Network Grammars

Joe Grimes

P
98
.S9
1975



NUNC COGNOSCO EX PARTE

THOMAS J. BATA LIBRARY
TRENT UNIVERSITY

Network Grammars

Edited by Joseph E. Grimes

SUMMER INSTITUTE OF LINGUISTICS PUBLICATIONS
IN
LINGUISTICS AND RELATED FIELDS
PUBLICATION NUMBER 45

EDITOR
Irvine Davis

ASSISTANT EDITORS
Alan C. Wares Iris M. Wares

CONSULTING EDITORS
Doris Bartholomew Eugene Loos
Robert E. Longacre William R. Merrifield
Kenneth L. Pike

PUBLISHER
Benjamin F. Elson

Network Grammars

Edited by Joseph E. Grimes

A Publication of the
SUMMER INSTITUTE OF LINGUISTICS
of the
University of Oklahoma
Norman

1975

P98 .S9 1975

© Summer Institute of Linguistics, Inc. 1975

ISBN 0-88312-055-0

Printed in U.S.A. by

Summer Institute of Linguistics, Inc.

19891 Beach Blvd.

Huntington Beach, California 92648

CONTENTS

Introduction – v

PART 1 PROCEDURAL GRAMMARS

Chapter 1
Procedural Grammars in Surface Analysis – 1
Joseph E. Grimes

Chapter 2
Gurenne Clause Structure – 13
Nancy Schaefer

PART 2 TRANSITION NETWORK GRAMMARS

Chapter 3
Transition Network Grammars: A Guide – 47
Joseph E. Grimes

Chapter 4
Verb Clusters of Lamnsok – 85
Karl Grebe

Chapter 5
Transition Network Grammars
as a Notation Scheme for Tagmemics – 107
William M. Leal

Appendix
Notes on LISP – 191
Joseph E. Grimes

Bibliography – 197

265739

Digitized by the Internet Archive
in 2019 with funding from
Kahle/Austin Foundation

https://archive.org/details/networkgrammars0000summ

INTRODUCTION

Writing grammars in the field could go more smoothly if we were only able to ask the right questions at the right time. To see a grammar as a whole, to understand how each part of it influences the others, and to know exactly where to tuck each piece of information that turns up, would help us cover more ground with fuller understanding. Ideally, we should be able to get a useful picture of a language even before any single component of the grammar is complete. In addition, we should be able to build a grammar that can be tied to meaning.

For putting grammars together rapidly in the field and testing them while they are being built to see what needs to be looked into next, a promising tool has come from developments in computational linguistics and artificial intelligence. The tool goes by the generic name of **network grammar**, of which there are two principal types. First, Winograd's **procedural grammar** looks at a grammar as a set of procedures for interpreting what we hear. These procedures involve recognizing words in text, trying them out as parts of constructions, comparing them with conclusions already made, looking at their context to see if a particular analysis fits it, and even testing whether the proposed analysis is compatible with what is being talked about. The result is a **parse** or grammatical breakdown of text, which in a full scale grammar includes semantic and referential information. Only the basic parsing procedures, however, are considered in this volume; semantic and referential analyses need to be treated separately. Grammars which tell how to produce speech are also possible; they are very similar to grammars which analyze (Walker 1973).

Procedural grammars are quite close in concept to the second kind of network grammars, the **augmented transition network grammars** developed by Woods (1970,1972, Woods and Kaplan 1971, Woods, Kaplan, and Nash-Webber 1972). Both look at a text, store information that is extracted from what is found there, and use the results to decide what grammatical and semantic structures lie behind the text. Both permit a grammar to be presented pictorially as well as in words or formulas, and thus make it easy to visualize grammatical relations.

The Woods grammar is adaptable to expressing and comparing a wider variety of theoretical positions in linguistics than the Winograd grammar. As a consequence, however, it requires the grammar writer to handle more details directly. The Winograd model, on the other hand, has built into it enough realistic assumptions about language that it relieves the grammar writer of having to pay attention to many details. It balances its built-in assumptions by being theoretically the more complete representation of language understanding, embodying semantic and referential components in a way that has important implications for the rest of linguistics.

Procedural grammars or augmented transition network grammars could each be extended to cover most of the things the other does better. It is not yet clear, therefore, whether either one will ultimately be of greater use to linguists in the field who are trying to write grammars of previously unanalyzed languages, or to linguists engaged in

preparing language teaching materials who want to develop, say, sequences of grammars that represent the stages of learning through which a student goes. The best way to evaluate them for these purposes is to try them both.

The papers in this volume were all written in the field by members of the Summer Institute of Linguistics. During the academic year 1972-73 I was engaged in research on discourse which took me to West Africa and Nepal. One byproduct of that research was the use of both kinds of network grammars to untangle relationships in languages in which primary field work was being done. The papers by Schaefer on a language of Ghana and by Grebe on a language of Cameroun therefore represent the proof of the pudding. My own two contributions were written in the Ivory Coast and Nigeria to serve as orientation for the field studies. Leal, on the other hand, worked by himself in Nepal with an early draft of my transition network paper for orientation, some recent innovations by Kenneth L. Pike on tagmemic theory, and came up with proposals for refining the transition network grammar model as a result.

There was no computer available to use for writing or verifying any of these grammars. If there had been an adequate one in those parts of the world, we would have used it rather than checking everything out by hand. Nevertheless, even the small amount of verification that could be done by taking the grammars step by step turned up gaps in the data and raised questions that resulted in more comprehensive and consistent grammars than we would have written had we stuck to traditional means.

The research trip that resulted in this volume was partially funded by National Science Foundation grant GS-3180A#1, which made possible the series of field workshops in which these papers were prepared. Two other papers should be mentioned as part of the same research, the ones by Carol Gratrix and Norris P. McKinney that are cited in the bibliography, which have to do more directly with discourse structure and hence are being published elsewhere, but which make use of network grammar concepts. I appreciate the help of Ursula Poltéra, Rebecca Monette, and Peggy Attenberger in preparing the manuscripts.

Joseph E. Grimes
Ithaca, January 1974

PART 1

Procedural Grammars

CHAPTER 1

PROCEDURAL GRAMMARS IN SURFACE ANALYSIS

Joseph E. Grimes

The first kind of network grammar the field linguist is likely to want to use is Winograd's procedural grammar (1971). I put it first because it imposes a lighter burden on the linguist than Woods's does, but at the cost of embodying more assumptions about the nature of language.

Some conventions I give for writing procedural grammars differ systematically, and therefore insignificantly, from Winograd's. They are adapted to give the linguist who is not familiar with computation or artificial intelligence a working vocabulary that is closer to his own jargon than it is to computerese. At the end of the chapter is a complete list of correspondences.

THE PROTOTYPE. Winograd's prototype of a procedural grammar of English is part of an artificial intelligence exercise designed to show how language understanding can be simulated. He restricts his field of extralinguistic reference to a table with blocks of various sizes, shapes, and colors on it. Within this field of reference he is able to do three things: to ask questions and get answers about the state of the field in good English, to give commands and have them carried out on objects in a way that changes the state of the field, and to add information to what is know about the field. The model of the field of reference, the semantic system describing it, and the grammar are simulated on a computer. It takes in typed English, analyzes it, does what it is told or answers what it is asked, and displays the resulting state of the field of reference on a screen. The plan behind the model is broad enough to permit it to be extended in almost any direction: add new grammatical constructions to the ones already understood, increase or change the vocabulary, define new referential situations, or amplify the information about how things in the field of reference behave. In short, the project gives a realistic simulation of linguistic behavior in a model that allows any part to be examined and modified.

FORM. Linguists can learn to use procedural grammars with a minimum of effort. The terminology is natural enough: there are constructions, constituents, word classes, and a dictionary that gives grammatical and semantic information. Conclusions which a procedure reaches about a construction it is analyzing are embodied in the form of **features.** Each feature is put into the grammatical analysis (the report of which is called the **structure tree**) at the place where it is most relevant for the semantic analysis. The grammar makes use of information about the context of words, both in terms of their relationship to other things in the structure tree and in terms of their relationship to the whole discourse in which they are found.

2 NETWORK GRAMMARS

In order to see how closely a procedural grammar represents a language, it is verified by simply taking its statements literally and seeing what the outcome is when they are applied to text. The statements are precise enough to permit this to be done either with pencil and paper or with the aid of a computer. Not only can the grammar be validated against text in this way; it is possible to pinpoint where it goes wrong and change it. This is accomplished by keeping track of how each analysis is reached, together with remembering all the significant changes to the field of reference that have occurred since the discourse began. In addition, when an obligatory condition is not satisfied, a note in the grammar at that point helps the grammar writer to track down the deficiency. As soon as the grammar is modified, it can be tested again. By using the grammar procedures to produce text, it is possible to get a different kind of verification, the inverse of parsing.

NOTATION. The notation used for procedural grammars, like the terminology, is not awkward for linguists. Each statement or rule of the grammar has two parts: first a condition to be met by the next word or words in the text, then directions that tell what constituent to look at next if the condition is met on the one hand, or if it fails to be met on the other. Complex conditions are built up by joining simpler conditions together. The way in which the parts of a statement are put together is always made explicit by parentheses.

There are three levels of shorthand vocabulary that are used to define the grammar of any language: basic, feature oriented, and specialized. The vocabulary for expressing the **basic** components of a procedural grammar is exceedingly simple–(1) the names of constructions themselves, like CLAUSE and NP, which activate the corresponding procedure in order to decide a condition, (2) the names of word classes, like NOUN and ADJ, which are looked up in the dictionary, (3) the directions NEXT, FAIL, and RETURN, and labels created by the grammar writer for specifying where else in the procedure to look next.

The second level of shorthand allows manipulation of **features** in the structure tree. In many ways it is parallel to the structure condition of a grammatical transformation, though more flexible. It includes ways of recognizing, adding, or removing features at some node of the structure tree. It has specific names for some nodes of the structure tree, a pointer called PT that can be moved to any node in the tree to pick up or add information, and another pointer called PTW that can help identify the structural context of a form. It requires that the logical relations AND, OR, and NOT be added to the basic components to permit complex conditions to be formed.

In the third area, linguists do not really have an established notation they can call their own, because it is at the borders of what has ordinarily been thought of as linguistics. This is in the area of **reference** (including inferred reference) and **semantics.** Procedural grammars use a notation that builds a bridge between grammatical form and logical inference. It is based on the LISP language, which is a widely used means of expressing

relationships that involve complex data structures. Only the basic and feature oriented parts of the shorthand are discussed here; semantics and reference require separate treatment.

THE BASIC GRAMMAR

The basic procedural grammar is a collection of definitions of grammatical constructions. In Winograd's model, following a suggestion made by Halliday, only two kinds of constructional units are defined: phrases and clauses. These are chosen because they represent the domains within which his semantic analysis works best. The list can be extended to more comprehensive units such as sentence, paragraph, and discourse. The word, which is the minimal unit in Winograd's treatment, is looked up in the dictionary rather than defined as a construction. Other grammars might take the morpheme or formative as minimal.

CONDITIONS. Each unit consists of a series of structural **positions.** Each position is identified in the grammar by a **condition** that contains the name of one of the units in the grammar. In the basic grammar, meeting a condition successfully results in a new branch being added to the structure tree to represent the unit and to give its internal structure if it is a construction. After each constituent is found (or found to be absent), the directions associated with that condition tell what structural position to look at next.

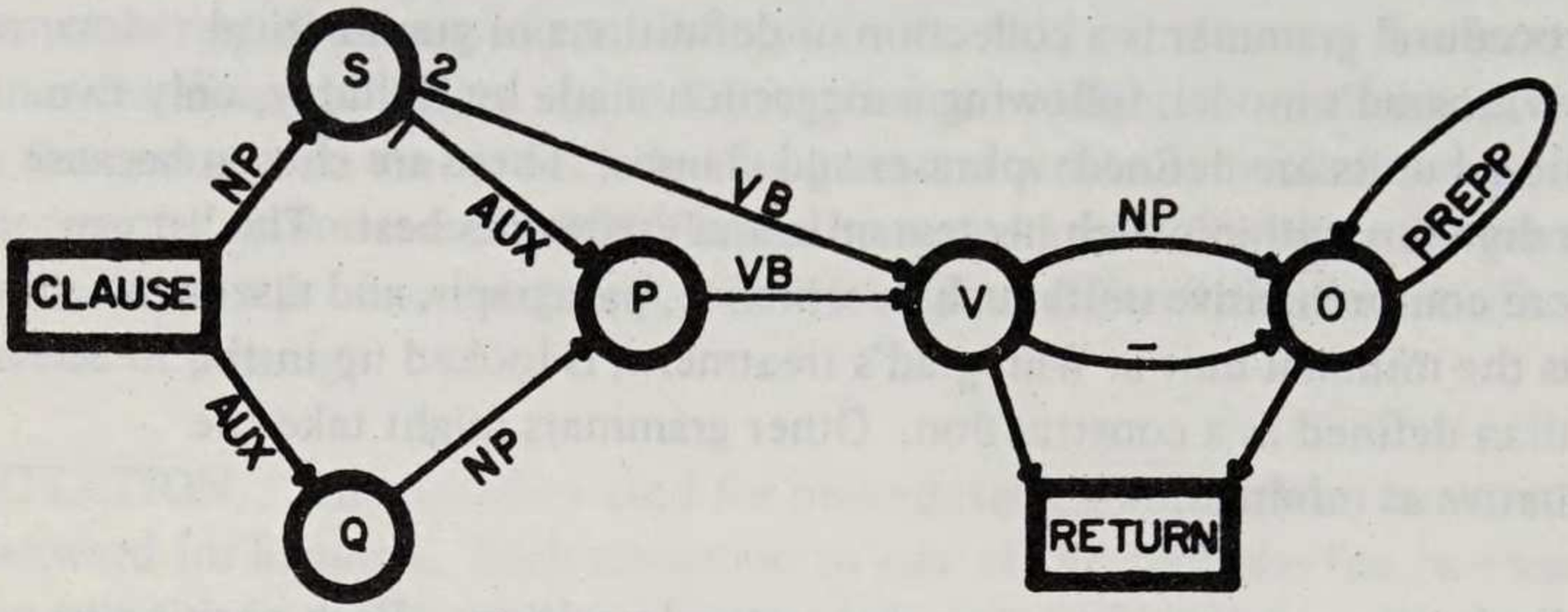
Each unit except the smallest, the word, is defined by a statement of the form (START X (. . .)), where X is the name of the unit. For example, in Figure 3 the prepositional phrase of English is defined as

```
(START PREPP (
   ((PREP) PR FAIL)
PR
   ((NP) RETURN FAIL))).
```

PREPP is the name of the unit being defined. The structural positions within it are two: the preposition slot identified by the word class name PREP and its associated directions, and the object slot identified by the construction name NP and its directions. The NP unit is itself defined in Figure 2 by START NP. PREP, on the other hand, is the name of a word class rather than of any of the complex units defined by START. Any unit that has no internal slots or positions defined by START is looked up in the dictionary, Figure 7. Each preposition in it, for example, is tagged PREP.

DIRECTIONS. The process of **parsing** (a short but little used word that refers to breaking down or analyzing a text into its grammatical components and labeling those components) is done in the order in which words come in the text, first word first, second word second, and so on. At any one point in the text there may be several possible paths which the analysis could take, depending on what has already been found. The path alternatives are expressed by the **directions** that are part of each statement.

