and is well worth investigating. It should be possible to develop an intriguing theory of octopus-like, many-one structures representing generalised substitution operations. However, the development of such a theory will not be investigated here.

APPENDIX: SUGGESTED SYNTAX FOR ALGOL SUBROUTINE AND SCOPE DECLARATION

The syntax below is designed to introduce lexicographically independent, but intercommunicating, subroutines into the Algol language. A subroutine is similar in structure to a procedure. However, a "subroutine identifier" is regarded as a global identifier, whereas a procedure identifier is defined only within the block in which the procedure is declared. Subroutines differ from procedures in that they permit "scope declarations" for the purpose of defining global identifiers. In order to restrict modifications to Algol 60, the syntax below permits subroutines to be used only as subroutine statements. However, the syntax could easily be adjusted to permit subroutines to be used as function designators.

Syntax

< subroutine declaration> : : < type > *subroutine*< subroutine heading>< procedure body>
< subroutine heading> : : =< subroutine identifier>< formal parameter part>;< scope declaration>< value part>< specification part>
< subroutine identifier> : : =< identifier>
< scope declaration> : : =< scope symbol>< identifier list>;| < scope declaration>< scope declaration>|< empty>
< scope symbol> : : =*common*|*public*|*external*
< subroutine statement> : : = < subroutine identifier><actual parameter part>
< unlabelled basic statement> : : = < subroutine statement>

Semantics

Subroutine identifiers are global symbols defined by the "subroutine heading" part of the "subroutine declaration". Subroutines may be executed by means of subroutine statement, which consist of a subroutine identifier followed by an actual parameter part corresponding to the formal parameter part of the subroutine declaration. Global symbols may be declared within subroutines by means of scope symbols *common*, *public* and *external*. The symbol *common* serves to introduce global symbols for data and data blocks.* The symbol *public* indicates that identifiers in the identifier list which follows are global and defined within that subroutine. The symbol *external* indicates that identifiers in the identifier list which follows are global, but are defined in some other subroutine.**

REFERENCES

1. W. P. Heising and R. Larner. A Semi Automatic Storage Allocation System at Loading Time, Communications of the A.C.M., October 1961.
2. John Iliffe. The Genie System Manual, Preliminary Report.
3. The Descriptor, Burroughs B5000 System Manual, February 1961.
4. F. Corbato and J. McCarthy. Linking Segment Subroutine Loader, Share Secretary Distribution No. 64, C-1511, January 1960.

*The *common* declaration would in fact be a logical candidate to replace the *own* declaration.

**Strictly speaking, only a single declaration is needed for definition and use of global symbols. However, two separate declarations are helpful both to the programmer and to the compiler writer.

5. Naur et al. Report on the Algorithmic Language ALGOL 60, Communications of the A.C.M., May 1960.
6. E. W. Dijkstra has given some thought to the problem of computer oriented intermediate languages in his Algol Bulletin, Supplement No. 10, November 1961; and in a privately circulated paper entitled 'Substitution Processes', January 1962.

GENERAL PANEL DISCUSSION

Are Extensions to ALGOL 60 necessary and if so What Ones?

Moderator:	P. DREYFUS	(France)
Members :	R. KOGON	(U.S.)
	P. LUCAS	(Austria)
	M. NAGAO	(Japan)
	J. SCHWARTZ	(U.S.)
	H. R. SCHWARZ	(Switzerland)
	A. Van WIJNGAARDEN	(Netherlands)
	M. WOODGER	(U.K.)

DREYFUS (France)—I would like to make a side remark. There is another general Panel discussion tomorrow which says: "Is a unification Algol-Cobol Algol-Fortran possible." Therefore, we will restrict this discussion to extensions which do not include, of course, extensions to Cobol or to Fortran, which will be the subject of tomorrow's panel.

SCHWARZ (Switzerland)—I would like to make some principal remarks concerning extensions to Algol 60. At first I would like to recall that Algol was designed as a problem-oriented language and thus it was originally established to formulate numerical algorithms for solving specific problems. The first version of Algol, that was Algol 58, turned out to contain some obvious ambiguities and obscurities, and thus it was necessary to revise it. At the same time, some more general features have been added and Algol 60 became a very flexible language. The two years of actual experience prove that the present version is very powerful and general enough to fulfil its purposes, namely (a) for communication between men, and (b) as a language well suited for translation into machine code. Although there exist some situations in Algol which are not quite well defined in the reports, but which can be really settled by using common sense, I do not feel that for the moment any extensions are needed. Furthermore, if you really wish to use Algol and not to play with it, as a philosophical subject, and if we intend that the existing language is generally accepted, then any extension seems to me a non-advisable thing. It is my opinion that any extension will certainly be accompanied by changes of the language, which will cause a new version—Algol 62. As a natural extrapolation, everyone will expect new versions in intervals of 2 years. And thus Algol would never be generally accepted, and that is the reason why I think that to introduce extensions to Algol is quite a non-advisable thing.

WOODGER (U.K.)—I am rather against changes to Algol 60 from time to time. That is to say, I am concerned rather with Algol 60 as a communication medium rather than as an input language to a computer. As a communication medium, considering published algorithms for example, the reader who looks through the list of published algorithms wants generally an algorithm for a particular purpose and he wants quickly to understand what is going on, what it expresses. If for long periods of years there is only one reference language which is certainly a trouble, quite a

lot of trouble perhaps, to learn, then for this period he will completely understand what is going on in an algorithm. Changes involve a great deal more bother on the part of the reader, which is unnecessary if he knows there is one reference language. The whole principle of communication breaks down if the language is changed and one who has suffered, as I did, for changing syllabus in college will realize what I mean.

DREYFUS (France)—If all the speakers disagree with the extensions, of course, this will make the Panel discussion much shorter. But I think there will be some different points of view.

SCHWARTZ (U.S.)—I used to think I knew, but now I wonder if there is any majority opinion on the purpose of Algol. I therefore make an assumption, that its main purpose is to serve as a standard programming language, which will allow people in different installations using a variety of computers to program in roughly the same way. This makes communication of programs easier than it is without this form of language. Based on this assumption I believe that extensions to it are essential. My primary reason for this belief is the simple fact that, as it stands, it is not sufficient for a wide variety of problems being solved on computers today. Let us assume that Algol is excellent for problems of a purely computational nature. What percentage of programs today are purely computational in nature? As a minimum, the majority of computations of programs require at least some input and output capability. On the other hand, most problems which are considered to be non-computational in nature, for example simulation or utility problems or others, certainly have a high percentage of calculations within their processing. Then what is it that is lacking in Algol which does not permit its use for this unknown but presumably high number of programs which cannot use it today? First, without doubt, is its considerable lack of imagination in the data definition area. I have said this before here. Consider the seemingly trivial computation A + B: all problems, regardless of their type, require this calculation A + B. Frequently Algol permits this, as long as A and B are fairly simple for their variables. But this input is not available to many types of problems. A and B may have to be part-word, alphabetic, signed or unsigned, parts of a list, etc. Algol is not useful to a large class of problems. Alphabetic data definition capability is permitted in Algol; the structure of the language will remain essentially the same.

The calculation A + B will remain the same as it is today, look the same to the reader, but it will now become more powerful, or powerful enough to express a wide variety of programs, and thus be much more capable of being accepted universally.