Most statements contain two directions. Some have three. The first tells where to look next if the condition in the first part of the statement succeeds. The second tells where to look if the condition does not succeed. If there is a third direction, it represents a successful condition like the first; but it is a short cut that applies only if there are no more words left in the text. If there are still words remaining to be analyzed, the first direction is taken.




```
1.01    (START CLAUSE (
1.02        ((NP) S NEXT)
1.03        ((VB AUX) Q (TYPE?))
1.04    S
1.05        ((VB AUX) P NEXT)
1.06        ((VB) V FAIL)
1.07    Q
1.08        ((NP) P FAIL)
1.09    P
1.10        ((VB) V FAIL RETURN)
1.11    V
1.12        ((NP) O O RETURN)
1.13    O
1.14        ((PREPP) O RETURN RETURN) ))
```

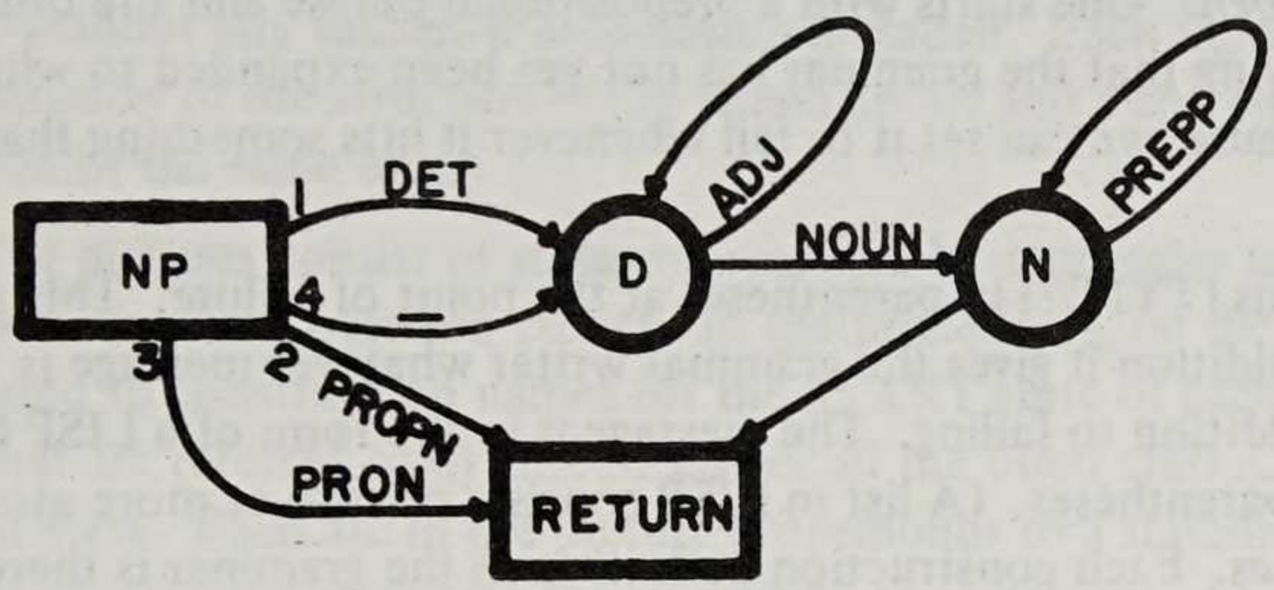
Figure 1. Basic grammar of English without features. Clause.

Directions can be given in five forms. (1) The simplest is the word NEXT, which means that the next statement in the grammar as it is written is the statement to try next. NEXT is used for the failure condition in Figure 1.02.

(2) Instead of NEXT it is possible to give an arbitrary **label** that names another part of the procedure that is to be tried next. The arbitrary label has to be what is known as a LISP **atom** to maintain compatibility; that is, it must begin with a letter of the alphabet, and must not contain any spaces or parentheses. In Figure 1, lines 1.04 and 1.07 are among the LISP atoms entered as labels just ahead of the statements in the grammar to which they refer. Every statement in this grammar makes reference to one of those labels in its directions. The label must appear within the same construction definition as the

statement that makes reference to it; there are, for example, no directions in Figure 1 that can refer to labels in Figure 2.

(3) The third kind of direction is the word RETURN. It means that the current construction has been successfully parsed. The resulting analysis of it is therefore added to the structure tree as a constituent within the unit that called for the current construction to be parsed. The words that were analyzed are removed from the text so that the analysis continues with the next word. For example, any one of the RETURN directions in Figure 2 signals the end of a unit that has the internal structure defined for the noun phrase (NP), regardless of how many or how few words there are in that noun phrase.




```
2.1   (START NP (
2.2       ((DET) D NEXT)
2.3       ((PROPN) RETURN NEXT)
2.4       ((PRON) RETURN D)
2.5   D
2.6       ((ADJ) D NEXT)
2.7       ((NOUN) N FAIL RETURN)
2.8   N
2.9       (( PREPP) N RETURN RETURN) ))
```

Figure 2. Basic grammar of English without features. Noun phrase.

(4) The other two kinds of directions are failure directions. FAIL rejects the analysis of the unit being parsed. For example, in Figure 2, even if a determiner and some adjectives are found, if no noun is encountered at 2.6, then the attempt to analyze a string of words as a noun phrase will fail. Suppose, for example, that the NP procedure has been activated in line 1.12 of Figure 1. The failure in 2.6 tells the clause procedure in 1.12 that whatever the string of words may be at that point of the text, it cannot be handled as a noun phrase. 1.12 therefore takes the second or failure direction, NEXT, to the next statement, rather than the 0 or RETURN direction it would take if the noun phrase procedure had succeeded. For failure, the noun phrase or other unit is not added to the structure tree, and whatever words the procedure looked at are treated as still unanalyzed.

(5) The other kind of failure direction pinpoints the cause of the failure explicitly. Sometimes a unit definition fails because the words in the text do not represent that unit, even though they do represent some other unit that the directions will lead to eventually. Sometimes, however, the grammar hits FAIL because it has attempted to parse the unparseable. This may come about for two reasons: either the text is poorly formed, or more likely, the grammar is not yet adequate.

For example, Figure 1 gives two ways of starting a clause: NP and VB AUX. Statements begin with noun phrases, yes-no questions with auxiliary verbs. This grammar will not handle English sentences like *After the party we went to Jerry's* and *Where did I put the red pen?* One starts with a prepositional phrase and the other with a question word. Knowing that the grammar has not yet been expanded to where it can manage this kind of clause, we can set it to fail whenever it hits something that has not been provided for.

Statement 1.03 contains (TYPE?) in parentheses at the point of failure. This is equivalent to FAIL, but in addition it gives the grammar writer whatever message is enclosed in parentheses in addition to failing. The message is in the form of a LISP **list**, which is always enclosed in parentheses. (A list in LISP consists of zero or more atoms or lists enclosed in parentheses. Each construction definition in the grammar is therefore a list, and each statement is a list.) This last form of giving failure directions, as a list, permits the grammar writer to leave notes to himself scattered throughout the grammar at failure points with a view to improving it later.

RESTRICTING A PARSE. Returning to conditions, note that 1.03 and 1.05 incorporate more than a unit name. They also name an additional feature AUX which must accompany any unit of the word class VB at that point. These two statements, therefore, succeed for verbs like *have* and *was* but fail for verbs that are not auxiliaries like *put* and *fail*. More than one restricting feature can be specified at a time. The restriction can apply both to units found in the dictionary, as here, and to units whose internal structure is specified by START, for which the unit procedure must place the required features on the structure tree node that represents the total construction in order for the procedure to succeed.




```
3.1     (START PREPP (
3.2        ((PREP) PR FAIL)
3.3     PR
3.4        ((NP) RETURN FAIL) ))
```

Figure 3. Basic grammar of English without features. Prepositional phrase.

Recursion

The procedures are activated recursively. The NP procedure in Figure 2 activates the PREPP procedure in Figure 3 to determine one of its possible constituents, and the PREPP procedure in Figure 3 in turn activates the NP procedure for one of its constituents. This recursion is a fundamental property of natural language, and as Woods points out (1970), by allowing a grammar of this type to activate procedures recursively, we eliminate Chomsky's objection in *Syntactic Structures* to finite state models of language.

PICTURES AND PROCEDURES. Figures 1 to 3 give a grammar of a restricted part of English without any feature manipulation included. Each figure consists of both a picture representation of the structure being talked about and a corresponding procedural definition of the same unit.

The pictures consist of **states** represented by rectangles and circles and **arcs** or **paths** represented by lines between circles. In the procedures, on the other hand, states correspond to construction names for the START state of each construction, and to labels that are referred to by directions for all the other states. RETURN is a special terminal state. Each arc in the picture corresponds to a statement in the procedure. Multiple arcs leaving one state are equivalent to sequences of statements. By keeping these correspondences clear it is possible to work from a graph that pictures a construction to its procedural definition, and vice versa.

In the basic grammar multiple arcs (as in 1.02 and 1.03) are tried in sequence and the first one that succeds is taken. In the full scale grammar it is sometimes possible to use semantic tests of meaningfulness in order to decide which of several possibilities to try next.

THE STRUCTURE TREE. Applying the procedures to text results in a structure tree or parse tree that reports what is found. This structure tree is a list in the technical LISP sense. It has two forms, an abstract form in terms of which pointer and feature instructions are defined, and a display form in which all the information the procedures turn up is shown to the linguist.

In the **abstract** form each node of the structure tree is a list of up to five lists: ((. . .) (. . .) (. . .) (. . .) (. . .)). Each of these lists has a shorthand name. FE, the first list, contains all the features that go with the node. CONF, the second, is a number that tells where in the text that constituent begins, starting with 1 for the first word in the text, 2 for the second, and so on. N, the third, is the number of the next word in the text after the end of that constituent. LAST, the fourth, is a list consisting of the nodes for all the constituents that depend from that node, reversed so that the latest one found is listed first. SM, the fifth, contains information attached to the node by the semantic analysis routines, which are not part of this discussion.

The visual or **display** form of a parse tree node is derived from the abstract form. First comes the string of words to which that constituent corresponds. Whenever the procedure finds a word in the dictionary, that word is treated as a terminal element of the tree and is displayed as an atom without parentheses. Whenever the procedure goes through a construction definition, the corresponding string of words (sometimes a single word) is displayed as a list in parentheses to indicate that it is nonterminal. The string is copied from the text beginning with word CONF and ending with word N-1. It may appear just above the rest of the information about the node, or it may be displayed on the right hand side of the page with the structural information displayed on the left in parallel column. The second form is used for the examples given later.

Next the feature list FE for the node is shown. The name of the unit (CLAUSE, NP, PREPP, VB, NOUN, and so forth) is always given as the first feature in this list. The daughter constituents of the unit, if there are any, are displayed under the feature list and indented. They are taken from LAST but are reversed again so that they appear in the order in which they were found.

EXAMPLE: PARSING WITH THE BASIC GRAMMAR. A simple clause like *Did George come?* is handled by the grammar of Figures 1 to 3 (adapted from an example in Woods 1970) as follows. We begin, since this grammar is only clause oriented rather than discourse oriented, with an implicit activation of the most comprehensive unit, CLAUSE. This activates the clause definition in Figure 1. The first element of a clause to look for, according to 1.02, is a noun phrase, so the clause procedure activates the NP procedure of Figure 2. Within it the first statement, 2.2, tries to establish *did* as a DET. Since there is no DET procedure, NP looks up *did* in the dictionary (Figures 7 and 8, the exact form of which we will discuss later) to see if it has the feature DET. It does not, so the NP procedure is directed by NEXT to try 2.3, the next statement. *Did* is not a proper noun either, so again NEXT tells the procedure to try the next statement. (2.4 is a label, not a statement.) 2.5 finds that *did* is not an adjective, and 2.6 finds that it is not a noun. In 2.6, however, instead of being directed on to the next statement by NEXT, the procedure NP embodies our knowledge that no string of words beginning with *did*, including *did* by itself, qualifies as a noun phrase. The procedure therefore fails, and takes us back to where we started in 1.02 with a failure condition. Failure leads to the second direction, so that the clause procedure follows the NEXT in 1.02 to statement 1.03.

The condition VB AUX in 1.03 works because *did* is found in the dictionary (with the help of a morphological analysis procedure that will be discussed later) as an auxiliary verb whose root is *do*. The direction for success is Q, so we jump down to Q in 1.07 and continue the procedure from there, just as though we had hit a go-ahead square in a parlor game. (In fact, there is a parlor game version of procedural grammar.)

Before going on, however, we should consider the effect the successful parse has on the structure tree. It hangs the structure it has found on it as a new node. We have an incomplete top node, the one corresponding to the clause we are in the middle of parsing.

There is one complete daughter node under it, the one-word constituent we have just found:

```
(CLAUSE                          (DID
   (VB AUX DO TRANS PAST)         DID
```

The features AUX DO TRANS PAST are filled in from the dictionary entry with the aid of the morphological analyzer.

Q in line 1.07 leads to activating the noun phrase procedure NP in 1.08. Like the first call to NP from 1.02, this one fails to find a determiner in the first statement, and so goes to the second statement at 2.3. *George*, however, is listed in the dictionary as a proper noun, so that statement succeeds. This adds two levels to the structure tree for the next constituent: the NP level, which may consist of any number of words even though it embraces only one in this case, and the PROPN level, in which the proper noun is a single constituent of the noun phrase which in turn is a constituent of the clause. Statement 2.3 returns to 1.08 successfully, so 1.08 is ready to go on to P (which could also have been written as the NEXT direction, since P is the next statement label). The parse tree now has the form

```
(CLAUSE                          (DID GEORGE
   (VB AUX TRANS PAST)            DID
   (NP)                          (GEORGE)
      (PROPN NS)                  GEORGE
```

At P in the clause procedure the condition VB looks for a verb without the restriction to AUX that 1.03 has. *Come* fills the bill. The information about it goes on the structure tree as the third clause constituent. Because there are no more words, however, the third direction to RETURN is taken rather than the nonfinal direction for success, which would go on to V to look for an object. RETURN finishes off the structure tree, closing all parentheses:

```
(CLAUSE)                         (DID GEORGE COME?)
(VB AUX DO TRANS PAST)            DID
(NP)                             (GEORGE)
   (PROPN NS)                     GEORGE
(VB INF ITRNS)                    COME
```

WHAT THE BASIC GRAMMAR DOES NOT SHOW. This simple grammar gives a definite account of the structure of a large number of English sentences. It could obviously be expanded. On the other hand, even with this rudimentary form a lot of information that is available and relevant to the meaning of the clause is overlooked. For example, we have enough evidence to know that the clause is independent and is a yes-no question, that the noun phrase is its subject, that it agrees with the verb in person and number, and that the form is active rather than passive. Information of that kind is kept track of by adding features to a grammar that is otherwise the same as the grammar of Figures 1 to 3.

FEATURE MANIPULATION

Two kinds of information besides unit names are handled in the form of features. The first describes which of several alternative forms has been chosen at some point of the structure; for example, if a NOUN is singular like *boy*, the number system feature NS is added to it, whereas if it is plural like *boys*, the feature NPL is added. The second kind of information tells how a unit is used as a constituent of a larger unit. For example, a noun phrase can have the feature SUBJ if it functions as subject of a clause, OBJ if object, or PREPOBJ if object of a preposition. The names of features are LISP atoms.

FINDING NODES IN THE STRUCTURE TREE. Since each feature is stored in the structure tree at the point where it contributes to the semantic analysis of that part of the tree, and since tests for feature relationships like agreement involve comparing different parts of the structure tree, a flexible means of finding any node in the tree is needed. Some nodes can be identified by their relation to the construction currently being analyzed, others by setting a **pointer** that can be moved anywhere in the structure tree, and occasionally others by a pointer that can be moved through the input text.

```
4.01    (START CLAUSE (
4.02        ((AND (NP (SUBJ)) (PUT CON "DECL")
4.03                (SETPT LAST) (TRNSF "NS NPL NFS")) S NEXT)
4.04        ((AND (VB AUX) (PUT CON "QUESTION YES-NO")) Q FAIL)
4.05    S
4.06        ((VB AUX) P P)
4.07    Q
4.08        ((AND (NP (SUBJ))
4.09                (SETPT LAST) (TRNSF "NS NPL NFS")) P FAIL)
4.10    P
4.11        ((VB (MVB)) NEXT FAIL)
4.12        ((AND (SETPT LAST PV (AUX))
4.13            (COND (IF (HAS PT "BE") THEN (SETPT LAST)
4.14                        (MAKE PT "ING") (PUT CON "PROG"))
4.15                    (IF (HAS PT "HAVE") THEN (SETPT LAST)
4.16                        (MAKE PT "EN") (PUT CON "PERF")) –
                            )) NEXT NEXT )
4.17        ((SETPT CONF NX (VB)) NEXT FAIL)
4.18        ((AND (MAKE PT "PRES PAST") (TRNSF "PRES PAST"))
              NEXT FAIL)
4.19        ((OR (AND (HAS PT "PAST") (NOT (HAS PT "BE")))
4.20            (HAS PT "MODAL")) V NEXT)
4.21        ((AND (HAS CON "NS") (MAKE PT "V3PS")) V NEXT)
4.22        ((AND (HAS CON "NFS") (MAKE PT "VFS")) V NEXT)
4.23        ((AND (HAS CON "NPL") (MAKE PT "VPL")) V FAIL)
```

```
4.24    V
4.25        ((AND (HAS LAST "TRANS") (NP (OBJ))
4.26                (PUT CON "TRANS")) O NEXT RETURN)
4.27        ((AND (HAS LAST "ITRNS")
4.28                (PUT CON "ITRNS")) O FAIL RETURN)
4.29    O
4.30        ((PREPP) O RETURN RETURN) ))
```

Figure 4. Restricted English grammar with features. Clause.

THE CURRENT CONSTRUCTION CON. The construction currently being analyzed is called CON. It is the unit within whose definition the feature test is being made. Its components are identified by the names already mentioned in connection with the structure tree. FE refers to whatever features have been assigned to CON; CONF refers to the location in the text of the first word that belongs to CON, and N refers to the next word in the next that has not yet been parsed. When naming components of the current construction, however, LAST refers only to the constituent of CON that was parsed most recently, overlooking all the earlier constituents. The others must be reached by the tree pointer instead. SM refers to whatever semantic information has been attached to CON, and will be left out of this description.

In Figure 4.27 a feature test condition looks at the node just parsed. (HAS LAST "ITRNS") asks whether the feature ITRNS for intransitive is found on the LAST node that was analyzed, using a form for specifying the test that will be explained later.

NODES REACHABLE FROM THE CURRENT CONSTRUCTION NODE. Composite node identifications are made by putting the shorthand forms in the last paragraph together in two-member lists. (CONF LAST) identifies the first word of the last constituent that was parsed within the current construction. (FE LAST) refers to the features of the last constituent. (FE (CONF LAST)), by extension, refers to the features of the first word of the last constituent.

THE TREE POINTER PT. The tree pointer PT is set by a LISP list of **instructions** that begins with SETPT. SETPT identifies the list as a tree pointer setting operation. The instructions that follow it tell how the tree pointer is to be moved up, down, or across branches of the structure tree. The final instruction may be followed by a condition, enclosed in parentheses, that causes the pointer to be moved by the last instruction as many times as necessary until the condition is met. PT by itself refers to whatever node of the structure tree the tree pointer was left on the last time it was set. The two tests in 4.19, for example, refer to the node on which the tree pointer was set by 4.17.

Most tree pointer instruction sequences begin by referring either to the current construction node CON or to one of its components. (SETPT CON), for example, sets

the pointer PT on the construction itself. (SETPT LAST) sets it on the rightmost daughter node of CON, and (SETPT N) sets it on the word that follows the last word in CON. These three instructions are parallel to node identifications that have already been discussed, except that they move the pointer as well as identifying the node.

```
5.01    (START NP (
5.02        ((AND (DET) (SETPT LAST)
5.03                (TRNSF "DET DEF INDEF NS NPL POSS")) D NEXT)
5.04        ((AND (PROPN) (SETPT LAST) (TRNSF "NS NPL"))
            RETURN NEXT)
5.05        ((AND (PRON) (SETPT LAST) (TRNSF "NS NFS NPL"))
            RETURN D)
5.06    D
5.07        ((ADJ) D NEXT)
5.08        ((NOUN) NEXT FAIL)
5.09        ((AND (SETPT LAST) (MAKE PT "NS NPL")) NEXT FAIL
5.10        (COND (IF (HAS PT "NPL") THEN (REM PT "NS") (PUT CON
            "NPL")))
5.11        (COND (IF (AND (HAS LAST "MASS") (NOT (HAS CON "DET")))
5.12                (THEN (PUT CON "MASS")))
5.13        (COND (IF (NOT (HAS LAST "NPL")) THEN (REM CON "NPL")))
5.14        (COND (IF (NOT (HAS LAST "NS")) THEN (REM CON "NS")))
5.15        (COND (IF (NOT (HAS CON "DET")) THEN
            (TRNSF "NPL MASS")))
5.16        ((MEET FE "NS NPL MASS") N FAIL RETURN)
5.17    N
5.18        ((PREPP) N RETURN RETURN) ))
```

Figure 5. Restricted English grammar with features. Noun phrase.

```
6.01    (START PREPP (
6.02        ((AND (PREP) (PUT CON (ROOT LAST))) PR FAIL)
6.03    PR
6.04        ((NP (PREPOBJ)) RETURN FAIL) ))
```

Figure 6. Restricted English grammar with features. Prepositional phrase.

An instruction of this form is used in Figure 5.05, where the pointer is set on the LAST node (a pronoun acting as head of a noun phrase) in preparation for transferring its number feature to the construction as a whole.

The second set of instructions for moving the tree pointer are relative not to the current construction but to whatever node PT is set on at the time they are given. This makes it possible to move the pointer anywhere in the structure tree by a sequence of instructions that begins with CON, LAST, or N but continues on with relative instructions, each of which moves the pointer from where the last instruction left it. This set includes six instructions. DL (down, last) moves PT to the rightmost daughter of the node PT now points to. DLC (down, last, complete) moves PT to the rightmost daughter whose analysis has been completed; thus it would not point to a node one of whose daughters was the construction currently being worked on. DF (down, first) moves PT to the leftmost daughter of the node PT now points to; that is, to its first constituent. PV (previous) moves PT from the node it is on to its left-adjacent sister node at the same level of the tree. NX (next) moves PT to the right-adjacent sister of the node it is pointing to, like PV but moving in the opposite direction. UP moves PT to its parent node.

For example, in the structure tree of the example given in the last section, if PT were set on the node that corresponds to the verb *come*, the final constituent of the clause, PV would move it to the NP node, but not to the PROPN node because that is not at the same level. To move the pointer from the final VB node to the PROPN node would require the sequence PV DLC or, what is equivalent in this case, PV DF.

SEARCH CONDITIONS IN POINTER SETTING. Some lists of pointer setting instructions go for a definite number of steps and then stop. One component of Figure 4.09, part of the preparation for agreement checking, sets PT in a single step: (SETPT LAST). Other pointer setting lists, on the other hand, end in a search condition enclosed in parentheses. The last instruction before the condition is repeated either until one of the nodes it sets the pointer on meets the **condition** or until there are no more nodes left that the instruction could move the pointer to. If the pointer cannot find a node that meets the condition, the whole condition of which the pointer setting instructions are a part fails.

In Figure 4.12 the procedure expects to find an auxiliary verb directly dominated by the clause node somewhere in the tree, but it does not know exactly where to expect it because its position is influenced by the clause type, declarative or yes-no question. It therefore uses a pointer setting sequence with a condition (SETPT LAST PV (AUX)). This sets the pointer on the last completed node, the one found in 4.11, then moves backwards using PV as many times as necessary until it reaches an AUX node at that level.

The most commonly used terminating condition is the name of a single feature like (AUX) enclosed in parentheses to make a list. If the pointer is set on a node of the structure tree that has that feature in its feature list, the attempt to set the pointer succeeds and the pointer remains on that node. If none of the nodes the pointer is moved to has that feature, the attempt to set the pointer fails, and it is returned to the position it was in before the search began–at the time the procedure encountered SETPT. More complex LISP expressions can also be used as terminating conditions.

THE STRING POINTER PTW. There is a second pointer, PTW, which is sometimes useful for looking directly at the text rather than at the structure tree. When a conjunction is found, for example, PTW is used to look ahead in the text to see if a matching conjunction is coming up later. The word string pointer PTW is set by an instruction sequence that begins with SETPTW. The same arrangement of instructions in a list that may terminate with a search condition, which activates the instruction ahead of it as many times as necessary, is used after SETPTW.

One group of string pointer instructions set PTW relative to the entire input sentence. N puts PTW on the next word after the current construction, the same word that N would set PT on. SFW puts the string pointer on the first word in the sentence, and SLW puts it on the last word in the sentence.

Another group of string pointer instructions set PTW relative to where the tree pointer PT is at the moment. FW puts the string pointer on the first word of the constituent that PT points to, LW on the last word, and AW on the word that come immediately after the constituent that PT points to.

As with the tree pointer, it is possible to set the string pointer relative to its own current position. NW moves it on to the next word and PW sets it back to the previous word. The PTW pointer is not used in any of the examples given here.

SAVING AND RESTORING POINTER VALUES. Sometimes it is desirable to save the value of a pointer by setting a variable that can be referred to later on in the procedure to restore the pointer, thereby eliminating the trouble of going through two searches for the same node. For the string pointer this can be done straightforwardly by the LISP program command (SETQ X PTW) which associates the current value of PTW with the variable X. (SETQ PTW X) then reassociates the value stored in X with PTW when needed. For the tree pointer, however, it is necessary to save not only the node it is set on but a record of the path by which it got there. To save PT in X it is therefore necessary to use the special command (PTSV X), and to restore the value of PT from X it is necessary to use (PTRS X). These take care of the return path automatically.

WORD REPRESENTATION. The abstract form of a node in the structure tree was given earlier: FE CONF N LAST SM. A node that consists of a single word, however, has a different abstract form, since there is no need to talk about CONF or LAST as its beginning and ending constituents. It is a list of four lists: ((. . .) (. . .) (. . .) (. . .)). The first is called FE, as with other nodes, and is the feature list associated with the word. It is taken from the dictionary with modifications made during the process by the morphological analyzer. For example, a dictionary listing that includes NS for singular number in the listed form of a noun is changed to NPL if the form in the text has a plural ending on it, or a feature INF for an uninflected verb form is changed to PAST if the analyzer recognizes a past inflection.

SMWORD is the second element in the semantic definition of the word. It is taken from a property in the dictionary entry which does not concern us at the moment. The third node, WORD, points to the text form of the word itself. The final node, ROOT, contains the decision of the morphological analyzer: for the text word *did* the root is *do*. Each word in the text is looked up in the dictionary and analyzed as it is read in, so that the full representation of the text is as a list of these four-member lists, one for each word.

To illustrate the use of a specific word node, Winograd's grammar of English stipulates that if a quantifying word is found early in a noun phrase, special action must be taken if it is a negative as in *no bananas* or *none of our friends.* (WORD (CONF LAST)) is the text form of the first word of a noun phrase constituent just parsed. A LISP expression like (EQ (WORD (CONF LAST)) (QUOTE NONE)) succeeds if the word is *none* and fails if it is not.

FEATURE TESTS. Often it is necessary to know what information was found at an earlier stage of parsing. For this there is a test that succeeds if some node has a particular feature and fails if it does not. The most common form of the feature test is (HAS A "B"). It succeeds if the node pointed to by A has the feature B. For example, one part of the complex feature test in 4.23 is (HAS CON "NPL"), shorthand for 'Test for features. Does the construction you are working on have the feature NPL?'.

HAS takes the general form (HAS A Z), where Z is a LISP expression that defines what features are meant. Since the most common use of the test is for a single feature, for which the appropriate LISP expression is (QUOTE B) to distinguish the name of a feature called "B" from the use of B to name some other LISP expression, HAS commonly gives the feature name in double quotation marks, as in the preceding paragraph. Z may be any defining expression, however; (HAS PT (FE N)) would succeed if the features (FE) of the next word in the text (N) were also shared by the node under the pointer (PT). Without the quotation marks around it, Z is always taken as the name of an expression that has features as its value; with them, Z is the name of the feature itself.

(NEXTW R), instead of testing the features of the next word, tests its root to see if it is R that is coming up. It is equivalent to (EQ (ROOT N) (QUOTE R)).

A simple feature test is used in the clause procedure at line 4.25. There (HAS LAST "TRANS") looks at the last node parsed by the current procedure, the verb found in 4.11, to see if a feature represented by the atom TRANS is present there.

COMPLEX FEATURE TESTS. Many of the feature tests in the example are complex. They are built up by using the logical operators AND, OR, NOR, NOT, and COND. The logical operator always comes first in a list and is followed by its terms. AND succeeds only if each of its terms succeeds. OR succeeds as soon as one of its terms succeeds; it does not even try any terms that come after that one. NOR fails if any of its terms succeeds. NOT can be followed by only one term; it succeeds if that term fails and fails

if it succeeds. COND is followed by one or more lists of the form (IF (. . .) THEN (. . .)). The if-then pairs are tried in order; as soon as an IF succeeds, the corresponding THEN is implemented, and no more if-then pairs belonging to that COND are tried. An example is found in 4.13-16.

Procedures themselves behave like a kind of test in that they can lead to a successful RETURN or to FAIL. This is why the first part of a statement is called the condition no matter whether it consists of a construction name, a word class, or a feature test. AND can be used to cause an entire series of tests to be carried out together. Commands, such as the feature assignment commands mentioned in the next section, automatically succeed, with the consequence that they can also be incorporated into such a sequence. OR can be used to try alternatives, since its terms are examined one by one until one of them succeeds.

Figure 4.25 gives a complex condition that involves doing three things: first testing to see if the last node parsed has the feature TRANS, and if it has, looking for a possible NP as object for it. If the NP is found,a further command is given. If any part of the condition fails, since they are joined by AND, the whole thing fails and the second direction is taken.

FEATURE INTERSECTION. (MEET A B) takes two feature lists, A and B, and produces their intersection, a list of all the elements that appear in both A and B. Any element that appears in only one of them is dropped by MEET. MEET succeeds if the lists have at least one element in common, and fails if they do not. Figure 5.16 insures that the noun phrase construction has at least one of the number features NS, NPL, or MASS assigned to the phrase as a whole.

FEATURE ASSIGNMENT. The last aspect of feature management, after pointer management and feature testing, is feature assignment. A procedure reports what it finds by assigning features to nodes in the structure tree. Features are therefore the principal means of keeping track of information as it is found. The semantic analysis procedures make considerable use of the features in the structure tree.

RESTRICTING FEATURES. In analyzing a constituent of a construction, it has already been mentioned that the grammar writer may specify features that must be present in the constituent if it is to be acceptable to the procedure that tried to identify it. These features are added following the name of the unit. Thus the word class names in 4.04 and 4.06 accept not any verb, but only a verb like *have* or *be* that has the feature AUX. This feature restriction on the word that is to be parsed excludes archaic forms like *saw you the king?* but accepts the contemporary form of yes-no questions like *have you seen the chairman?* or *were you writing on this pad?* Features that have to be on the constituent that is being parsed may be added in any number. Unless all of them are present, the parse fails.

CONTINGENT FEATURES. Sometimes, however, features are to be added to a node after it is parsed successfully, without constituting a restriction on that node. These features are given in a contingency list enclosed in parentheses after the name of the unit that is to be parsed and the restrictive features if there are any. In 4.02 the feature SUBJ is added to the noun phrase if it begins a clause, so that instead of appearing in the structure tree as, say, (NP NS), as the NP procedure might leave it, it is expanded to (NP NS SUBJ).

The features that are added to the name of a constituent, whether they are required for the parse to be successful or are to be added in case the parse is successful, are assigned to a variable called REQ (for 'requirements'), where they can be interrogated by conditions as in 9.01.

ADDING FEATURES TO THE CURRENT CONSTRUCTION. The most important feature assignments are those the grammar makes to the construction currently being analyzed (CON). Some information that is found during the analysis of one constituent of a construction often is passed upward in the form of a feature that is attached to that construction as a whole. Thus in 4.02 when a noun phrase is found first in a clause, (PUT CON "DECL") adds the feature DECL to the clause construction as a whole, signifying that it is declarative. The top node in the structure tree has the form (CLAUSE DECL) rather than just (CLAUSE) as a result of 4.02.

(PUT CON "A") adds the single feature A to the currently active construction node. Like HAS, PUT is a streamlined notation for the more cumbersome (PUT CON (QUOTE A)), in which QUOTE (a LISP function) takes the character or string of characters that follows and PUT attaches it to the CON node. In place of a single quoted feature, the element that goes with PUT can be a list of features or indeed a LISP expression of any complexity, as in 6.02. The feature list that results at the CON node is the union of the features on the list that goes with PUT and the features that are already present at the CON node. In other words, the resulting list consists of every element that is in at least one of the two lists.