VAN WIJNGAARDEN (Netherlands)—Well, I feel a little embarrassed about this question of whether or not extensions to Algol 60 are necessary, and if so which ones these are, because of the fact that next week there will be a conference of the Algol 60 authors and they want to refine the report of 1960. If I should now advocate to make any changes, this would not be a good omen for this conference. I do not think that in general extensions to a language like Algol 60 are advisable. If you intend to do something to a language like Algol, then you should generalize it, not extend: this is my opinion, and I showed a couple of days ago that with two pages of text or so you could define a language incomparably much more general than Algol, which moreover is for the programmer practically the same; if he does not want to be clever, it looks very much like Algol, if he wants to be clever, it does much more. But this goes on to another point, and you might say: we have restricted the question to whether or not Algol 60 should be extended. Now, a couple of years' experience in programming in Algol have shown—at least to me—that I am not so very much interested in other extensions to Algol. Input-output (transput, as I call it in general) has to be dealt with in some way or another. I worked for these last two years in committees, in ECMA, in SSA on this subject, and I tried to convince my co-workers there that it is really not necessary to extend Algol, but I do not know whether I have convinced them. Now, you know that Algol is quite general, because you can declare procedures, the body of which is expressed in code; since its code may be any language, it may be anything at all, and therefore Algol is, as it stands officially, an extremely general language. Now, I think something might be done with respect to "transput". Of course, it should be done in the spirit of Algol itself. That means: nothing should be done that has any vague resemblance to machines, and nothing of any ad hoc nature should be done; it should be as general as the rest of Algol itself. If you try to do that, then there are some difficulties, and I find it advisable—this might be discussed sooner or later—to really extend Algol slightly and to introduce a new type format alongside the existing types. The constants in type formats should be formats, formats being all those sequences of letters,

digits and strings which are neither identifiers nor numbers. This means that they are syntactically different from numbers and identifiers, and therefore can be distinguished and no confusion can ever occur. That means, all programs which run so far now still can run, because they do not contain these elements. Now you could introduce format statements—just as we have now assignment statements and procedure statements—which look exactly like assignment statements and procedure statements, apart from the fact that most of the constituents are replaced by formats. Well, I will not annoy you by explaining exactly either syntactically or semantically what this means, but you can introduce in that way a kind of transput expressions which tell how sequences of basic symbols transput thoughts from the program (transput is a kind of transpositioning of basic characters). You can get in that way a definition which covers input and output independent of types of particular equipment. I think that in this respect some extension of Algol 60 might perhaps be necessary and at least advisable.

NAGAO (Japan)—I think the object of Algol 60 has been freedom. However, there are also some inconveniences, especially when we want to use it in practical computation. For example, you cannot write easily double precision arithmetic in Algol. Of course, it can be expressed in code, but if we have to use a code for the expression, this is very troublesome. I have no good idea how to solve this problem, but this double precision arithmetic may be very important. I do not know if this can be properly called extension or not, but I think some kind of notes are very useful which can help the compiler to generate an efficient object program and which can make it easier to do compilation. These notes will be machine-dependent, of course. Memory allocation, for example, will be done very well by adding proper notes in a program, when the computer has a hierarchy of storages. In this sense I think that some kind of extension will be very important.

KOGON (U.S.)—I will limit my remarks to what I would like to consider the variable changes, and whether these seem to be appropriate at the present time or not, that must be considered separately. The first class I would like to consider are those changes which would make out of Algol what has been called by some people a complete language, and by that I mean it will enable the processor to be expressed in the language itself. In

this morning's panel discussion I suspected certain differences of opinion were caused by the fact that some people were talking in the context of implementing the processors in machine language or symbolic language, and others were considering it strictly in terms of implementing it by means of a higher level language. Among these changes a particular one I have in mind is the ability to have access to part of the data field or word. Also, another thing which is very necessary and desirable in order to be able to express the processor itself in a higher level language, is to compute not only values, as is presently very conveniently done in Algol, but also to be able to compute names. Other abilities present themselves and they are evident to all those who have attempted to express processes in a higher level language, namely the ability to table operations and string manipulation. I consider all of those in one class.

The second type one could call: unifying changes; those might be changes or restrictions. For instance, let me cite just one example: the concepts of array and switch are very similar, and by introducing initialization within a declaration, namely a declaration which initializes a value with certain variables, one could consider a switch declaration to be an initialized declaration of an array of labels. This is what I would call a unifying change. These two types of changes, I think, would be very desirable in order to make the language more uniform.

LUCAS (Austria)—We tried to describe in Algol processes out of the field of switching algebra. It turned out that Algol is a very uncomfortable language for this task. First, one cannot handle directly strings or parts of strings of logical values as entities; and secondly there is no numerical interpretation of Boolean values, which is useful for switching algebra. Now we decided to have our own private language for this process, but, if Algol is intended to cover logical problems too, then it is necessary to extend Algol in the direction mentioned before.

DREYFUS (France)—By looking at the answers of the Panel members I will note that, of the 7 panelists, five favour Algol extension, and 2 oppose it. It does seem that the majority of the experts here present think that Algol as such is not satisfactory. I am making this statement to explode instead of moderate. I did this purposely. I think it is now time to give a chance to the floor, unless Mr. Woodger wants to make a short remark on my explosive remark.

WOODGER (U.K.)—I would like to contradict on two counts, please. I favour the extension, but not extensions of Algol in publications which report to be in Algol, no progressively changing versions. Also, I am in favour of extensions for practical use, certainly.

STRACHEY (U.K.)—I do not propose to take this panel discussion too seriously, because panelists do not seem to have either. I am very glad that the discussion this afternoon for the first time during this conference begun to center on the content of Algol instead of other rather, to my mind, irrelevant considerations. I would like—before considering whether any extensions to Algol are necessary—just to make clear what, in my view, are the most important ideas which Algol has introduced into this field. I think they are very important, and there are several of them. I will then say what I think are the deficiencies of Algol, where one or two things were left out, one or two things were definite mistakes and suggest that perhaps we might correct these, not in an extension to Algol 60, but by doing something else.

In the first place, the most important idea, I think, introduced by Algol is on the block structure. This is, I think, absolutely fundamental; it will remain central to all languages of this sort, which are going to be a generality. The treatment of procedures is the next central point, I think, which is very important. In a way, this is going back to mathematics instead of writing codes, it is getting closer and closer to writing functions, and I should like to see this process go further still. The last point which is perhaps somewhat less important has very considerable practical effect, and is the introduction of conditional expressions, opposed to conditional statements, conditional statements being in use for a very long time. I think Algol is one of the first languages which allowed the use of conditional expressions on the right-hand side. Things which I think are lacking in Algol, and therefore which presumably would be up for consideration in an extension or new languages, are that there is no possibility in Algol for the introduction of new types. The number of types is very limited and there is no possibility of extending the syntactical forms—which of course is a dangerous operation—or the types of variables allowed. I mean, the example of precision arithmetic is one which clearly shows the advisability of having new types. It is not enough to be able to write procedures in machine code: you must be able to write a program in Algol or in high level language, and then be able to run it at varying pre-

cisions, if you find this is a convenient way of doing things. The fact that Algol itself is a language which is very poor in some of these respects means that it makes it very difficult to express some algorithms in it. To write a list processing in Algol is virtually impossible, unless you do a trick and effectively write everything out in machine language. I do not think the treatment of name and value is very satisfactory in Algol. I think the fact that you cannot calculate names, and the whole business with names, is rather unsatisfactory. I think that to introduce the whole mechanism of substitution—which is the sort of thing that you do when you have an expression called by name in a procedure—simply when you wish to have a name on the left-hand side of an assignment statement, is to confuse two things which ought to be separated. I think the treatment of arrays also is not entirely satisfactory. To come to more trivial things that ought to be cleared up, 'for' statements are a mess. It is most undesirable that the 'for' statement, the components of the 'for' statement, should be evaluated by name when you meet them the first time. I think this can only have been a mistake. The confusion and the mixture between 'for' statements, step 'for' statements and while 'for' statements is unfortunate. You do not normally need a controlled variable when you are dealing with a while statement. And there are various other features like this.

WOODGER (U.K.)—I would like to defend John McCarthy, who is not here, just to point out that conditional expressions were introduced by him in his system.

CAYREL (France)—My opinion has already been expressed by distinguished panelists, so I make it short. I think that Algol has to be extended, because the absence of input and output expressions does not seem fully justified. Obviously, any compiler for Algol will provide something to fill up this lack, and I do not see any reason why such simple expressions are read (A, B, C) and format declarations could not be adopted within the frame of Algol. Input and output expressions are part of the program and should be incorporated into a homogeneous programming language. Secondly, I think that the idea that no information manipulation is needed in numerical problems is a wrong one. I almost never wrote a program without at least a small amount of information manipulation. I do not include in numerical problems such tasks as how to teach a machine how

to play, so I will take as example the impossibility in Algol to manipulate the exponent parts and fractional parts of a floating point number. Because I am not a pessimistic type, I shall end with a positive opinion on Algol: the possibility to write and to compile recursive procedures is a very attractive feature. I am not sure that the machine available right now permits to use the full effectiveness of this process, but it is a stimulating challenge for constructors to improve their machine language in that respect. Indeed, recursive procedure allows to satisfy this golden rule in programming, to run each section of a program as few times as possible to get the desired results, but as many times as possible to avoid writing new insertions.

SWINNERTON-DYER (U.K.)—What I have to say is a criticism of the description of the language, rather than its content. I am primarily concerned with Algol 60. What I have to say will hold true of any sufficiently complicated language. When I write a program, I know what I want to do and need rules for expressing myself correctly in the grammatical form. This requires the sort of description which is given in the Algol 60 report. It enables you to put pieces together into larger and larger syntactic units until finally you have one single program. Consequently, one ought to call this a synthetic syntax. To understand someone else's programs—and many of us are liable to want to do this—needs an entirely different account of the syntax. We now start with one large syntactic unit and have to break it up into smaller and smaller pieces which fit together in the correct way. The rules for such a decomposition constitute an analytic description of the syntax. Every compiler must start with the syntax analyzer, and it is better to have a method for this once for all, and proved to be consistent with the synthetic syntax, than for everyone to do it by himself. But there is a much more serious point. It is not desirable that a program should be capable of bearing two different meanings. We are concerned with the syntax of a language, simply because syntactic units are those which naturally have semantic content. For example, it would not be logically disastrous to be able to regard A + B + C as (A + B) + C and as A + (B + C), but we must know exactly what syntactic ambiguities are possible, so that we may verify that they cannot give rise to semantic ambiguity. So far as I can see, the only way to do this involves giving a syntactic analyzer as an integral part of the description of a language. Therefore, a language that

pretends to be as rigorously defined as Algol does require as part of it a syntactic analyzer.

So much for the methods in syntactic description of the language. There is also, however, the semantics. I should like to start with a plea: those parts of the Algol 60 report which are devoted to meaning are not always clear and not altogether adequate. There are, of course, many popular principles of what Algol means. But one gets the impression that they are mostly reprints of one of our lecturers, and the lecturer, wisely, ran out of time before coming to the difficult bits. If, in fact there is agreement on what the language means in all its details, it is desirable that we should have a comprehensive and comprehensible account of it. We cannot yet ask for a formal semantic account of Algol 60 because we do not possess a semantic language, and no one, except possibly Brooker, knows how to write one. However, we do know something about what such a language must do. Remembering that the matter syntactic description of the language must link up with the matter semantic description when we have it, we can lay down now certain rules for the syntax. I should like to state two such rules, neither of which is found in the Algol 60 report: firstly, in each syntactic unit, except those that are too small, there should be semantic content. In other words, we ought to write the syntax purely as an aid to the meaning. The second rule: if we have an example of the matter syntactic formula, and if one of the pieces in it is replaced by something else, which is of the same syntactic type and the same semantic content, this should not alter the semantic content of the whole expression. In other words, if we know the meanings of the pieces and the way in which they are put together, we ought to know the meaning of the whole thing.

In conclusion, I should like to say this: the creation of Algol was a major step forward in our understanding symbolic languages. We now know enough to be able to criticize its deficiencies both of form and of substance. We largely know this, because, by possessing Algol we have learned a lot more about languages. At this moment, we are probably so much more advanced in criticism than in construction, that it is not possible for us to make the language which will now satisfy us. If so, it is worth patching up the holes and using Algol for some considerable time more. But I feel that the right thing to do is to give Algol an honourable place in a museum and start all over again from the beginning.

DREYFUS (France)—If I understand correctly the speaker's intervention on the Panel discussion "are extensions to Algol 60 necessary", I understand that what he intends to add is a big cross along the whole thing.

WOODGER (U.K.)—I oppose very strongly the words of the last speaker—the last words of the last speaker. To scratch it out and start again seems to be the only thorough-going method. Those who have their noses to the grindstone, of course—and there are many of us—have to live with machines and live with compilers and live with all sorts of things like that, and extensions input-output seem to be their prime requirement. Input-output, of course, does involve data manipulation, treating parts of units, and I suggest that consideration of these things should not be ignored merely because they are insufficient for our future needs.

HUSKEY (U.S.)—In listening to the comments of the Panel I do not think there was anything mentioned as desirable things to do that were difficult to do in Algol, but that could be done at least by the use of a code procedure in the own code facilities. So I have two suggestions to make: those who like Algol use it; those who do not like Algol, why do not they go away and work on their extension versions, their simplified versions and stop hampering those who like it.

WEGNER (U.K)—It is rather unfortunate that I follow the previous speaker, because I am going to do some rock-throwing, and I am obviously included in the class of people that he is referring to. My approach to Algol differs somewhat from that of the other speakers, or some of the other speakers. I feel that the principal achievement of Algol to date has not been necessarily that it provides an international language for computation, but from my point of view it has been an extremely useful exercise in language definition. Now, personally, I have learned from Algol language definition the criteria and problems that occur here, and I suspect that many people who had studied out Algol rather than used it in the solution of problems have in fact had the same experiences that I had in this respect. I feel that we should use the very valuable knowledge that we have gained from Algol as an exercise in language definition for the further development of languages. There are three principal types of changes to be considered. On the whole, these go along the lines of the questionnaire that has been distributed by Peter Naur: first re-

formulation of ambiguities and obscurities; then the question about whether or not recursive procedures should be included or whether or not side effects are desirable and so on with double precision and matrix facilities; and finally embedding of Algol in a context. Now I regard reformulations rather necessary, but essentially rather trivial, and I feel that embedding of Algol in a context is the real thing that should concern us in the next run around on international languages. We are rapidly learning that the language cannot be considered in a vacuum but must be considered in a context of the programming system of which it is a part. This includes both specific context, such as data description and environment description, but it also includes more general types of context. For instance, the problem of information flow between the various logical subproblems which constitute a larger problem and also information between jobs and parallel processing. These problems of information flow will have to be considered in a language—in future international languages—and in this connection I feel that the problem of information flow within Algol is not adequately catered for by the current scope facilities in Algol. In this respect I would like to disagree with Mr. Strachey: I do not think the current scope facilities in Algol are the last word to be said on the subject, although I feel that they would probably form an integral part of scope facilities in the future. There should be additional scope facilities. I will suggest some additional scope facilities in Algol in my paper tomorrow, and these facilities will have the property that current scope facilities will not be interferred with.
Finally, I should like to reiterate a suggestion that I made in my answers to the questionnaire, namely that extensions to Algol should be made at two levels: first, the development of an agreed set of reformulations to obscurities and ambiguities which should eventually, after being circularized, be included in the second edition of the Algol report, possibly early in 1963; and then the development of a brand new international language for expressing both numerical and non-numerical procedures. This language should not necessarily suffer by being compatible with Algol. Algol 60, then, I feel, has contributed a tremendous amount in the understanding of international languages for computers. However if Algol 60 is regarded as sacred and is accorded to hold the state of an elder statesman, it must not stand in the way of further progress.

MATTHEWMAN (U.K.)—I wish to consider Algol, as Mr. Woodger does, as a means for expressing communications between man and man, rather than between man and machine. In this case, if we want to express ourselves clearly, it does not matter whether we use exactly the same notation, provided what we mean is clear. For this purpose all we need is an acceptable main framework with the recognition that any further facilities may be introduced, provided their use is unambiguous. In this way, usages may grow up and presumably facilities not at present provided are often used. The most acceptable notation will eventually come into general use. That is, if Algol is to be retained for man-to-man communication it must be allowed to develop and not be frozen for all time.

WOODGER (U.K.)—I would just like to agree with that very much. My point was that when in an algorithm which is published something is used which is not in the reference manual on this language, then the reader may be confused. I omitted to state that he may also not be confused, and this is, of course, fine.

RIESEL (Sweden)—I would like to answer Huskey's obviously very popular question, why those who do not like Algol do not go away and make their extensions to it. The Algol group has always been understood to point out very strongly that Algol is the one and only common language for programmers.

HUSKEY (U.S.)—I just have to say one more word. I think the remarks I made may sound somewhat facetious, but I think the point is that Algol 60 is the first language in this field, that has been adequately defined at all. If you look at what happened when Algol 58 was changed, and people who were in the middle of developing processes had to change what they were doing, and if they are further announced that there are changes to Algol 60, this will cause new difficulties. Some people would really like to use programs that other people develop, so extensions will cause trouble in this respect. Therefore the whole point of my remarks is: here is a reasonable standard to use for communication and let us use it. But also, as I have said many times and in many places, let us not stop research in other languages, and I am all for the studying of extensions such as have been proposed here, but let us not destroy what little standard we do have at the moment.