REM (for 'remove') is the negative counterpart of PUT. (REM CON "A") removes the feature A from the feature list of the currently active node.

(TRNSF A) picks up the entire list of features from the node to which the tree pointer PT is pointing, intersects it with the feature expression A (frequently a quoted list), and adds whatever features result to the feature list at CON by union. If no features can be transferred (that is, if the feature list under the pointer PT has no members in common with the features in A), then TRNSF fails.

Feature transfer is used in Figure 5.05 to give the noun phrase the same number feature as a pronoun at its head has. The condition has three parts joined by AND. First a word of the PRON class is tested for. If it is found, the pointer PT is set on it by (SETPT LAST). Then the features of that pronoun are intersected with the quoted list NS NPL NFS so that any features that are not number features are screened out as

unimportant for this transfer. If the pronoun has any feature of the three, that feature is united with the feature list of the current construction, which in this case is the NP node because TRNSF is activated as part of the NP procedure. Since every pronoun should be tagged for number in the dictionary, the inability of TRNSF to transfer at least one feature to the NP node results in a failure condition that calls attention to a defect in the grammar somewhere.

ASSIGNMENT TO OTHER NODES. It is also possible to put features on the node that PT points to. This is not often needed, but when it is (as for example when evidence has been found that requires a node found earlier to be reinterpreted in a different way), the commands (PUT PT F) and (REM PT F) are available, where F is an expression that specifies the features to be added or removed at the node under the pointer PT. PUT and REM apply only to CON and PT, unlike HAS, which can apply to any node.

(MAKE A B) is needed in grammars for speech production. If B is an expression whose value is a single feature, as in 4.21, MAKE adds it to the node identified by A just as PUT does. If B is a list of two or more features, however, MAKE chooses one of them (at random or by choice of the speaker, depending on what the grammar is used for) and adds it at A, as in 4.18. In parsing MAKE is equivalent to (MEET (FE A) B), and acts as a test.

THE DICTIONARY

Every grammar is written relative to a particular dictionary and every dictionary to a particular grammar. On the whole, a linguist tries to put general statements in the grammar and particular statements in the dictionary. Where the dividing line between the two actually comes, however, is far from clear in principle.

For a procedural grammar the dictionary is set up to contain several kinds of information: grammatical features, irregularities in form, and semantic information. Each word in the dictionary is treated as a LISP atom, and each kind of information for that word is treated as a separate property of the atom.

```
(DEFLIST WORD
   (A (DET INDEF NS))
   (BACK (ADV LOBJ))
   (BE (VB AUX INT INF BE INTRNSL))
   (BOY (NOUN NS))
   (COME (VB INF ITRNS))
   (DO (VB DO INF TRANS AUX))
   (EAT (VB INF TRANS))
   (FIND (VB INF TRANS VPRT))
   (FOR (PREP COMPLEMENTIZER FOR))
```

```
(GEORGE (PROPN NS))
(GREETING (NOUN NS))
(HAVE (VB AUX INF TRANS HAVE))
(HER (DET PRON POSS NS))
(HIS (DET PRON DEFPOSS POSS NS))
(HOWARD (PROPN NS))
(IN (PREP))
(IT (PRON NS SUBJ OBJ))
(MAN (NOUN NS))
(ME (PRON NFS OBJ))
(NICE (ADJ))
(OLD (ADJ))
(OUT (PRT PREP))
(PLACE (NOUN NS VB INF TRANSL2))
(ROSE (NOUN NS))
(SEND (VB INF TRANS TRANSL2))
(SUPPER (NOUN NS))
(TELL (VB REPOB2 INF TOOB2))
(THAT (DET DEM DEF NS COMPLEMENTIZER THAT))
(THE (DET DEF NS NPL))
(THOSE (DET DEM DEF NPL))
(TO (PREP COMPLEMENTIZER TO))
(VILLAGE (NOUN NS))
(WIFE (NOUN NS))
(YET (ADV TIMW))
(YOU (PRON NS NPL OBJ SUBJ))
(YOUR (DET PRON POSS NS NPL)) )
```

Figure 7. Dictionary entries for grammatical features.

Most of the information used by the procedural grammar is stored under the WORD property indicator of each word. It consists of a list of features that identify the word as a member of its word class and give it other characteristics that the grammar can test for. Figure 7 gives a short dictionary listing of grammatical features, again using parentheses to relate things in a completely explicit way.

Irregular word forms have a different kind of information stored under a special property indicator WORD1. This consists of a three-part list: first, the root under which the regular features of the word can be found, second, a list of features that should be added to the features that are entered under that root if this irregular form of it is found, and third, a list of features that should be removed from the list of features given for the root, as in Figure 8.

```
(DEFLIST WORD1
   (ARE (BE (PRES VPL) (INF)))
   (ATE (EAT (PAST) (INF)))
   (DID (DO (PAST) (INF)))
   (FOUND (FIND (PAST EN) (INF)))
   (HAS (HAVE (PRES V3PS) (INF)))
   (TOLD (TELL (PAST EN) (INF))) )
```

Figure 8. Dictionary entries for irregular forms.

For example, Figure 7 shows that *eat* has the features (VB INF TRANS) under its property WORD, indicating that it is a member of the verb class, has the uninflected or infinitival form, and is transitive. Under the property WORD1 of *ate* in Figure 8 is the list (EAT (PAST) (INF)). If *ate* is encountered in the text, this entry says that it has the feature list of *eat* with PAST added to it and INF removed, giving (VB TRANS PAST).

Some words, like *place* in Figure 7, can belong to more than one part of speech category. In order to pick upon only the part of speech that the procedure is asking for, the grammar contains a set of **elimination** lists. For nouns, for example, the form of the elimination list is (DEFPROP NOUN (NOUN MASS NPL NS POSS TIME TIM1) ELIM). The list is intersected with the feature list of any word that is being tested as a noun, so that only those features that match features on the elimination list are considered. If *place* is found by the noun phrase procedure within a phrase like *at your place*, only the features (NOUN NS) will be considered, whereas if it is being tried out in a construction that expects a verb, then only the features (VB INF TRANSL2) will be picked up.

MORPHOLOGICAL ANALYSIS. English has a large number of forms that can be decomposed by an English-specific procedure called a **morphological analyzer.** It recognizes the third person singular form of present verbs like *eats,* regular *-ed* past and participial forms, *-ing* forms, and the comparative and superlative forms of adjectives. All other forms are irregular and have to be entered as such in the dictionary. As each word comes in, it is looked up in the dictionary and decomposed morphologically.

The morphological analyzer for English will not, of course, work for any other language. The linguist who is writing a grammar of another language in procedural form needs to consider what kind of morphological analysis he needs in order to make his grammar work right. There seem to be three potential lines of development for morphological analyzers: (1) A special analysis procedure analogous to the one written for English will be best for some languages. (2) Letter-by-letter parsing routines will take a word apart entirely and decide on its morphological structure, looking up the stem in the dictionary. (3) Complex word forms will be written by some linguists with hyphens separating the morphemes of the word. A special command, first on the agenda of every word analysis procedure, will replace the hypens by spaces, then treat the exploded word as though it were a complete sentence. A regular procedural definition of the word

construction will then go through the word morpheme by morpheme and develop its structure as though it were any other kind of complex unit. The dictionary will not be a dictionary of words, but of roots and affixes. Possibly the second and third approaches can be combined.

For grammars that produce text, a morphological synthesizer is needed. It takes the root form of each word, the features that belong to the word or have been added to it by the grammar, and information about irregular forms and produces the form of the word that fits its context.

EXAMPLES WITH FEATURES

Two examples of parsing with features follow. The first is the same example as the one used to illustrate the basic grammar: *Did George come?* The second is more complex.

As with Figure 1, the implicit starting condition CLAUSE activates the clause unit procedure in Figure 4. The attempt to parse a noun phrase in 4.02 fails, but the attempt in 4.04 to parse a verb with the feature AUX succeeds. Up to this point the action has been identical with that of the basic grammar. In 4.04, however, the condition has two parts joined by AND. The second part adds a quoted list of features to the current construction if the first part is successful. We therefore have a structure tree that already contains more information than the basic grammar gave us:

```
(CLAUSE QUESTION YES-NO        (DID
  (VB AUX DO TRANS PAST)        DID
```

At Q in 4.07 we follow the call to the procedure for noun phrases, which again recognizes *George* as a proper noun in the dictionary. Instead of returning as soon as that is established, however, the NP procedure does some feature manipulation at 5.04. It sets the tree pointer on the node just parsed, then transfers its features to the current construction (the NP) through the filter list (NS NPL). In other words, the feature PROPN is not transferred; it is only the number NS of the noun that is important as fas as the whole noun phrase is concerned. We therefore return to 4.08 with the noun phrase tagged as NS in number. The procedure that activated the call to NP says that if we are successful in finding a noun phrase there, we add the feature SUBJ to it to show that it is the subject of the clause. We also transfer its number feature to the current construction, the clause (4.09), in preparation for agreement checking.

At this point the parse tree looks like this:

```
(CLAUSE QUESTION YES-NO NS     (DID GEORGE
  (VB AUX DO TRANS PAST)        DID
  (NP NS SUBJ)                  (GEORGE)
    (PROPN NS)                   GEORGE
```

At P we find the verb and add the feature MVB to it to indicate that it is the main verb (4.11). The basic grammar finished here; but there is more about English that can be expressed at this point: feature testing to establish aspect, tense, agreement between the subject and the verb, and transitivity.

4.12 and 4.16 use the pointer to establish that the form of the main verb is compatible with the auxiliary if there is one. PUT attaches an aspect feature to the clause in case certain combinations are used; but there are none in this example.

Next 4.17 to 4.20 find the tense of the first verb and transfer it to the clause, and determine whether the tense requires a test for agreement or not. Since *did* is past, there is no agreement check.

Finally at V (4.24) the transitivity feature of the main verb is examined. If it is transitive, 4.25 looks for an object. If it finds one, the feature TRANS is added to the clause as a whole in 4.26. If the verb is not transitive or if it has no object, it is tested for the feature ITRNS, which allows it to appear without an object.

The final analysis using features is

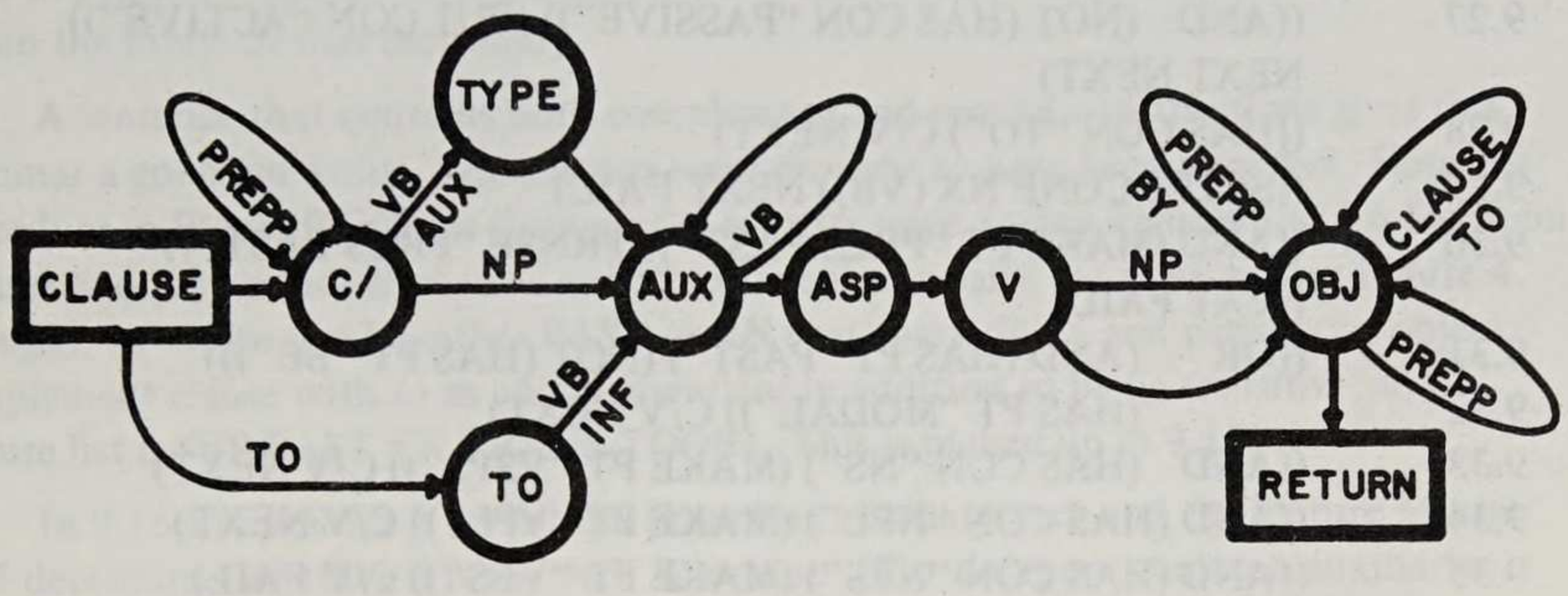
```
(CLAUSE QUESTION YES-NO PAST ITRNS)      (DID GEORGE COME?)
   (VB AUX DO TRANS PAST)                 DID
   (NP NS SUBJ)                          (GEORGE)
      (PROPN NS)                          GEORGE
   (VB INF ITRNS MVB)                     COME
```

A more complex example contains several levels of embedding and uses the agreement test of 4.21 to 4.23. *That nice old man in the village sends his greetings* comes through the procedures as follows:

```
(CLAUSE DECL NS PRES TRANS)                (THAT NICE OLD MAN IN THE
                                              VILLAGE SENDS HIS
                                              GREETINGS)
   (NP DET NS SUBJ)                        (THAT NICE OLD MAN IN THE
                                              VILLAGE)
      (DET DEM DEF NS)                     THAT
      (ADJ)                                NICE
      (ADJ)                                OLD
      (NOUN NS)                            MAN
      (PREPP POSTNOMINAL)                  (IN THE VILLAGE)
         (PREP)                            IN
         (NP DET NS PREPOBJ)               (THE VILLAGE)
            (DET DEF NS NPL)               THE
            (NOUN NS)                      VILLAGE
   (VB TRANSL2 TRANS V3PS PRES)            SENDS
   (NP DET NPL POSS OBJ)                   (HIS GREETINGS)
      (DET PRON DEFPOSS POSS NS)           HIS
      (NOUN NPL)                           GREETINGS
```

EXPANDING THE GRAMMAR

The grammars of Figures 1 to 3 or of Figures 4 to 6 are clearly not the grammar of English one would hope to end up with, though they are not bad for a start. Figure 9 illustrates a slight expansion of the same grammar. It includes clause initial prepositional phrases, passives, perfects, and object complements of one type.




```
9.01    (START CLAUSE (
9.02        ((AND   (HAS REQ "TO") (COMPLEMENTIZER TO)
9.03                (PUT CON "TO")) C/TO C/)
9.04    C/
9.05        ((PREPP (SENMOD)) C/ NEXT)
9.06        ((AND   (VB AUX) (PUT CON "QUESTION YES-NO"))
            C/TYPE NEXT)
9.07        ((AND   (NP (SUBJ)) (PUT CON "DECL")
9.08                (SETPT LAST) (TRNSF "NS NPL NFS")) C/AUX FAIL)
9.09    C/TYPE
9.10        ((AND   (NP (SUBJ))
9.11                (SETPT LAST) (TRNSF "NS NPL NFS")) C/AUX FAIL)
9.12    C/TO
9.13        ((VB INF) C/AUX FAIL)
9.14    C/AUX
9.15        ((VB) NEXT C/ASP)
9.16        ((AND (SETPT LAST) (SETPT PV (AUX))
9.17                (OR (AND (HAS PT "BE") (MAKE (SETPT LAST)
                    "EN ING")
9.18                        (OR (AND (HAS PT "EN")
9.19                                 (REM PT "PAST")
9.20                                 (PUT CON "PASSIVE NOAGENT"))
9.21                            (AND(HAS PT "ING")
9.22                                 (PUT CON "PROGRESSIVE")) ))
```

```
9.23                    (AND    (HAS PT "HAVE") (MAKE (SETPT LAST)
                        "EN")
9.24                            (REM PT "PAST")
9.25                            (PUT CON "PERFECT"))  )) C/AUX C/AUX
                                C/ASP)
9.26    C/ASP
9.27        ((AND   (NOT (HAS CON "PASSIVE")) (PUT CON "ACTIVE"))
            NEXT NEXT)
9.28        ((HAS CON "TO") C/V NEXT)
9.29        ((SETPT CONF NX (VB)) NEXT FAIL)
9.30        ((AND (MAKE PT "PRES PAST") (TRNSF "PRES PAST"))
            NEXT FAIL)
9.31        ((OR    (AND (HAS PT "PAST") (NOT (HAS PT "BE")))
9.32                (HAS PT "MODAL")) C/V NEXT)
9.33        ((AND   (HAS CON "NS") (MAKE PT "V3PS")) C/V NEXT)
9.34        ((AND (HAS CON "NPL") (MAKE PT "VPL")) C/V NEXT)
9.35        ((AND (HAS CON "NFS") (MAKE PT "VFS")) C/V FAIL)
9.36    C/V
9.37        ((AND   (HAS LAST "TRANS") (NP (OBJ))
9.38                (PUT CON "TRANS")) C/OBJ NEXT RETURN)
9.39        ((AND   (HAS LAST "ITRNS")
9.40                (PUT CON "ITRNS")) C/OBJ C/OBJ RETURN)
9.41    C/OBJ
9.42        ((AND   (HAS CON "NOAGENT") (NEXTW BY) (PREPP (AGENT))
9.43                (REM CON "NOAGENT")) C/OBJ NEXT RETURN)
9.44        ((AND   (HAS (SETPT LAST PV (VB)) "TOOB")
9.45                (CLAUSE TO (SEC)) (PUT CON "TOOB")) C/OBJ NEXT
                    RETURN)
9.46        ((PREPP (SENMOD)) C/OBJ FAIL RETURN)  ))
```

Figure 9. English grammar expanded to include voice and object complements with *to*. For NP and PREPP use Figures 5 and 6.