INGERMAN (U.S.)—I have got some points that I would like to mention. The first one is that it was suggested to me—it is

certainly not original—that there are 3 official languages at this conference: English, French and Algol. And, I would like to suggest that perhaps Algol should be extended the way English and French have been extended: concepts have been developed that needed words or equipment; words were coined, words were thrown out, new words were coined, and ultimately something became accepted in the language. Now there are clearly certain things in Algol that are missing; the problem of communication between an Algol program and the outside world is the most obvious one. There are perhaps a few others. But I think that the extensions should, rather than being legislated even in some sort of informal sense, tend rather to grow like weeds, and eventually something will remain that will be permanent.

The second question that arises is what is meant exactly by an extension to Algol 60. I suggest that there are two separate things that could be meant: The first one would be an extension such that any algorithm that had been published was still considered to be acceptable Algol, even after the extension had been made. It was pointed out to me that this is an entirely different point from arranging that any Algol program which compiled would again compile after the extension had been implemented in the compiler. I think these are two entirely different questions, and I think account should be taken of them.

The last point I would like to make is considerably more trivial, but it was mentioned that it would be nice to have both an analytic and a synthetic syntax. I would like to suggest that the syntax as it stands, for example, in the Algol report, is either, neither or both analytic and/or synthetic, and the problem is to present the syntax along with a processor in some sense and give both the syntax and the processor.

DREYFUS (France)—I hope we may meet before the year 5300. This seems to be about the time for French and English to get stabilized. Let us hope Algol stabilizes a little earlier.

SCHWARTZ (U.S.)—I happen to belong to one of the schools which did divert from the original Algol, and we often wondered if we were guilty of something. We also tried in any case at first to stick as much as possible to the original specifications, but as time went on and Algol did not go on, we diverted more and more. I often wonder who is in control. Is it a good thing to have many languages? It is obvious that it should grow. We kind

of look at Algol 58 as a plateau, where different languages at that time, better features, and some new ones were brought together. And four years later now we have Algol 60. Why not let it grow! There are of course financial problems with some people who have implemented compilers, but if they change seriously, I think the long range gain, assuming that people agree with the changes—or the majority of people—the long range gain is better.

WOODGER (U.K.)—I feel very sympathetic with Mr. Schwartz's point of view. But I cannot help asking: has someone taken the trouble to define exactly what is permitted in, say, one of his versions that he is rather proud of, in as precise a way as they can manage. I would not ask any particular syntax, any particular way of doing it; just find a way and do it.

SCHWARTZ (U.S.)—Yes, there is a rather rigorous description, which most lay programmers avoid. It happens to have a title. It is an SDC document TN555, which is open for delivery. The syntax used is different than Algol and the intermediate language is different. It has scared off quite a few people. Supposedly, it is rigorous. Some holes have been found in it, but it does exist. The thing is if you look at a page of Jovial code, except for a lot of dollar signs, which we have used for high word characters, you would not know the difference between it and Algol. The structure of the language is very similar. It is the things you are failing with, the A's and the B's, which are the difference. And these set up by themselves in what we call the variable definitions section. Incidentally, where this variable definition change comes in, can be pointed out with Mr. Nagao's request, for example, for double precision. If we define A and B to be double precision as one of their characteristics, you can manipulate A and B without changing or adding any operators or procedures to the language. You change the translator, of course. If someone happens to get an Algol program which had defined double precision A and B and he has a 64 bit machine, he may just desire to erase the double precision definition. But this is as simple a change as I am talking about.

Van WIJNGAARDEN (Netherlands)—I think it is often overlooked in some of the extensions of Algol 60 how difficult it is to define them, for instance, the introduction of such a concept as double precision, obviously wanted—I will not say needed, but wanted by a lot of people. It is by no means obvious how you can define

such concepts in the generality of Algol at all, because the concept of single precision has not been defined, so what double precision means I do not see at all.

KOGON (U.S.)—If, indeed, the only objections to changes in Algol are due to difficulties in implementation, namely that the implementation of the processes is too rigid, I think it might be possible to take the point of view that the implementor, the person who builds the processor, has a certain responsibility to make that processor more flexible and foresee such things. Namely, when he implements the translator for a particular language, certain things become obvious—certain inconsistencies. And the implementor, I think, at this time, when he discovers such inconsistencies in the definition of the language, should anticipate that this will gradually become more consistent for the entire language.

DREYFUS (France)—I should like to ask now if Mr. H. R. Schwarz has been convinced by all these arguments. He was the first opponent to the extension.

SCHWARZ (Switzerland)—I am sorry that I am not convinced by all these arguments, because I still think that Algol is very flexible and does fulfil its job for scientific purposes. That was the original purpose of the language, to do this, and not do something like generating names and all these things.

NAGAO (Japan)—I think that the Algol system is a very good system and should survive. But if we do not improve the system, it will be forgotten and I should like to propose that some extension, some improvement should be done.

STRACHEY (U.K.)—I would like to put my word in favour of the development of all mathematical symbolism, that it be evolutionary and not by legislation. I must say that I disagree very firmly with Dreyfus' hopes that the development of Algol will cease before the year 5300. I hope very much that it will not be finished by that time, because if the general mathematical language will be finished by the year 5300, mathematics also will be finished. Now, I think it is very important—I remember saying so in 1958 at a conference—that mathematical notations are really useful, because they embrace important mathematical ideas and should bring them out. The development of mathematical notations is a difficult and arduous task, and it can only be done by the general agreement of a large number of people who

use it. So I am quite convinced that this gradual development is absolutely essential, and I think that any attempt to legislate about what notations should or should not be used in public communication between human beings can only have a bad effect on the development, if it has any effect at all. I do not think it at all a good idea, as Woodger said, that the *Communications* should insist in using Algol 60 or any other particular form of language for the communication of algorithms. They may recommend it for people who want to use it, or who have no particular difficulties, but I think it would be a great pity if they refused to have algorithms in some other form, and indeed I do not believe they would.

WOODGER (U.K.)—People in the northern part of England speak very differently from those in the South and that is O.K. until they wish to communicate. The relevance of this is: there is an algorithm with world-wide circulation, hopefully not used in a notation whose changes are not communicated as far and as quickly as the algorithm. That is to say, it is fine to have local conventions, but one must have this variety locally.

MORIGUTI (Japan)—I am all for the idea of maintaining the present form of Algol 60 as long as possible and by some responsible central body. On the other hand, I, and my fellows in Japan want some practical features to the actual implementation of Algol 60, and if this could be done under the present framework, we should be very happy. I want to ask the Panel members whether the following desire could be satisfied by the present framework or not. What we want is the intercommunication between machine and man. This can be understood in two different phases. First, during the compilation time, and in this respect I propose in my own lecture the concept of constant to be extended by a new declaration, which specifies that this quantity should be computed during the compilation time. This could be done under the present framework, I believe. But to generalize it a little bit, we want to find a way of having a sort of interlude, the piece of program compiled as an interlude, which will be executed during the program input, rather than at run time. I wonder whether this could be done in a simple way under the present framework, but as I listened to Mr. Mazurkiewicz's lecture today, I feel that it could be done. Second, during the run time, we hope to have a lot of facilities. We want the computer run in every occasion, we want it to warn us, and we want

to do something to instruct the computer on those occasions. This is related to the input-output treatment in some way or another and this must be possibly done in the framework by the wide use of codes. However, I would like to ask if anything of this sort has been done successfully in actual implementation.

Van WIJNGAARDEN (Netherlands)—I wanted to make some remarks concerning the last speakers. First of all I want to make a remark with respect to what has been said by Mr. Moriguti. Such a question as the introduction of a declaration constant in Algol. Well it exists already in Algol, and therefore does not need to be introduced, because, if you want, for instance, the identifier p to stand for 3.14, you can write down in your program "*real procedure* p; p: = 3.14" That will do exactly the job. Now, you might think that this would be slow—some similar remarks have been made. That means that people can make confusion in a language, and there are different ways of implementing a language. If you say that your computer would be inefficient under this official use, well, that is your own affair, of course. The object should be: make a better compiler. But you could help your compiler, of course, by your local hardware language, mainly that your basic symbol constant stood for real procedure with additional information, with the remark to your compiler that this is such a particular case.

Now a remark of a general nature. I think we are getting old —at least, that is what we are hoping—and I feel a certain lack of enthusiasm compared with the days of a couple of years ago. There was, for instance, what one of the Algol enthusiasts once called "bad Algol manners". I do not know exactly what it stood for, but I know that under the 'good Algol manners' was always understood that some speaker jumped up from his chair, jumped over the table to the microphone, and started his speech by saying "I have very strong feelings on this subject"—I have not heard these words this afternoon, and I feel a certain lack of enthusiasm in this respect is beginning to appear.

WOODGER (U.K.)—I have a comment to Mr. Moriguti's question. He wanted input and output facilities during run time, and it is of course very easy to arrange that one uses the reserved identifiers for this purpose, such for example as "read". All that is required is that one has a number of reserved identifiers, which you are encouraged to have in the Algol report, which you use at your own installation. Before I could send a program to

the Mathematical Centre in Amsterdam, I had to learn the reserved identifiers "nlcr" which they use, to cause the printer to produce a new line and carriage return. Also the reserved identifier called "next", which I could use as a function name and would give me the next value on tape. It is a trivial point, but it may not be so widely realized as I thought it was: the way in which one can use reserved identifiers to stand for little machine coded facilities which you have in your own institution. There may be a case of some common agreement on these things; where they are absent, one just invents them and uses them.

SCHWARTZ (U.S.)—Earlier in this Panel it stood roughly 6 to 1 in favour of some kind of extensions. It seems to me this may be statistically not good, since some of us are here because we probably in the past few days have suggested changes. It now seems that there are really 4 options: one is to scratch and start over; the other is do not touch it at all; another is add to, but keep what you have; and perhaps the fourth is add to and change to a relatively minor extent, not scratch. Would it be out of order, since I gather there is going to be a meeting to discuss this subject, perhaps more officially, to have a sort of hand vote here from the audience.

Van WIJNGAARDEN (Netherlands)—I would like to say that the meeting of next week of the old Algol committee is exclusively concerned with clearing up some of the—might I say—ambiguities or unclearnesses of the old report, and nothing else.

DIJKSTRA (Netherlands)—It was definitely my intention not to enter the discussion, but somehow I could not resist the temptation. A couple of people seem to be in favour of extensions, some are not. I should like to express as my opinion that I really would not mind, for the following two reasons. First of all, I am hardly interested in any legislative steps. If I wish to communicate with other people, I regard Algol as a free citizen of the world. Nobody can force me to use Algol 60, 62 or 60+ anything. On the other hand, I shall certainly use it if so is my purpose. Use it in order to communicate and to be understood. So, no matter what happens, something may come out of it which is generally accepted or not.

The other remark is about difficulties of implementation if extensions are made. Although I regard myself, in the first instance, as an implementor, I do not think the argument valid

that the difficulties of implementation should enter the discussion at all.

LANDIN (U.K.)—Somebody has talked about a hand vote on whether we should change Algol. I think we need about 10 votes, and I want to say what they ought to be. It might be any of the following things: that I undertake not to conceive of an extension of Algol, or that I undertake not to talk about an extension of Algol, or that I undertake not to publish an extension to Algol, or that I undertake not to publish a language which is rather like Algol, but not quite, or that I undertake not to talk about some special extensions or implementing extensions, or it might be that I undertake not to organize committees to start extensions. Or it might be that I undertake not to talk to anyone who votes here.

Van WIJNGAARDEN (Netherlands)—Mr. Landin spoke so fast that I have not heard all of his alternatives. But I know there is certainly a difficulty and if it is not inappropriate at this meeting of the International Computation Centre, I shall refer to past history. More than 10 years ago, when on the constituent meeting, the members who were to vote first voted on the question that they had the right to vote.

GARWICK (Norway)—There is at least one alternative that has been mentioned here today and which I think has a lot to say for it. People talk about scratching Algol and beginning a new (sic), and another alternative we discussed is to clear up ambiguities in it. Well, why not try and do something in between: to clear Algol so it gets a nice working language, and call this Algol 62, and let this be unchanged. We have another two years' experience, so we can probably do something fairly decent, and then we start a new language.

NAUR (Denmark)—There has been expressed some curiosity as to the general feeling about extensions. I do not know if it would be correct for me to present here some facts which were collected recently by means of a questionnaire which has been circulated to quite a lot of people. There were returns from something like 46 interested people. Among many of the questions there were some concerned with changes and extensions. The questions were usually expressed in such a manner that you were supposed to give your general reaction to some proposals which had been circulated elsewhere. You had three ways of

expressing your reactions. The first reaction might be that you felt that the extension in question would be very urgent, that you would like to see it adopted quickly. The second way of indicating would be to say you would like to see it included eventually. And the third kind of reaction would be that you did not think that this particular kind of extension would be desirable in Algol. Some of these questions ran as follows: the Hockney proposal—which, I think, was described this afternoon—is it right? Is it not? Well, it has been published in the Computer Journal, and it is probably known to many of you. Those who were strongly in favour were six; those who would like to see this eventually were 22, and those who opposed were 8. The next question was about string manipulation, with some indication to the proposal by Wegstein and Youden. Those in favour were 11; those who would like it eventually were $25\frac{1}{2}$, and those who opposed were $3\frac{1}{2}$. The next was on a notation on conditional expressions and statements—this is a very minor detail. Those who liked it very much were 9, those who would like it eventually were 16, and those who opposed were 16. Another point was a slight extension of notation in some manner: those strongly in favour were 8, those who would like it eventually were 12, those who opposed were 19, and one did not understand the question. Further proposals: one was about combination with Cobol: 2 would like this very much as a very urgent thing; 27 would like it eventually; 11 opposed any combination of Algol with Cobol. The proposals by Strachey and Wilkes, which have also been published in *Communications of the ACM*: 7 were in favour, $9\frac{1}{2}$ would like these eventually, $16\frac{1}{2}$ opposed these proposals. Finally, it might be of interest: there were some questions about timing, in particular simply a question of what time would be proper for the adoption of changes and/or extensions. There were 4 possibilities and the figures ran like this: during the current year, 4; during 1963, 18; during 1964, 10; later, 8.

DREYFUS (France)—Would anybody on the Panel be willing to summarize? Mr. Schwartz.

SCHWARTZ (U.S.)—I do not know if it is a good summary. However, it seems as though the majority of people do feel there is a need to make extensions, and this is verified by some of the figures that Mr. Naur gave. On the other hand, the extensions themselves vary, but it is emphasized on the input-output and data handling in general, and for symbol manipulation to some

extent. There are other people who feel that 'who cares: I'll go on with what I have', and I am not sure that there is anybody who does care.

DREYFUS (France)—Thank you, gentlemen. I would like now to thank the Panel for this discussion and to thank the audience for their participation.

GENERAL PANEL DISCUSSION
Is a Unification ALGOL-COBOL, ALGOL-FORTRAN Possible?
The Question of One or Several Languages.

Moderator:	C. E. FRÖBERG	(Sweden)
Members:	E. HUMBY	(U.K)
	L. LUKASZEWICZ	(Poland)
	H. RIESEL	(Sweden)
	G. SEEGMÜLLER	(Germany)
	W. L. van der POEL	(Netherlands)

FRÖBERG (Sweden)—On the cover of a well-known computer journal I saw sometime ago there was a picture of the tower of Babylon. All stones had names taken from more or less familiar programming languages. The tower was—as far as I remember—somewhat unfinished, but the tower on the yellow cover of our programs seems to be finished, this picture probably being meant as symbolic. I take this rather as a view of the future than as a description of the present situation.

The subject of this panel discussion is supposed to be "Is a unification of Algol-Cobol, Algol-Fortran possible? The question of one or several languages." In my opinion the title indicates a limitation since the question, if unification is possible, seems to be of a rather technical character. Therefore it seems to be reasonable also to discuss if we want such a unification. That will be no extra charge.

VAN DER POEL (Netherlands)—Well, to start a discussion on this panel I just wanted to relate a few facts. We have already been studying Algol and Cobol and the problem of whether a combination of both can be applied to administrative processes in Holland, together with the "Study Center of Administrative Computation", for almost one and a half years. And for this reason we have assembled a great many programs in administrative and business problems in both languages, sometimes in Algol, sometimes in Cobol, and then we have compared them; and I just wanted to talk to you about the general conclusion we reached. In the data description division Cobol was rather well suited. Algol did cope with the problem but only when one had devised machine code body procedures which could do the input and output. It needed only a few of them to handle the situation. On the other hand the "procedure division" of Algol is much stricter and much more neatly defined than the "procedure division" of Cobol, and I have seen from the examples we compiled that the Algol "procedure division", although it is a little bit more illogical, is still preferable to that of Cobol. I think some sort of unification should be possible along the following lines: weld together the procedure division of Algol with a data description division in Cobol. Already some features in that direction have been made by Jean Sammet in the United States whose applications are well known.