Figure 9, as can be seen, is not very different from Figure 4. A few arcs are added and more conditions are inserted, yet the grammar is basically the same. No matter how much more comprehensive a version of the grammar is, it still has to incorporate the things Figure 4 asserts about English, since they are part of the language.

The state names and corresponding labels in Figure 9 follow a useful principle of label construction suggested by Woods and Kaplan (1971). Although the labels in procedures are essentially arbitrary, it helps the grammar writer if he can adopt a naming scheme that will remind him what part of the grammar he is in. Woods's naming scheme is as follows: each label consists of two parts separated by a slash. The first part

identifies the construction being worked on. The second part gives an idea of what has been found in that unit so far. For example, C/AUX is part of the clause procedure, and comes at a point where the auxiliary verb of the clause, if there is one, has been identified regardless of which path into that state was taken.

The dictionary can be expanded straightforwardly by analogy with the forms already given in Figures 7 and 8. In the examples that follow no attempt is made to remain within the limits of that dictionary.

A sentence that contains both complement and passive constructions gives this grammar a good workout: *The tourists were thought to have been stranded.* Here the procedure in Figure 9 follows through *the tourists were* (using Figures 5 and 6 for noun phrases and prepositional phrases as before) in substantially the same way as Figure 4. *Thought*, however, can be either PAST or EN (past participle), and permits an object complement clause with *to* as complementizer in addition to being transitive, so its feature list is (VB PAST EN TRANS TOOB). This is picked up in 9.15.

In 9.16 the pointer is set on the preceding auxiliary verb and the feature test in 9.17 determines whether it is the verb *be* or not. (For the more common auxiliaries it is convenient to list the root of the auxiliary as one of its features, even though it is redundant to do so.) The auxiliary is *be*, so the test in 9.18 looks for an *en* form of the last verb. Upon finding it, 9.20 deletes PAST and adds to the current node the features PASSIVE for voice and NOAGENT in case no *by*-phrase is found later. Then it returns in 9.25 to C/AUX to see if there are any other verbs in the sequence. The structure tree at this point has this form:

```
(CLAUSE DECL NPL PASSIVE NOAGENT          (THE TOURISTS WERE
                                           THOUGHT
  (NP DET NPL SUBJ)                       (THE TOURISTS
    (DET DEF NS NPL)                       THE
    (NOUN NPL)                             TOURISTS
  (VB BE AUX PAST VPL)                     WERE
  (VB EN TRANS TOOB)                       THOUGHT
```

Back at C/AUX the attempt to parse another verb fails. This takes the procedure to C/ASP. 9.27 would tag the clause as ACTIVE if it were not PASSIVE. The rest of that section of the procedure operates as in Figure 4 to test for agreement. At C/V, even though the main verb is TRANS, there is no object, so the procedure mores on to C/OBJ. The statement at 9.42 fails on its second component, since the next word is not *by*.

The last verb parsed is located by the tree pointer in 9.44. It has the feature TOOB, so in 9.45 the clause procedure is activated recursively with the requirement that the clause it finds have the feature TO.

The first statement of the clause procedure in this grammar tests to see if the embedded clause to be parsed is a *to*-clause without subject. It has been given the

feature requirement TO, so the word class feature COMPLEMENTIZER TO looks to see if the next word is *to*. The next part of the statement, 9.03, adds the feature TO explicitly to the structure tree for the clause construction in order to satisfy the requirement that the construction found by this procedure have that feature. Next at C/TO comes a call to parse a verb in the infinitive form and go to C/AUX. From there on the action of the procedure is as before. The final parse tree is:

(CLAUSE DECL NPL PASSIVE NOAGENT PAST TOOB)	(THE TOURISTS WERE THOUGHT TO HAVE BEEN STRANDED
(NP DET NPL SUBJ)	(THE TOURISTS)
(DET DEF NS NPL)	THE
(NOUN NPL)	TOURISTS
(VB BE AUX PAST VPL)	WERE
(VB EN TRANS TOOB)	THOUGHT
(CLAUSE TO PERFECT PASSIVE NOAGENT SEC)	(TO HAVE BEEN STRANDED)
(COMPLEMENTIZER TO)	TO
(VB HAVE AUX INF)	HAVE
(VB BE AUX EN)	BEEN
(VB TRANS EN)	STRANDED

On the other hand, this grammar gives an erroneous analysis for something like *At the airport the passengers agreed to a search of their luggage.* The verb *agree* is a TOOB verb in sentences like *The police agreed to release the demonstrators.* Here, however, the word after *to* is not a verb in the uninflected form, so the attempt to parse the rest as an object complement clause fails. Instead it is treated as a prepositional phrase with *to* as the preposition, and is added by 9.46 as a sentence modifier. This analysis is completely consistent with the grammar in this form, but any linguist will reject it. The next stage of modification of the grammar, therefore, would be to develop the class of particle verbs like *agree to*, and possibly (though this might better be handled by semantic procedures) to elaborate the function of deverbal nouns like *search.*

GLOSSARY

Here is a complete list of the 46 terms (exclusive of feature names) that make up the vocabulary of procedural grammars, together with the linguistic terms that are used in explaining some of them.

AND – logical relator for complex conditions. Succeeds if all its arguments succeed.
AW – word after the constituent that PT points to. Used with PTW.
backup – on FAIL, undo the analysis back to a state from which at least one path leads that has not yet been tried.
CON – the construction currently being analyzed. Named in the START state of the network in which the statement that refers to it is located. Winograd's C.
COND – set of IF-THEN pairs.
CONF – first constituent of CON. Winograd's NB.
condition – the part of a statement that determines success or failure.
constituent – one element of a construction.
construction – an internally complex unit of the grammar.
DF – down, first: move PT to the first constituent of the construction it points to at the moment.
dictionary – listing of root form of each word, features, irregular inflected forms, and in a complete dictionary, semantic information.
directions – next state name, NEXT, FAIL, or failure message in parentheses.
DL – down, last: move PT to the most recently formed constituent of the construction it points to at the moment.
DLC – down, last, complete: move PT to the most recently formed constituent of the construction it points to at the moment of which CON is not itself a part.
FAIL – the construction cannot be parsed in this way, so back up.
FE – feature list of CON or one of its constituents.
feature – linguistically important property of a word or construction, represented by a single atom.
feature expression – a LISP expression that gives the name of a feature as its value instead of simply quoting it.
FW – first word of the construction PT points to. Used with PTW.
HAS A B – feature test. Succeeds if the constituent at A has the feature B. Winograd's IS, ISQ, CQ, NQ.
IF – first member of a COND pair. If the expression that follows it succeeds, do THEN and finish with the condition as a whole. In standard LISP the expression is not preceded by IF.
LAST – most recently completed constituent of CON or of one of its constituents. Winograd's H.
LW – last word of the construction PT points to. Used with PTW.
MAKE A B – for production, choose any one of the features listed in B, then assign that feature to the constituent at A. For parsing, equals (MEET (FE A) B). Not used by Winograd.

MEET A B – intersect lists A and B. Fails if they have no elements in common.
N – the word that follows LAST, not yet processed.
network – the state-and-path representation of a construction.
NEXT – try the next path. Winograd's NIL.
NOR – logical relator for complex conditions. Fails if any of its arguments succeeds.
NOT – logical relator for complex conditions. Fails if its one argument succeeds, succeeds if it fails.
NW – move PTW to the next word.
NX – move PT to the next constituent at the same level.
OR – logical relator for complex conditions. Succeeds as soon as one of its arguments succeeds without trying the rest.
parentheses – used in pairs to group things that go together: the parts of a construction definition, a rule, or a complex condition. Used after a construction name in a condition to enclose names of features that are to be added to that construction if it succeeds, like (SUBJ). Used at the end of a pointer setting sequence to enclose the name of a feature that a repetitive instruction is to look for.
path – transition from state to state depending on meeting a condition.
procedure – instructions for analyzing or building a construction.
PT – pointer to one node of the structure tree.
PTRS X – restore pointer value from X.
PTSV X – save pointer value in X.
PTW – pointer to one word in the string.
PUT – add a feature to CON or PT. Winograd's FQ, F, PUTF.
PV – move PT to the preceding constituent at the same level.
PW – move PTW to the preceding word.
quotation marks – written in pairs around a feature or a list of features to name them directly.
REM – remove a feature from CON or PT. Reverse of PUT. Winograd's RQ, R, REMF.
REQ – list of required features like AUX and features to be added like (SUBJ). Winograd's REST.
RETURN – final state of a construction. Upon reaching it, if the required features in REQ have been found, it permits a path to be taken at the next level up.
ROOT – the uninflected or dictionary form of a word.
rule – a condition and the corresponding directions, represented by a path in a network.
SETPT – begin a tree pointer setting sequence. Winograd's *.
SETPTW – begin a word string pointer setting sequence. Winograd's **.
SFW – first word in input sentence. Used with PTW.
SLW – last word in input sentence. Used with PTW.
SM – semantic information about a construction.
SMWORD – semantic information about a word.
START A – first state of construction A. Winograd's PDEFINE'
state – point in putting a construction together at which one or more new conditions can be tried. Represented by a label in a procedure, a circle or square in a network.

START and RETURN are states with special properties.

THEN – identifies an instruction to be followed in a COND condition if the IF part immediately ahead of it succeeds. In standard LISP the expression is not preceded by THEN.

TRNSF A – transfer to CON all the features of the constituent that PT points to that match features in list A.

UP – move PT to the construction of which the constituent that PT is currently on is a part.

WORD – form of a word actually used in text, possibly inflected. If uninflected, identical to ROOT.

word class features – basic grammatical identifiers of words: ADJ DET NOUN PREP PRON PROPN VB.

FEATURES USED

This is a list of the grammatical features to which the simple grammar makes reference. English as a whole requires us to recognize many other features as well.

ADJ – adjective, a word class.

AUX – auxiliary, a kind of verb that can precede another.

BE – feature used to provide quick identification of *be*. Sounds redundant, but saves time.

CLAUSE – construction name.

DECL – declarative, one type of clause.

DEF – definite, one type of determiner.

DET – determiner, word which gives the logical status of a noun phrase.

EN – past participle, controls verb inflection.

HAVE – quick identification of *have*.

INDEF – indefinite, one type of determiner.

ING – present participle or gerund, controls verb inflection.

ITRNS – intransitive, a verb that does not require an object.

MASS – mass, one kind of noun like *water*.

MODAL – an uninflectable auxiliary verb like *could*.

MVB – main verb of a clause.

NFS – first person singular noun or pronoun like *I, me, my*.

NOUN – noun.

NP – noun phrase, construction name.

NPL – plural of a noun, proper noun, or pronoun.

NS – singular of a noun, proper noun, or pronoun.

OBJ – object of a clause.

PAST – tense feature on verbs.

PERFECT – perfect, tense-aspect feature expressed by a complex verb sequence.

PREP – preposition, word class feature.
PREPP – prepositional phrase, construction name.
PREPOBJ – prepositional object, attached to noun phrase.
POSS – possessive, a special kind of determiner.
PRESENT – tense feature on verbs.
PROGRESSIVE – progressive, tense-aspect feature expressed by a complex verb sequence.
PRON – pronoun, word class feature.
PROPN – proper noun, word class feature.
QUESTION – question, one type of clause.
SUBJ – subject of a clause.
TRANS – transitive, a verb that takes an object.
VB – verb, word class feature.
VFS – first person singular verb, only *am*.
VPL – plural verb, same as the root form except for *be*. Includes the form that goes with singular *you*.
V3PS – third person singular verb, different from other forms in present tense.
YES-NO – one kind of question.

CHAPTER 2

GURENNE CLAUSE STRUCTURE

Nancy Schaefer

Examination of how the clause structure of Gurenne[1] fits into discourse has resulted in a grammar which follows the procedural model developed by Winograd (1971).[2]

This chapter helps demonstrate that the procedural model is useful in linguistic analysis as a means of building a grammar rapidly and of testing it as it is being built.[3] In order to model the actual operation of understanding and producing language as closely as possible, the linguist can look at a grammar as a set of procedures for analyzing what he hears. This results in a grammatical breakdown of text to which semantic and referential information can also be added.

A procedural grammar is expressed in two equivalent ways. First, networks provide a pictorial representation for each type of construction, as in Figures 1 through 6. Each network is represented by circles and rectangles, called **states**, which are connected by arrows, called **arcs**. The arcs are labeled and serve as paths between states. Each state after the first is reached by successfully meeting the **condition** on the arc which leads to it. The condition may be simple or complex, and may involve the recursive activation of other procedures.

Second, a compact prose form permits a detailed statement that embraces all the restrictions of the grammar. This summary of the grammar describes the exact procedures to be followed, gives all the conditions and features which determine the path for the individual clause, and provides for a structural representation of what is found (Figures 7 through 9). Each grammar goes along with an explicit dictionary (Figure 10) that gives word class information and grammatical features. The shorthand that is used to make the statements compact is described in Chapter 1.

---

[1]Gurenne is classified as a member of the Moore-Dagbani subgroup of Gur languages. It is spoken in the Bolgatanga district of the Upper Region of Ghana. Language data used in this study are from the village of Zuarungu, located four miles east of Bolgatanga. The author, under the auspices of the Institute of Linguistics, has been engaged in language research on Gurenne since April 1972. This paper is the outcome of a linguistic workshop held in Ghana from March to June 1973 under the direction of Joseph E. Grimes, partially funded by National Science Foundation grant GS-3180ANo.1.

[2]The version of procedural grammars used in this paper was adapted for use in field studies by Grimes on the basis of Winograd's paper (1971). The equivalent network representation is from Woods (1970).

[3]It has already been used for Lamnsok of Cameroun by Grebe and for Godie of Ivory Coast by Gratrix.