SEEGMÜLLER (Germany)—I'd like to give some comments on the Algol-Fortran chapter. I believe we should regard the ques-

tion only in the sense of a set-theoretic union. I am more of the opinion to look at these or those features of the two languages which have turned out to be very useful, and are not contained in the other one. Clearly the scope of a discussion cannot be so widespread as in the Cobol-Algol case because both the languages considered are essentially algebraic, although Fortran goes a little bit more into the direction of Cobol because of the input and output facilities. On the other hand, Algol is much more generalized and we all know that this is due to the block structure which allows dynamic storage allocation, and also due to the more general procedure declaration. I shall now enumerate some features in Fortran which have shown—I believe—their usefulness. Please forgive me if I forget this or that. The Fortran language in its early stages was not as complicated as Algol, and therefore the danger to make errors was not so great. I do not believe that Fortran has such pitfalls as Algol, for instance, when a semicolon is forgotten after an "end". The "for" statements of Fortran are restricted in such a way that very efficient object programs may be generated. More generally I might say that Fortran does not contain some completely unnecessary generalities as we have in Algol, for instance, the possible side effects of functions, and the name replacement concept, which means only hidden subroutines. I am strongly in favour of polishing Algol because these features are only barriers in the way of a wide acceptance of Algol. I summarize my opinion on the Algol-Fortran question: No union of the languages in the literary sense, but the Algol forces should come down and should accept just a little bit more of the spirit which comes from Fortran and which means "we want to make good use of our present computers".

LUKASZIEWICZ (Poland)—I would like to express my opinion concerning the question of using one or several languages in a computing center.

Now, in Poland, we are using indeed two autocodes for numerical problems. That is, for practical every day computations, we are using a machine oriented autocode Sako. For publication of algorithms we are using Algol 60. Why do we use Sako and not Algol for practical computations? Here are the reasons. Sako takes advantage of all machine potentialities. Sako covers the whole field, not only the numerical field, but also logical applications. Thus we can put in the Sako language two numbers in

one machine storage place, we can manipulate strings and so on. There are no such possibilities in Algol. Further, the resulting object program when using Sako is very efficient both from the point of view of execution time, and that of economizing on machine storage. And last but not least, Sako is very easy to learn and very easy to use. We think it is simpler than Algol.

So the advantages we have reached with Sako we could not have reached using Algol. So we are now implementing Algol, not for practical computing but rather for testing algorithms. We wish to publish our results but first we want to be sure that there are no mistakes in them. The object program in Algol has not been very efficient. In general it seems to me that a practical full realization of a fully universal language independent of every individual machine feature will always lead to compromises at the expense of the full economical use of the machine. Whether the machine is fixed or floating point, binary, decimal or alphanumeric, using magnetic tapes or not, etc., all this must of necessity influence the programming system which claims to make efficient use of the machine. But the difference between practical languages designed for different machines should not be considered such a misfortune if similarities in their general structure and other common features exist. To obtain similar languages designed for different machines, it would be helpful to establish some generally accepted pattern. Algol of course should serve as such a standard pattern, but I personally think that Algol is not the best, not a full solution for such a purpose, and that another standard should be created. In conclusion, I think that we must agree on the existence of many different languages individual to the respective machines. But what we should try to achieve today is a similarity among these languages.

HUMBY (U.K.)—I believe that it is mainly the difference in applications which gives us the requirements for several different languages. We must remember that in the scientific field we are probably dealing with problems with a variable move according to the rules set by God, and to find our way above this beautiful pattern we shall need more and more generalization. Unfortunately, in the world of commercial activity, we are dealing with variables the rules of which are set by men, by the laws of governments, by promises made to shareholders, by contracts with special customers, by agreements with trade unions; these are very pretty things and the more complex the commercial

structure the more particular must be the language in which we describe the problems.

I think that it is a feasible project to talk of an amalgam of Algol and Cobol. I think it might be a useful exercise for - say - students of programming languages to think about the problems of creating a total language near Algol and near Cobol with both sub-sets. But I think that, in fact, we would find that for commercial problems we tend to use the facilities which lay at one end of this language, and for scientific work we would tend to use the facilities which lay at the other end of the language. There might be very marginal use for such a language in problems of production control, where there is an overlap between scientific work and the production of commercial documents. I think - as I said - that this is an exercise.

I see programming languages as a development from the assembly stage code right up to the kind of language in which the programmers might specify the record output and the possible input and leave a compiler to do the rest. In this Conference we have been very well disciplined. I hope well disciplined and not shortsighted as most of the language talks have been on the Algol-Cobol level. In my opinion it is a little bit too early: we have just reasoned about the area where we can see communality of languages, we have only just got away from machine dependence. Surely this is a little bit early to talk about communality. Mr. Van de Poel said yesterday "let us let the theories loose". I agree with this, but I hope, when we do, they sore up towards this level and not just oscillate backwards and forwards across the Algol-Cobol level.

RIESEL (Sweden)—First I want to consider what should be meant by a unification of Algol-Cobol or Algol-Fortran. Then we are forced to find out what is meant by Algol. According to the lectures given here during the past week, there seems to be a great variety of systems for automatic programming with Algol. The same common feature of all these dialects seems to be the capacity of handling arithmetic and Boolean expressions and the typical feature of the "begin" "end" parenthesis on the procedure device. If the coding system can be easily changed so as to contain these formal features, we might claim that this variant of the coding system is a dialect of Algol. In this respect, at least, a unification of Algol-Fortran seems possible to carry out. So for everyone who wants such a unification it must be a very

happy coincidence that the Algol committee, who, according to the statements of some of its members, worked Algol out without thinking of the construction of any existing computers, produced something similar in so many respects to Fortran, which is a system designed for use in existing computers. Therefore, I think that at least such a unification as I mentioned is easily worked out. If by unification we do not mean the said unification, by a complete unification, it looks quite different. Let me first look at this problem purely theoretically. To carry out a complete unification of given languages is always possible. The only thing you have to do is to mark pieces by special signs in order to avoid ambiguity in the program that is written in the different languages you want to unify. In this way, I think it is always possible to carry out a complete unification of Algol-Cobol, or Algol-Fortran.

But I doubt whether it is a practical desirable unification. I believe that Algol and Cobol are so different that a practical and not too complicated unification is impossible. Then arises the question, what are we going to do instead of such a unification. And I must say that I do not share the fears of the speakers of yesterday night's panel discussion: that extension to Algol is dangerous to make. If we make proper extensions so that everything old always works in the same way as before then there is no problem for the users of the coding system. Old programs will always work in the way they have done from the beginning and perhaps it is more desirable to extend Algol by some device for handling strings in order to manage data problems, for instance, by the system that has been constructed by Wegstein and others.

SCHWARTZ (U.S.)—It seems that there is a continuous going back to the same problem, although it has been stated. It has been said here that there are two extremes to this programming backward and forward. At least perhaps you can accept that the commercial application is one extreme and the purely scientifical is another extreme. However if you then say "we can separate and have two languages for these two extremes" you begin to say that there are many classes of problems which do not follow either extremes, for example, a bridge plan program. If you think of many other classes, you don't have two-language problems, you have many-language problems.

Perhaps this is a solution, it is all we have today, but if one really investigates the differences between Algol and Cobol and

how one can bring these two extremes together I don't believe that it is quite as bad as it may appear. As I said, it is not cheap, but as this is possible perhaps it is worthy.

BASTIAN (U.S.)—It has been my personal observation that in most areas where it would seem to be advisable that there exist a joint compiler for Fortran-Algol or Cobol-Algol, what was really needed was simply a tie between subroutines written in the two different systems. That is to say, in a program which required the use of both languages, and I do agree that there are programs which have such a need, the whole thing could be solved by simply running on two different compilers leaving certain subroutines open to be written in the other compiler language, so long as the source program can be read and controlled by the same monitor. I feel for this reason that writing one big compiler is a great waste of space in many applications.