Successful transitions from one state to another permit a construction to be analyzed. The record of the arcs followed and the states reached is embodied in a **structure tree** which represents the grammatical breakdown of the construction.

CLAUSE

Clause patterns are described first since they have been the focal point of field work. They are given in network form in Figure 1 and in the prose procedures of Figure 7. Other constructions are taken up later.

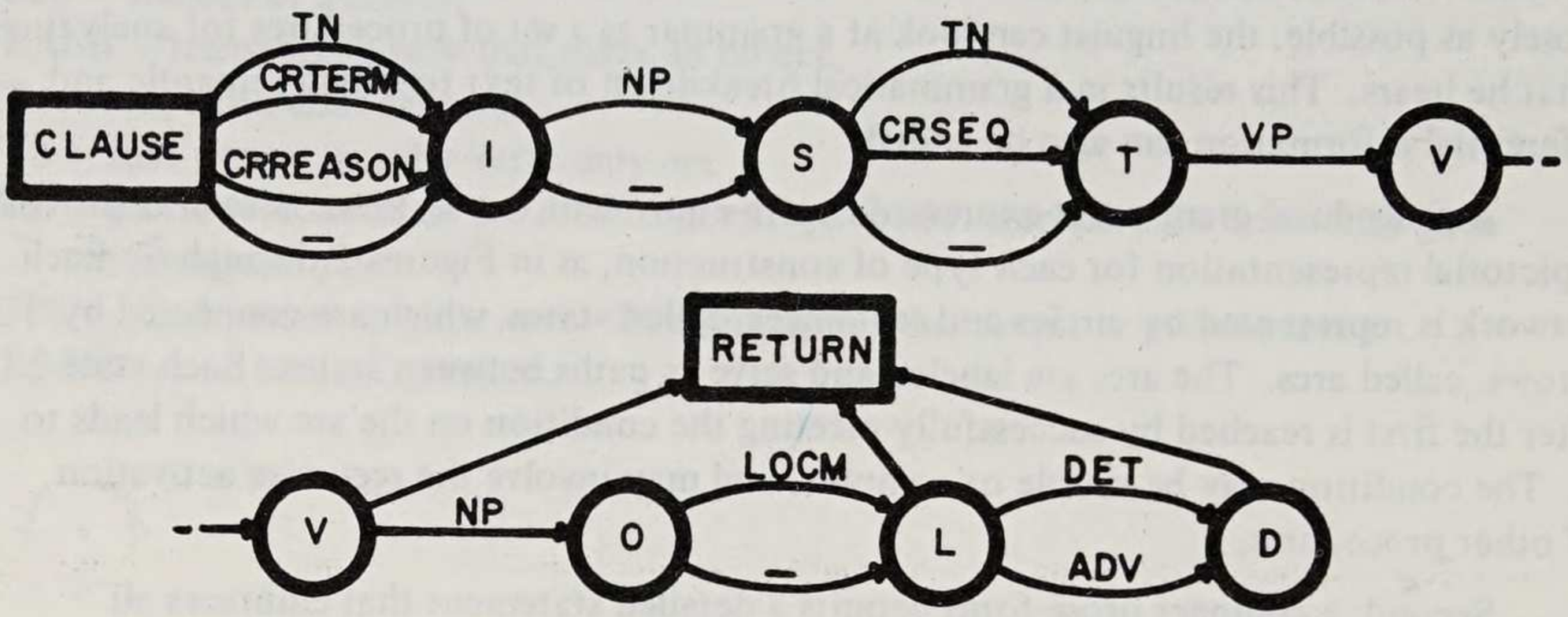


Figure 1. Clause network.

The clause may begin with any one of the arcs with the conditions TN, CRTERM, or CRREASON leading to state I. The fourth arc permits a transition, even if none of the other conditions are met. Then an arc with the condition NP leads to state S. Either arc TN or arc CRSEQ leads to state T. Arc VP leads to state V; arc NP to state O; arc LOCM to state L. At state L, either arc DET or arc ADV leads to state D. The clause can run out of words and RETURN successfully from any one of the states V, O, L, or D.

CLAUSE TYPES. Most clauses in Gurenne discourses are declarative clauses which can be divided into two groups: those with subjects and those without subjects. Those with subjects have been designated as **independent** clauses (I or INDEP). Tense is marked only in independent clauses. Those without subjects take as their subject referent that of the preceding independent clause,[4] so they have been designated as **dependent** clauses (D or DEP).

---

[4]Karl Grebe (chapter 4) discusses the difference between dependent constructions and complements in detail.

CLAUSE STRINGS. As many as four dependent clauses have been found following an independent clause (see Figure 6): **tì ǹ dá péé! bá' áhí lá**[5] (reason I past wash finish textual-determiner (I)) 'so that I finished washing', **tì tì dá ví lɨ́b zúárú nɨ́, sígí ágɨ́ cɨ́ltór mɔ́gódé** (reason we past return go-back Zuraungu here (I), go-down Agriculture dam (D)) 'therefore we returned to Zuarungu, and went down to the Agriculture dam', **tí dá ńyá tárɨ́, tì lɨ́bɨ́! lá! sólmíá yídí nɨ́, wì zé'ílí lóódí lɨ́ bì lɨ́, gè nýá! kúlí** (we past then take (I), reason go-back specific-determiner white-man house here (D), make stand lorry textual-determiner there textual-determiner (D), terminal then go-home (D)) 'therefore I took it, and therefore went back to the residency, left the lorry there, and then went home', **sébà núré bá dám, ètà pótígúré étógó, gè dé'íní dé'ímí! yɨ́mí yɨ́mí, wá'áré, gè yúúné** (some drink their beer (I), behaving satisfaction behavior (D), and play plays different (D), dance (D), and sing (D)) 'some drank their beer, did what they wanted to do, and played different plays, danced, and sang', **gé à ńyókɨ́ zı̨hó! bóyímɨ́, ǹyà tárɨ́! kúlí nɨ́, dɨ̀kì̀ pí'ó lé, ǹyà vúgɨ́** (and she catch fish different (I), then take go-home here (D), take basket textual-determiner (D), then cover (D)) 'and she caught a different fish, then took it home, took the basket, and covered it'.

CLAUSE RELATORS. There are three clause relators: **gé!** (to which we assign the grammatical feature CRTERM) the terminal clause relator, **tì** (CRREASON) the reason clause relator, and **ǹyá** (CRSEQ) the sequential clause relator.

Gé! can begin both independent and dependent clauses. It conveys the idea of something additional or consequential to what has been mentioned before. **Gé!** occurs as many as two times in the final clauses of a paragraph: **gè dé'ímɨ́ dá búí** (**terminal** plays past were (I)) 'in addition, there were plays', **gè dé'íní dé'ímí y ́mí y ́mí** (**terminal** playing plays different (D)) 'and in addition, playing different plays', **gè yúúné** (**terminal** singing (D)) 'and singing', **gé! ńyá kúlí** (**terminal** then go-home (D)) 'so then went home', **gé à ńyók ́ zı̨hó! bó yɨ́mɨ́** (**terminal** she catch fish different (I)) 'so she caught a different fish', **gé! 'íní wàn tàní** (**terminal** I will bring (I)) 'so I will bring', **gé! ńyá báhé lá! zı̨hú lé** (**terminal** then leave specific-determiner fish textual-determiner (D)) 'so then left the fish', **gé bà kà fą́ą́rɨ́** (**terminal** they not take-it-from-you (I)) 'so they would not take it from you'.

Tì can also begin both independent and dependent clauses. Any clause which begins with **tì** either explains the reason for the action in the preceding clause or gives its result: **tì tì kìŋ bólígí** (**reason** we go Bolga (I)) 'therefore we went to Bolga', **tì tà dá! ví lɨ́b zúá!rú nɨ́** (**reason** we past return go-back Zuarungu here (I)) 'therefore we returned here to Zuarungu', **tì tà dá! pé à lóódí** (**reason** we past wash his lorry (I)) 'so that we could wash his lorry', **tì lɨ́ bɨ́ là sòlmíá yídí nɨ́** (**reason** return specific-determiner white-man house here (D)) 'therefore returned here to the residency', **tì bà búgúgí** (**reason** they wet (I)) 'so that they became wet', **tì bà kílím tɔ́nɔ́** (**reason** they

[5]Robert Schaefer (ms) describes the phonological system of Gurenne. It has consonants /p, t, ty, k, kp/; /b, d, g, gb/; /f, s/; /v, z/; /h, '/; /m, n, ny, ŋ/; /l/; /w, y/; and /ɣ/. Vowels are /a, e, ɛ, ɨ, i, u, o, ɔ/. Two tones are distinguished: high (́) is automatically lowered after low (̀), and in some places (!) where an underlying low has dropped out or changed to high.

became mud (I)) 'so that they became mud', **tì mà yídí ǹ dè nà'bà yídí lɨ́** (**reason** my house it is the chief house (I)) 'because my house is the chief's house', **tì bíá wóò, là mà dùmà wóò, là sɔ̀ dùmà wóò, tì bà dá!dí'** (**reason** child all, the mother people all, the father people all, **reason** they past eat (I)) 'therefore all the children, all the mothers, all the fathers, they all ate',

The third clause relator, **ǹyá**, is the sequential clause relator, roughly equivalent to the English word 'then'. Normally in any clause it precedes the verb, and in an independent clause the pronoun is in the initial position: **tì dá!ńyá tárɨ́** (we past **then** bring (I)) 'we then brought it', **tà ńyá yìm bérí** (he **then** saw tomorrow (I)) 'then daybreak came', **tà ńyá zɔ́'** (he **then** ran (I)) 'he then ran'. In the next example, **ǹyá** has changed places with the pronoun **tà**: **ǹyá dá!tá péé** (**then** past we-to-ic wash (I)) 'then we washed it'. The action in this clause has been unsuccessfully attempted before in the text, so the sequential successful attempt is emphasized at this point. A dependent clause, which has no subject, can begin with **gé!**, as in: **gé!ńyá kúlí** (so **then** go-home (D)) 'so then went home', **gé!ńyá báhé lá!zíhú lé** (and **then** leave specific-determiner fish textual-determiner (D)) 'and then left the fish'. Without another clause relator **ǹyá** comes first in dependent clauses, as in **ǹyá dɩ̀ kì zíhu le** (**then** take fish textual-determiner (D)) 'then took the fish', **ǹyá tárɨ́!kúlí** (**then** take go-home (D)) 'then took it home', **ǹyá vúgí** (**then** cover (D)) 'then covered it', **ǹyá sígí!kó'óm** (**then** go-down water (D)) 'then went down to the water', **ǹyá tárɨ́!kúlí nɨ́** (**then** take go-home here (D)) 'then took it home'.

In addition to the dependent clauses beginning with **tì**, **gé!**, and **ǹyá**, there are dependent clauses which begin with the verb, because they lack both a subject and a clause relator. In the summary of the grammar, clauses which begin with a verb are labeled SERIAL: **sígí ágɩ́cílɨ́tór mógódé** (**go-down** Agriculture dam) 'went down to the Agriculture dam', **wɨ̀ zé'ɨ́lí lóódí lɨ́ bɨ̀ lɨ́** (**make stand** lorry textual-determiner there textual-determiner) 'left the lorry there', **zúm bà púwáy** (**climb** them inside 'climbed in it', **dógó! kúdíkúdí nɩ́nɔ́** (**cook** pig meat) 'cooked pork', **dógó! kí'ìnì nɩ́nɔ́** (**cook** guinea-fowl meat) 'cooked guinea fowl', **ètà pòtɩ́ gúdé étógó** (**behaving** satisfaction behavior) 'did what they liked to do', **wá!'áré** (**dance**) 'danced', **tárɨ́ kíɲ kólégá** (**take-go** river) 'took it to the river', **mɔ́ní! ságábó** (**make** porridge) 'made porridge', **!dítí** (**eat**) 'ate it', **!zéɨ̀lí** (**stand**)'keep it'.

Serial clauses are different from the auxiliary verbs discussed in the next section. A serial clause does not have to be preceded by another verb immediately, and verbs and their auxiliary verbs are treated as a unit in clauses.

VERB PHRASE

The verb phrases are described in detail in the network of Figure 2 and in the VP procedure of Figure 9.

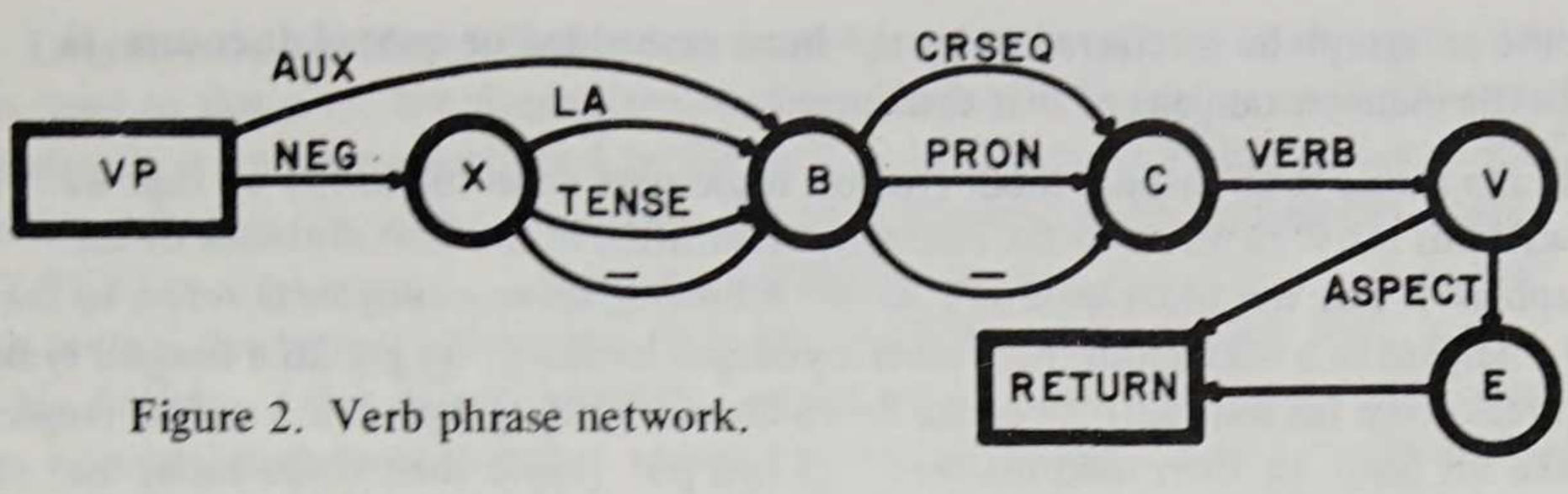


Figure 2. Verb phrase network.

AUXILIARY VERBS. Most verb phrases have only one verb, but a few have two which seem to act as a unit. These have been designated auxiliary verbs. They fall into two groups. In the first group, the first and sometimes the second verb is in a shortened form: **ví lí̀b** (return go-back) from **víhí** 'return' and **lí̀ bí̀** (go-back), **wɨ́ zé'ɬ́lí** (make stand) from **wɨ̀m** (make), **zí'mé!bíhá** (stand watch) from **zé'ɬ́lí** (stand), **zí'í tùm** (sit work) from **zí'írí** (sit), **wá' pú'úhé** (dance greet) from **wá'áré** (dance).

The second group of auxiliary verbs consists exclusively of the motion verbs **kíŋ** 'go' and **kúlí** 'go-home' preceded by the verb **tárɨ́** 'take': **tárɨ́ kúlí** (take go-home) 'take it home', **tárɨ́ kíŋ** (take go) 'take'.[6]

The tense and aspect systems of Gurenne remain to be spelled out in detail. In general, the tense indicator comes between the subject and the verb and the aspect indicators follow the verb. Both may possibly be indicated as well in the form of the verb stem and in its tone. Here they are given the features of TENSE and ASPECT without being distinguished further.

NOUN PHRASE

Noun phrases are described in the network of Figure 3 and in the NP procedures of Figure 8.

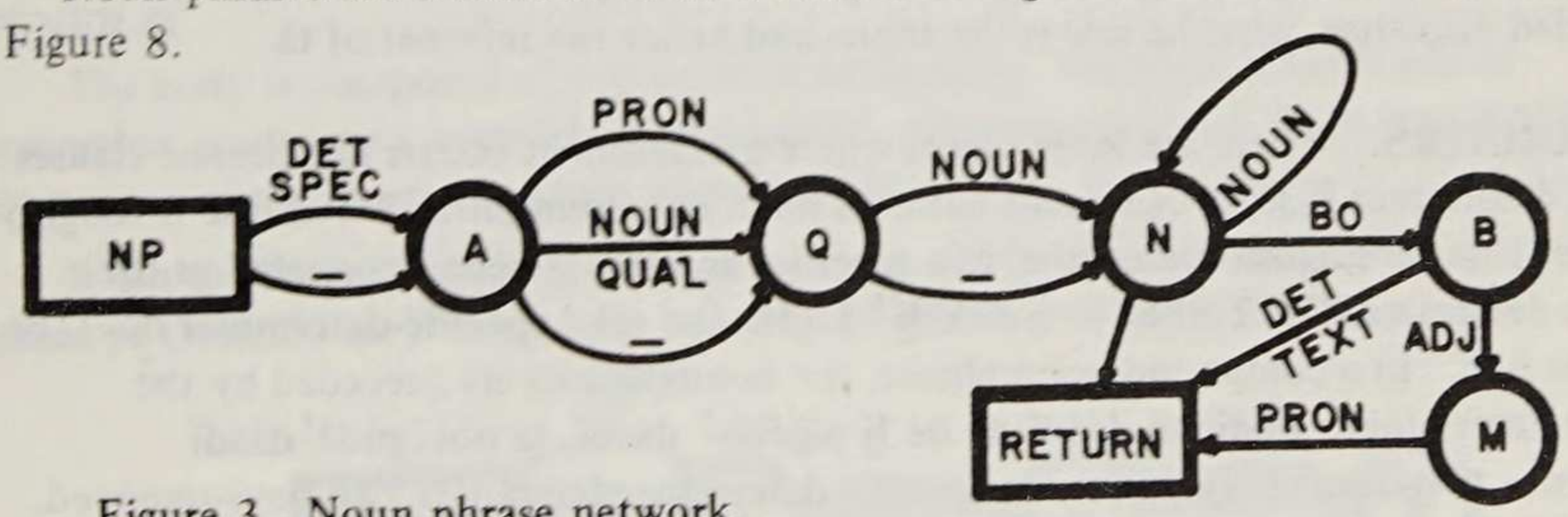


Figure 3. Noun phrase network.

MAIN PARTICIPANT PRONOUN. The main participant (or participants when they act as one) in a discourse is usually referred to by the pronoun **tà** 'topic'. When a new paragraph about the main participant begins, he is reidentified by a noun, but thereafter

[6]Philip Hewer (ms) calls this construction a verb cluster and describes it in detail.

within the paragraph he is referred to by **tà**. In an embedded or quoted discourse, **tà** refers to the main participant of that discourse.

In the clause **tì tà dá!pé à lóódí** (reason **topic** past wash his lorry) 'so that we could wash his lorry' **tà** refers to the characters identified in the first sentence of the paragraph as 'I' and 'my older brother'. In the following three examples **tà** refers to the main participant in a text about 'my father's younger brother': **tà gbí'ídí̵ à dàngòn tyáká** (**topic** breakdown his wall old) 'he broke down his old wall', **tà ǹyá dɩ́k à ná'má** (**topic** then take his feet) 'he then used his feet', **tà ǹyá pɔ́'** (**topic** then shape-balls) 'he then made mud balls'.

The remaining examples in this section are from a text in which a farmer is the main character. He is introduced in the opening sentence of the text. In the first paragraph he is referred to by the pronoun **tà**: **tà dɩ́kí pí'ó** (**topic** take basket) 'he took a basket'. In the second paragraph his wife is introduced in relation to the farmer by the use of a possessive construction: **tà pógá ńyá pání̵** (**topic** wife then come) 'then **his** wife came'. She is the main participant in this paragraph and the husband is not mentioned. However, she is only a secondary participant as far as the whole discourse is concerned,[7] so she is referred to by the unmarked third person singular pronoun **à**. In the third paragraph, the husband comes back on the scene with a noun identification in the first sentence: **à sírí mí'· kíŋ̵í̵** (her **husband** come) 'her husband came'. The husband is again referred to by the pronoun **tà** throughout the rest of the paragraph: **tà yɩ́lí!ná'bá** (**topic** tell chief) 'he told the chief', **tà ǹyá yím!bérí** (**topic** then saw tomorrow) 'daybreak came'. There are two examples within this paragraph, however, in which **tà** does not refer to the farmer. When the farmer speaks, his speech is an embedded discourse: **tì 'íŋ̵í̵ ǹ yélí!sé'lá . . . tà!yúní̵ . . . tà yúm tì bà ké!líhé** (reason body I saw **something . . . it** sing . . . **it** sing reason they hear) 'because I myself saw something . . . it was singing . . . it will sing so that they will hear'. Within that embedded discourse, what he saw is the topic, and hence the referent of **tà**.

DETERMINERS. There are three places where a determiner occurs in Gurenne clauses. First, a determiner **lɩ̵́, lé, lá** can occur before a noun in a noun phrase. This use is roughly equivalent to the English use of 'the' as a specifier and so has been designated as the **specific** determiner (DET SPEC): **à dɩ́k!lɩ̵́! kúdé** (he take **specific-determiner** hoe) (he took the hoe'· In a compound noun phrase, the noun phrases are preceded by the specific determiner: **búdínyá dáádí dá dé lɩ̵́ pɩ̀pélɩ̵́n! dáádí, lá pòtɩ́gúdé!dáádí** (Christmas two-days-removed past is **specific-determiner** happiness two-days-removed, **specific-determiner** satisfaction two-days-removed) 'Christmas day, it was happiness day, satisfaction day'. **tì bíá wóò, là mà dùmà wóò, là sɔ̀ dùmà wóò, tì bà dá dí'** (reason child all, **specific-determiner** mother people all, **specific-determiner** father people all, reason they past eat) 'so all the children, all the mothers, all the fathers, they all ate'.

[7]Mona Perrin (ms) discusses primary, secondary, and tertiary characters in discourse.

Determiners can also follow either a noun in a noun phrase or an entire clause. When used in this way, the determiner modifies the noun phrase or an action which must already have been mentioned in the text or in the conversation which immediately preceded it.[8] This use of the determiner has been designated the **textual** determiner (DET TEXT): **tì bà búgúgí . . . bàn bùgù bá!áhí! lé** (reason they wet . . . they wet finish **textual-determiner)** 'therefore they became wet . . . when they were wet . . .', **tì tì kiŋ bólígí . . . tì ǹ dá! pá! bólígí lí** (reason we go Bolga . . . reason I past reach Bolga **textual-determiner)** 'for that reason we went to Bolga . . . so I reached Bolga', **ǹyá dɪ́kí zi̱hù lé** (then took fish **textual-determiner)** 'then took the fish' (the one mentioned in a song which precedes the text).

THE CLAUSE IN DISCOURSE

This section of the paper sketches the higher level networks that provide the discourse context for clauses. It is incomplete, but serves to give an idea how the total system works.

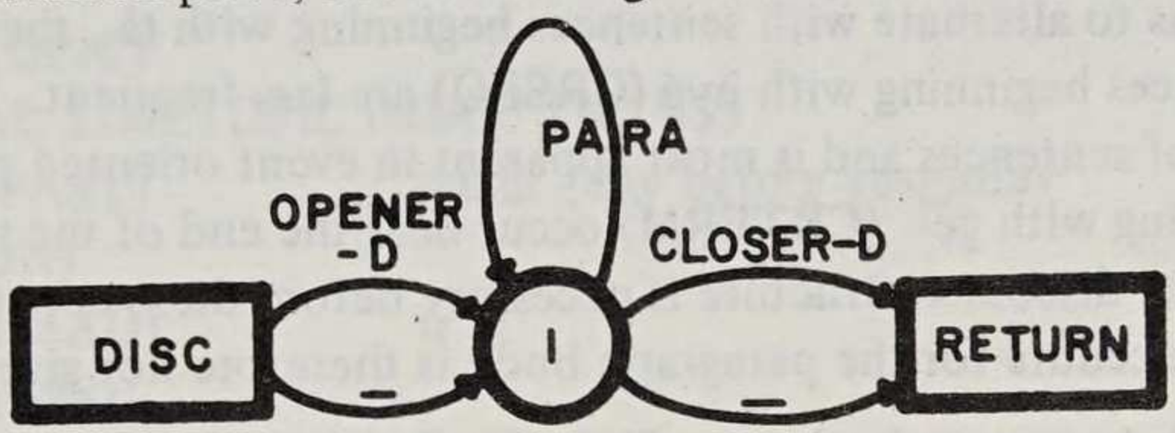


Figure 4. Discourse network

The discourse has three basic parts: an opening formula, the body, and a closing formula. The opening formula is an optional sentence: **tì sɔ́ dùmà lá bà !ísígé** (our father people textual-determiner they got up) 'in the days of our forefathers', **Àbámbídé hù sɔ́!dúwá lé** (Abambide your father tree textual-determiner) 'Abambide, this is your tree'. In the procedural grammar this is represented by the special procedure OPENER-D.

The body is composed of one or more paragraphs. There are three kinds of paragraphs: participant oriented, event oriented, and conversational. Each text contains only one, or at the most, two kinds of paragraphs.

When the story is finished, the narrator may choose to close with the formula **tì mùm yé̱** (reason I see) 'that's how I was it', CLOSER-D in the grammar.

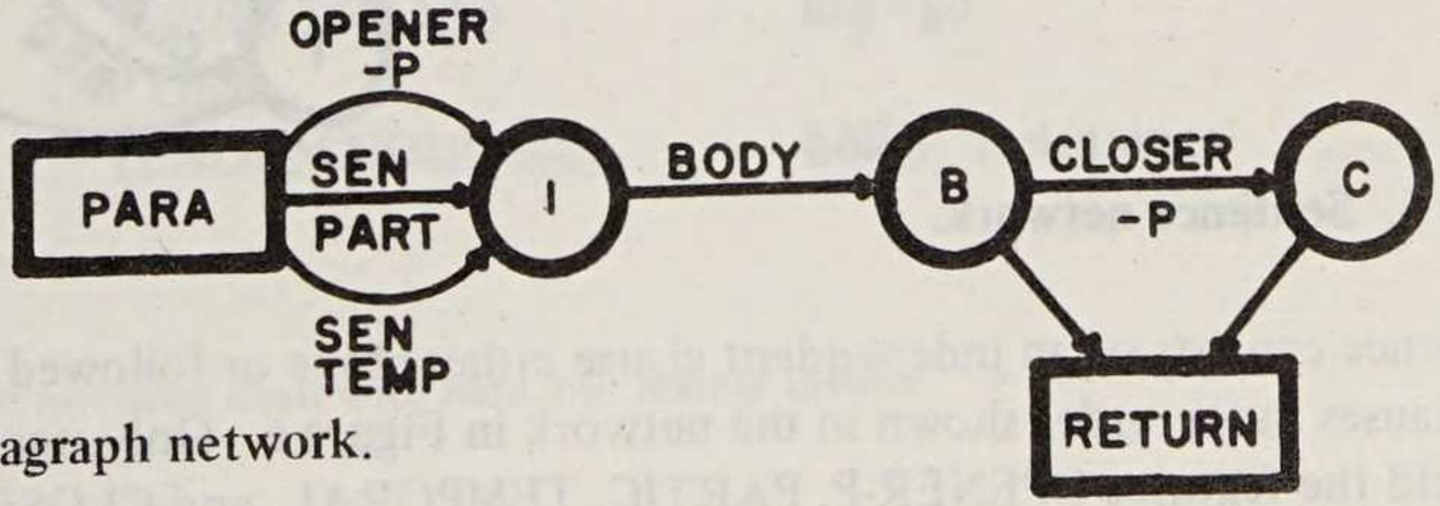


Figure 5. Paragraph network.

[8]The use of determiners at the end of both noun phrases and clauses is parallel to what Hunt (ms) finds for Hanga, a related language.

The paragraph also has three parts. It may begin with an OPENER-P formula which is repeated for each paragraph in the text: **Tɔ̀. Aỳì mɨ̀ yétíá** . . . (O.K. One said . . .) 'O.K. One said . . .'. A paragraph which begins with a formula is part of a text which is reporting conversation.

A paragraph may begin with a PARTIC sentence that introduces a participant as subject: **tà pógá ǹyá ꜜpáɱ́** (topic wife then came) 'this wife then came'. In this case the paragraph is a participant oriented paragraph.

A paragraph may begin with a time topic sentence such as **zám ꜜǹ zám zíꜜmé ꜜbíhá ǹ sɔ̀ yíbígɨ́** (yesterday I yesterday stand look my father younger-sibling-same-sex) 'yesterday I watched my father's younger brother'. The time topic sentence begins paragraphs which are event oriented.

There is a pattern in the use of clause relators within the paragraph. The clause relator **tì** (CRREASON) is frequently in all paragraphs. In participant oriented paragraphs it seems to alternate with sentences beginning with **tà**, the main participant pronoun. Sentences beginning with **ǹyá** (CRSEQ) are less frequent. However, **ǹyá** fits into other kinds of sentences and is most apparent in event oriented paragraphs. Sentences beginning with **gé!** (CRTERM) occur near the end of the paragraph. More study of Gurenne discourse structure is necessary before these constraints are fully understood. The procedure for the paragraph body is therefore not given here.

A text may also have a CLOSER-P closing formula for each paragraph such as: **nànánwá! bálɨ́ ká ɨꜜ bɔ́ɱ́** (now that not determiner it) 'now it's not that way'.

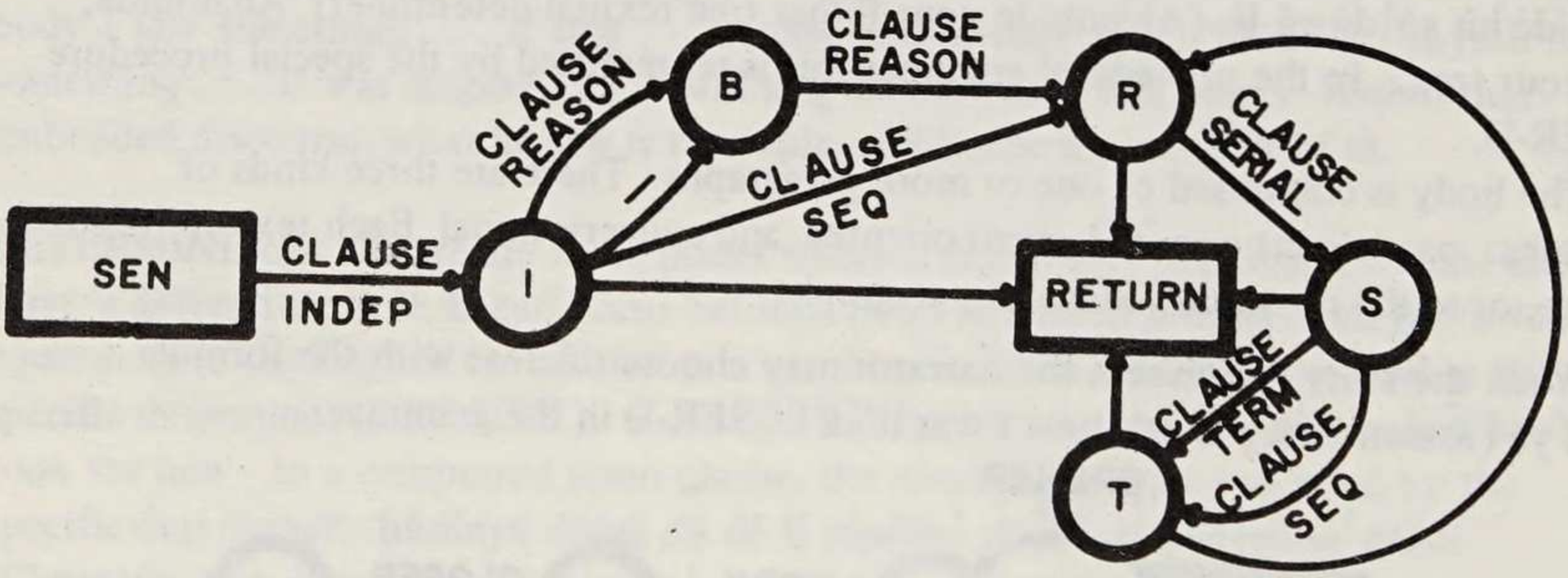


Figure 6. Sentence network.