WEGNER (U.K)—I agree with Mr. Bastian that we do see that the problem of several languages versus one language is not a real problem. In particular I'd like to re-formulate what he said and suggest that in a programming system, when we consider the subroutines as the basic units from which we translate into a common intermediate language, the difference between one language and many languages tends to disappear. We then have various alternative types of languages in the system, we translate into this common language and then we carry on and have all the separate subroutines compatible with each other in the system.

BERNSTEIN (U.S.)—I think that somehow everyone seems to be missing what I think is a very simple point. In Algol or Cobol or any of the available algebraic or algorithmic languages, the philosophy of describing files on the one hand, and assuming floating point numbers on the other, is the only distinct difference, as one manipulates all these specified objects essentially in the same way.

INGERMAN (U.S.)—I'd like to say again here something very similar to what I said yesterday in the panel discussion. This topic seems that of combining Algol with Cobol, or Algol with Fortran, and I'd like to point out that it is certainly true that those who are in the programming field have to find how to do it. It is not particularly easy to talk with our friends who are not programmers about what we do for them, but usually by being somewhat imprecise you can get across with what you think you are doing.

I think the problem is not so much of combining the language Algol, whatever that is in fact, with the language Cobol, which is also something pragmatic, but rather trying to find some sort of language in which all of these are legitimate dialects, and hopefully this overall language will still be able to express the problems which can be expressed in any of its constituent languages. However, the same thing would probably apply here that applies in natural languages, that is that one would expect that the problems would be stated certainly less elegantly.

VAN DER POEL (Netherlands)—I don't want to make any comments nor an observation. I think the main reason for starting different types of languages in the business data processing field and in some scientific fields, is that we talk to different people. A businessman is usually the man who could not learn mathematics in school, and later we meet him again. He is the boss and expects to understand the system. We have to talk to him in plain English, and I think this semipoint of Cobol is the most important point of it. For example, on the Cobol side there was more talking about having a Cobol compiler before it was really in existence than was the case in the Algol field. On the other hand I would like to point out that in machine coding there has never been a difference at all. Machine coding for scientific machines is virtually or practically the same as for business machines. There is no difference at all. So for me, I think, it is more a question of the people who handle it and have to learn each other's languages and of the laziness of these people, than it is the real need for different languages.

SCHWARTZ (U.S)—I'd like to point out two practical considerations in the separation of compilers and languages. Firstly, in many problems it is very hard to define what kind of problems it is and we certainly don't want to have a special language for every obscure area. Secondly, there is a very practical consideration in the field of compilers, and that is once you have a compiler, no matter if your language works for a specific matter or not, people want more in the language and in the compiler. In addition to that, compilers are frequently required to operate within some large system which requires a particular kind of output. Well, if you have five compilers for five different types of problems, you need fairly good specialized people, one or two perhaps on each one. Here you have an economic problem: you have five times n people to maintain your five compilers and

to improve them - this is considerably expensive. If you have one large compiler the total cost is actually less. This is a practical consideration. In addition, you find a lot of duplication of effort in the five compilers. You find that in order to produce an efficient code in one case, you do an operation, and somehow the original compiler was written differently, and the second compiler in order to produce the same code has to do something else, unless you have a very good intermediate language.

STRACHEY (U.K.)—When we are considering the problem whether it is either possible or desirable to unify Cobol, Algol and other languages, I think that it is useful to consider not only whether it is either possible or desirable but whether it is likely to happen. That is to say, I think it is worthwhile considering for a moment or two the motives of people who design and want to use a new language. And if we look into this, we will see that the motives are rather different.

Some people want to use a different language because they want it for notation, for using it to write in English or longhand, or they use it to write mathematical notations or for some other reasons like statistics, or whatever it may be in their own particular problem area. And for this reason: in order to save time and a lot of trouble for themselves and everybody else - so they think - they like to construct an entirely new language. Well, it is possible but not necessarily certain that they might be satisfied if somebody else were able to construct a dialect or a perfectly general language, but it is absolutely essential that this general language should have the facility of using new syntactic forms and new primitive procedures. This project is not one that can be laid down in advance, which can be worked out by an international committee, it is a really difficult problem. It is not at all certain that such a language exists. I think that the concept of merging the existing languages whereby you can pick little bits or write in terms of them, will create difficulties and disadvantages. Even if it is possible to produce a combination of Cobol and Algol, I do not think that anybody will ever use it. I think the motives of producing new specialized and much better languages will still remain. I just don't think that people will stop inventing new languages, and you ought to make it easier and not more difficult for them to do so because I think that the invention of languages is one of the things which has to go on before we understand processors and the use of languages.

RIESEL (Sweden)—Obviously the desire to compile effective object programs for different types of problems is a very strong desire, and I think that this desire has led to the present state of splitting up into scientific oriented languages and into business oriented languages. But it is not necessary to have different systems of coding and different compilers to obtain effective object programs.

The general solution of this problem is a compiler with a syntactical analyser that can accept the description of the different problem oriented languages and translate them all into efficient object programs.

KATZ (U.S.)—Mr. Strachey said before that if combination of these languages were to be made, that it would probably not be used, and that the languages should be put together on a higher level. I agree that there should be research and work on establishing a complete language on a higher level for teacher's use. But nevertheless we are dealing with machines which have to be used today on many types of problems. The "Gecom" system we have is a combination of Algol and Cobol and some others. It is in the field today and it is being used extensively.

There was a great deal of talk yesterday that one of the extensions that was necessary to Algol in order to make it a more useful language is some way of handling data. The data description that is available in Cobol is ideally suited - I feel - for a combination with the more powerful procedure language of Algol.

It was also stated a little while ago that a combination of these two languages into a single system would make a much too large system. This also is not really the case. Once you get passed the horizontal source language, most of the rest of the compiler - if you had built two separate ones - would be nearly identical with the two systems. Some subroutines, some generators which you have in the library will differ, but the expense of building a combined system is certainly a lot less than building two separate systems. Also, if you build two separate systems and you try to combine them at some lower level - as it was suggested before - you lose a great deal in what the basic purpose of these languages is. That is, you lose the documentation that you would have if you were able to intermix the various statements we have in Algol and Cobol, if you try to combine these at some subroutine level.

There is one warning that I should make. I said that a combination of these two systems would not be larger than the sys-

tems themselves, and in this case I was referring to Cobol. Cobol - I think you will find - requires a much larger configuration of machine than Algol because of the various divisions, because of the data division that allows you the description of types of data, and the fact that these things have to be brought together with the procedure division and so on.

INGERMAN (U.S.)—I think I'd better say over again fully, or more carefully, what I said before. I'd like to point out that English, for example, is not a scientific language nor a business oriented language, and I'd like to suggest that Algol, Cobol and the like, and specifically the problem of translation to machine languages, can be considered as models of the problem of translating natural languages from one into the other; because they are models, they have certain nice properties and are a little better behaved than natural languages, but sometimes not much. I'd like to point out that there is a fairly well defined growth process for natural languages: it takes time, and I think that yesterday it was mentioned that perhaps Algol will be completed by 3500, but there is a world of fine growth processes, a world of fine ways of introducing words; somebody makes them up, somebody else uses them and if they happen to be a need, they stay in the language.

Now, because Algol and Cobol are in fact mechanical languages, the growth process - I think - can be accelerated slightly. The translation problem is still a model of the translation problem for natural languages and the problem of combining these two languages - I think - does not exist as a problem to be worried about because if it is in order for it to happen, if there are people who are in fact interested in both fields, languages will, under their own weight, combine. And, again, it does not have to be legislated to happen; either it won't happen, or if there is a need for them, it will happen.

HARRIS (U.S.)—I am particularly interested in large information systems, on the one extreme, and in small business on the other extreme. Also I am looking far into the future: 63, 65, even 68 and 70. I am particularly interested in seeing - and I feel it is possible to have - modularity in language. It is my feeling that we can have a single language, and I am attempting to foster research in this direction, a single language within which we could embody those which are particularly concerned with business application as well as those concerning information handling -

where Jovial might well place itself - and those which are extremely scientific, as Algol claims to be. I also feel that this language does not come upon us immediately but, as I say, there will be five or ten years before we can really reasonably expect such a language.

BALKE (U.S.)—It seems to me that there are three problems connected with the topic. Money, machines and motivation. We have some problems which are so large that no inefficiency whatsoever is permissible in their execution. These things go on hour after hour, week after week, year after year, using up most of the computer time of the machine, and 1% inefficiency in a problem like this is absolutely intolerable. These problems will probably never be done in any sort of automatic coding language until we have machines of infinite capacity.

The needs for small programs in our installation has so far been nicely met by Fortran in the 704 and the 7090. We have currently no plans to use Algol. This does not mean that we have any dislike of anything like that, but so far there is no need for it; if and when we have a real need for the facilities of this language, then presumably we will either make a translator ourselves or we will use one which is produced by our commercial supplier. At that time we will not be too interested in combining the languages but we will probably want a translator from Fortran to Algol from the less general language to the more general language.

Now in connection with making more and more sophisticated languages, Ingerman points out that when you code up natural languages for the machine, the limitation is in the machine. There comes a point simply where the machine either is too small for the language you want to make or the language is too sophisticated for the machine. It would have been quite unfeasible, for example, to develop a colossal language for the 704, the machine is too small, too slow, the logical manipulation does not work well enough, it is necessary to specify the characters. It is probably marginally feasible on the 7090. We have considered that, but these things, the development of more sophisticated languages, must wait till the technology advances to the point where larger and faster computers can be made, or perhaps even computers which have a completely different logical organization.

Furthermore, it seems to me that there is a very important difference between a large class of scientific problems and business problems. Frequently in scientific problems, the central processing unit of the computer will be occupied for periods as long as one hour at a time with no input or output whatever. The thing sits there and chews away on its input and then after an hour or so it gives five lines and then starts computing again. And usually in business applications it seems to me that you have a very large input and output to be done and therefore you have questions of parallel operations within input and output units and overlapping of several problems operating simultaneously.

RIESEL (Sweden)—I want to say something about the point that was raised, that languages gradually grow together if there is a need for this.

I will also put a counter-question: is there a need for a common world language in the ordinary sense? And the answer is definitely "yes". And then the next question: does everybody speak the same language? The answer is "no". And we also know that when a language disappears or changes, then this is subject to political or other circumstances that forced it to change, and that gradual change of natural languages is a very slow process. And I think that if we find that there is a need for linking together two programming languages, we can't just sit here and wait for time to link the languages together. This will take too long. We must decide to do so if there is going to be a common language.

DIJKSTRA (Netherlands)—If people explain to me the great difference between Cobol and Algol, they always praise the point of the data description in Cobol. And, indeed, it seems a powerful thing and I can quite imagine that a language which has much of the structure of Algol but has also something of that sort in it, will indeed become a very powerful thing. On the other hand I have been investigating size of translators; it has become apparent now that making an Algol translator is a job of the magnitude of man years. When I first heard the size of a translator made in England, consisting of 40,000 instructions, I shivered, and I hear now that sizes of translators have reached 150,000 instructions, and when you ask the reason for this they tell you—I can imagine—that this is largely due to the fact that the translator has to process data description. The price you have to pay

to add this to Algol seems to me extremely high, and I have the feeling that if you are going to spend anyhow 600,000 instructions on compilation you might have Algol as a small subset in it; with the difficulties of translation, compilation, implementation, it seems to be an extremely useful subset.

WEGNER (U.K.)—I was looking at the title of the discussion which says "is a unification of Algol, Cobol and Fortran possible?" and I would like to concentrate on whether it is possible or not and in particular discuss some things concerning the implementation of programming systems for several languages.

In the implementation of programming systems many operations are common to all translation processes. I'd like to enumerate some of these. For instance, the scanning of source texts by the constituent—called recogniser—is a common constituent; the transliteration from one representation to another representation is another common constituent. Then we require data format definition facilities, data conversion facilities and, in ambitious programming systems, modifications of previously converted data and programs.

Since many operations in translation and processing, and in organizing these things into a system, are common, I feel that the system involving several languages need not necessarily be a great deal more complicated than it is at present, and in this I would like to disagree with the previous speaker. It might be the case that heavy weather has been made of building complex programming systems, but it seems to me that, looking at the requirements, a complex programming system does not necessarily involve an astronomical increase in the number of programming instructions. The increase might be necessary if we require to give the system flexibility of some kind, but I don't think that the several languages or the specific facilities lead to such an increase. It seems to me that the designing of a single programming system for a wide spectrum of languages and for a wide spectrum of modes of operations can be characterized in a general way, and I feel that a great deal of attention should be paid to this matter. This problem has been neglected at this conference and I hope that more attention will be devoted to it in the future: in other words, the problem of generally specifying the requirements of programming systems for a wide variety of languages and a wide number of modes of operations.

KATZ (U.S.)—I was one of the people that talked to Dijkstra about the number of instructions and cost of these combined compilers and of Cobol compilers themselves. I'd like to point out that Algol is certainly an extremely useful subset of a combined language such as this. The cost of building a Cobol compiler, be it combined with Algol or not, is a great deal more than building an Algol compiler, and I misinformed Mr. Dijkstra: this is not due entirely to the data division. The procedure division of Cobol is not generally defined: each instruction is different than every other instruction in the language; this creates a need for a certain specialization in the de-coding and understanding these instructions with the various variations that are allowed in each instruction. And these differ from one to the other. Also there is the case of the environment division. When you want to specify what machine you want to compile on and what machine you want run on, what type of configuration, your library of generators and subroutines has to be quite extensive, and your input and output system has to be quite large because it has to cover the case where you have a card machine, where you have a tape machine, where you are using a Ramac system, or anything else. However, I might point out that in the past few years there has been a number of advances made not necessarily in the machines themselves, although this is coming, but in the techniques for implementing compilers. About 4 or 5 years ago, after the first data processing compliler was put together, it was estimated that it would take something between 40 and 50 man years to build a Cobol compiler. Today we have got that down to about 15 man years, which is still substantially larger than many of you have spent on Algol compilers. Now you may ask "why do we go to the expense of building these things?" Our customers are doing business problems, they are doing scientific problems, they have many problems that fall in between. And as to Ingerman's comment I agree, but we cannot legislate. If we had a group of us sit here and say "we will combine these languages in such and such a way", this would be nice but I do not think that we are ready for that yet. Let SDC and General Electric and a few other people spend some money on experiments and see what they come up with, and after we have examined these various things, perhaps we should work on them in committees and select those things that are best to combine.

GARWICK (Norway)—It seems that many people like Algol 60 and

many people like Cobol, and therefore they think they would like a mixture of them. I personally, am very much in favour of eating "canard au sang", I also like "crèpes Suzette", but I wonder whether the mixture of them would be very palatable. If I have to look at the future and think of what food I would like to have, I would have some sort of a kitchen which would produce either the one or the other. I think the same thing has to be applied to programming languages. We have to have a more general language, not one composed of specific parts. A more general language would at will produce any subset which we want, and in that case I think we would have much greater prospects of being happy.

HUMBY (U.K.)—Could I start by thanking Dijkstra for the free advertisement regarding the comparatively few instructions with which we might write rapidly compilers in contrast with some of the other Cobol compilers that we have heard about this week. I'd like to assure him that the reason why we sold Rapidwrite to our customers and not Algol was not a pure oversight, it was because the majority demand from these customers was for a language in which they could more readily describe their problems and "Rapidwrite" fitted this goal rather better than Algol. And we are quite mercenary, we set about meeting this demand. And when we talk about in which direction we should move, which is the common language, we must remember that it won't be by the majority vote of delegates to this conference, it will be by the majority demands of the users of computers. It is amusing—if you don't take these things too seriously—to think that the number of people who are using computers for sordid things like pay roll, invoices, and stores accounting, probably far outweighs the number of people using computers for abstract mathematical problems, and it is these former who will set the pace and push for communality. Communality is very real to manufacturers who have this hard work of writing compilers and syntactic analysers, and the use of common intermediate languages reduces the actual work that a manufacturer has to spend on compiling. There is a great deal more work for a manufacturer in holding customers' hands and even though we may have a common base for 20 languages, this could mean that the manufacturer had to keep 20 people employed just to hold the hands of the 20 people who are using these 20 languages. So manufacturers—and these constitute not an inconsiderable element of

the effort that is going into the production of languages and compilers—manufacturers will certainly strive for something common. I hope it will be at a level above Algol—Cobol, but if this is going to take a long time, it may easily be—I think—Algol—Cobol level.

FRÖBERG (Sweden)—I think this discussion has come to an end and actually I think I ought to summarize but I find this somewhat difficult and in this situation I prefer to use the old phrase "let the discussion be an answer to the question".

At last it is a pleasure for me to thank all the participants in the panel and from the floor whether called by label or by name. Thank you.