A sentence consists of an independent clause either alone or followed by up to four dependent clauses in the order shown in the network in Figure 6. Only unmarked sentences yield the features OPENER-P, PARTIC, TEMPORAL, and CLOSER-P. The

OPENER and CLOSER sentences are from a very exclusive set of formulas and include only those given in the examples. The PARTIC sentences have either a noun or a pronoun subject. The TEMPORAL sentences begin with a time noun (TN): **zám** 'yesterday', **dáádí** 'two days removed'.

STRUCTURE TREE

The structure tree is built up as each arc is traversed. Each node of it is represented by a construction name or, for the terminal elements, a word class name. Features are attached to constructions by PUT and TRNSF statements of the grammar. The features of words are taken from the dictionary. The following example gives the analysis that this grammar implies of the first two sentences of the sample text that follows.[9]

(DISCOURSE . . .)	
(PARAGRAPH EVENTOR)	
(SENTENCE TIMETOPIC)	
(CLAUSE INITIAL TIMETOPIC INDEP TRANS)	
(TN REMOTEPAST)	**dáádí** 'day before yesterday'
(NP SUBJ PRON)	
(PRON 1S HEAD)	**ǹ** 'I'
(VP PAST TRANS)	
(TENSE PAST)	**dá** 'past'
(VERB TRANS)	**dóhí** 'follow'
(NP OBJ)	
(DETERMINER)	**lé** 'specific-determiner'
(PRON 1S POSSESSOR)	**ǹ** 'I'
(NOUN NS HEAD)	**kéémɨ́** 'older brother'
(SENTENCE)	
(CLAUSE REASON INDEP TRANS)	
(CRREASON)	**tí** 'therefore'
(NP SUBJ PRON)	
(PRON 1PL HEAD)	**tì** 'we'
(VP TRANS)	
(VERB TRANS)	**kìŋ** 'go'
(NP OBJ)	
(LNOUN NOUN NS)	**bólígɨ́** 'Bolgatanga'

---

[9]Tone changes have not been dealt with here, but mainly involve ' → ´ .

TEXT

Dáádí ǹ dá!dóhí!lé ǹ kéémí. Tì tì kìŋ bólígí. Tì tà dá!pé à lóódí. Tì ǹ dá!pá bólígí lí. Pɔ̀mpì dá pá!gámé. Tì tì dá!ví lɛ́b zúárú nì, sígí!ágɛ́cíltór! mɔ́gódé. Ǹyá dá!tá!péé. Tì ǹ dá!péé!bá!áhí lá. Tì dá ńyá tádí, tí!lɛ́bɛ́!lá!sólmíá yídí ní, wì zé!ílí lóódí lí bì lí, gé!ńyá!kúlí.

(Two-days-removed I past follow specifier my younger-sibling-of-the same-sex. Reason we go Bolga. Reason topic past wash his lorry. Reason I past reach Bolga textual-determiner. Pump past lock. Reason we past return go-back Zuarungu here, go-down Agriculture dam. Then past we wash. Reason I past wash finish textual-determiner. We past then take, reason return specific-determiner white-man house here, make stand lorry textual-determiner there textual-determiner, and then go-home.)

'The day before yesterday I followed my older brother; so we went to Bolga, that we could wash his lorry. So I reached Bolga. The pump was locked. So we returned to Zuarungu, went down to the Agriculture Ministry dam. Then we washed it. So I finished the washing. Then we took it and so returned it to the residency, and left the lorry there, and then went home.'

Figure 7. Gurenne clauses.

```
(START CLAUSE (
((AND   (HAS REQ "INITIAL")
        (TN)
        (PUT CON "INITIAL TIMETOPIC")
        (OR  (AND    (SETPT LAST)
                     (TRNSF "NEARPAST"))
             T)) I NEXT)

((AND   (HAS REQ "TERM")
        (CRTERM)
        (PUT CON "TERM")) I NEXT)

((AND   (HAS REQ "REASON")
        (CRREASON)
        (PUT CON "REASON")) I  I)

I
((AND   (NP (SUBJ))
        (PUT CON "INDEP")) S  S)

S
((AND   (NOT (HAS CON "INDEP"))
        (PUT CON "DEP")) NEXT NEXT)

((AND   (HAS CON "INITIAL")
        (HAS CON "NEARPAST")
        (TN NEARPAST)) T NEXT)

((AND   (HAS REQ "SEQ")
        (CRSEQ)
        (PUT CON "SEQ")) T  T)

T
((AND   (NOT (HAS CON (OR "INITIAL" "TERM" "REASON" "SEQ")))
        (VP)
        (PUT CON "DEP SERIAL")) V NEXT)

((AND   (HAS LAST "SEQ")
        (PUT CON "SEQ")) NEXT NEXT)
```

```
((AND   (MEET FE "INDEP DEP")
        (VP)) V FAIL)

V
(AND    ((SETPT LAST)
        (TRNSF "TRANS INTRANS")) NEXT FAIL)

((AND   (HAS LAST "TRANS")
        (NP (OBJ))) O NEXT RETURN)

((AND   (HAS LAST "TRANS")
        (PUT CON "DELETEDOBJ")) O NEXT RETURN)

((AND   (HAS LAST "INTRANS")) O FAIL RETURN)

O
((AND   (HAS LAST "NP")
        (LOCM)) L  L)

L
((AND   (NOT (HAS LAST (AND "DET" "TEXT")))
        (DET TEXT)) RETURN NEXT)

((ADV)  RETURN RETURN)    ))
```

Figure 8. Gurenne noun phrase.

```
(START NP (

((AND   (DET)
        (PUT CON "DET SPEC")
        (COND (IF (HAS (SETPT CON PV) "NP")
                THEN (PUT CON "COMPOUND")))) A  A)

A
((AND   (NOT (HAS LAST "DET"))
        (PRON (HEAD))
        (PUT CON "PRON")))) Q NEXT)

((AND   (NOUN QUAL)
        (PUT CON "QUAL")) Q  Q)
```

```
Q
((AND   (NOUN  (HEAD))
        (SETPT LAST PV)
        (COND  (IF (HAS PT "HEAD")
                   THEN (AND   (REM PT "HEAD")
                               (PUT PT "POSSESSOR")
                               (REM CON "PRON"))))) N  N RETURN)

N
((AND   (NOUN (HEAD))
        (SETPT LAST)
        (REM PT "HEAD")
        (PUT PT "POSSESSOR")) N  NEXT RETURN)

((AND   (NOR    (HAS CON "QUAL")
                (HAS CON "DET"))
        (BO)) B  B)

B
(AND    (NOT (HAS CON "QUAL"))
        (DET TEXT))) RETURN RETURN)

((AND   (HAS LAST "BO")
        (ADJ)) M NEXT RETURN)

((AND   (NOT (HAS CON "QUAL"))
        (ADJ)) M  M RETURN)

M
((AND   (PRON)
        (SETPT LAST PV (HEAD))
        (MEET FE LAST "1S 2S 3S 1PL 2PL 3PL")
        (PUT CON "REPRISE")) RETURN FAIL)      ))
```

Figure 9. Gurenne verb phrase.

```
(START VP (
((AND   (AUX)
        (PUT CON "AUX")) B NEXT)
```

```
((AND   (NEG)
        (PUT CON "NEG")) X  X)

X
((AND   (HAS CON "NEG")
        (LA)) B NEXT)

((AND   (HAS (SETPT CON UP) "INDEP")
        (TENSE)
        (PUT CON "TENSE")
        (SETPT LAST)
        (TRNSF "PAST FUTURE")) B  B)

((AND   (SETPT CON UP (SEQ))
        (PRON)
        (PUT PT "SWITCHED PRON")) C NEXT)

((AND   (SET PT CON UP)
        (NOT (HAS PT "SEQ"))
        (CRSEQ)
        (PUT PT "SEQ")) C  C)

C
((VERB) V FAIL RETURN)

V
((AND   (SETPT LAST)
        (TRNSF "TRANS INTRANS")) NEXT FAIL)

((AND   (NOR    (HAS CON "AUX")
                (HAS CON "NEG"))
        (ASPECT)) RETURN RETURN)     ))
```

DICTIONARY

à	PRON 3S
ágí cíltór	NOUN QUAL 'Agriculture Ministry'
bá'áhí	ASPECT 'finish'
bì	LOCM 'there'
bó	BO

bólígí	LNOUN NOUN 'Bolgatanga'
dá	TENSE 'past'
dáádí	TN REMOTEPAST 'two days removed'
dòhì	VERB TRANS 'follow'
gé!	CRTERM 'and'
kéémí	NOUN NS 'older sibling same sex'
kìŋ	VERB TRANS 'go'
kùlì	VERB INTRANS 'go home'
lá	LA
lé	DET 'the'
lùbì	VERB TRANS 'go back'
lóódì	NOUN NS 'lorry'
mɔ́gòdè	NOUN NS 'dam'
ǹ	PRON 1S
nì	LOCM 'here'
ǹyá	CRSEQ 'then'
pà	VERB TRANS 'reach'
págàmè	VERB TRANS 'lock'
pé	VERB TRANS 'wash'
pɔ̀mpì	NOUN NS 'pump'
sígí	VERB TRANS 'go down'
sólmíá	NOUN QUAL 'white man'
tà	PRON TOPIC 'main participant'
tàrì	VERB TRANS 'take'
tì	PRON 1PL
vìhì	VERB TRANS 'return'
wì	VERB TRANS 'make'
yídí	NOUN NS 'house'
zé'ílí	VERB TRANS 'stand, keep'
zúárú	LNOUN NOUN NS 'Zuarungu'

PART 2

Transition Network Grammars

CHAPTER 3

TRANSITION NETWORK GRAMMARS: A GUIDE

Joseph E. Grimes

Transition networks are a useful way to give an understandable account of linguistic structures. They show the possible surface structure patterns of a language in a common sense way. At the same time they give an explicit means of relating underlying structures to them. They permit a grammar to be presented through easily visualized diagrams. Also they place the details of restrictions and exceptions that are part of every language in perspective within the grammar. Beyond that, they are in a form that can be matched to actual language data by a computer for verification.

A SIMPLE GRAMMAR

A simple transition network grammar that covers part of English illustrates the basic concepts of **states**, **arcs**, and **recursion**. Other features will be brought in later.

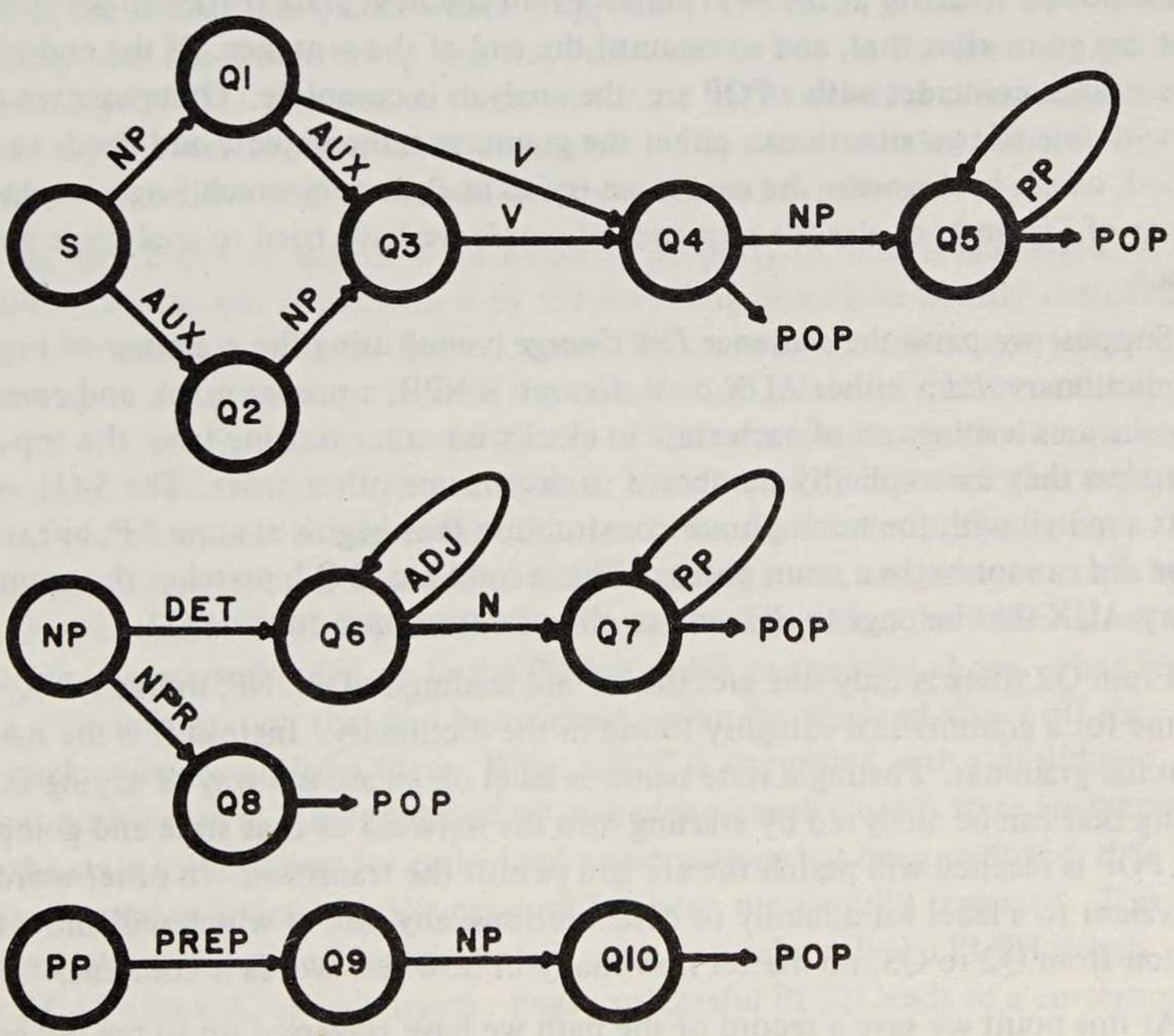


Figure 1. Transition network diagram for a simple grammar of English.

The grammar diagrammed in Figure 1 consists of thirteen **states** and twenty **arcs.** The states are indicated by circles and the arcs by lines. Each state has its own label. Arcs go from one state to another or from a state to the word POP, which will be explained later. Each arc is labeled either with the name of a word category like DET (for 'determiner') or with the label of another state of the grammar like NP, which begins the analysis of a construction.

Along with the grammar goes a dictionary. Each word in the dictionary is labeled as to its grammatical category: FROM is labeled PREP, for example, and THE is labeled DET, using the same category names that are called for in the diagram.

TRANSITIONS

One of the states in the grammar is designated as the starting state. For this grammar it is S. (This does not imply that transition network grammars are necessarily limited to the analysis of sentences.) The grammatical analysis of a sentence of text begins at state S, and proceeds by first trying to determine which of the arcs leaving S can be followed to arrive at the next state. From the next state it tries to see which state it can go to after that, and so on until the end of the sentence. If the end of the input sentence coincides with a POP arc, the analysis is complete. Otherwise we are faced with one of two situations: either the grammar is incomplete and needs to be modified, which is obviously the case if we try to analyze very much English using the grammar of Figure 1, or else the sequence of words we have tried to analyze is not a sentence.

Suppose we parse the sentence *Did George come?* using the grammar of Figure 1. In the dictionary *did* is either AUX or V, *George* is NPR, a proper noun, and *come* is V. We try the arcs leading out of each state in clockwise order starting from the top of the circle unless they are explicitly numbered to show some other order. The S-Q1 arc expects a match with the noun phrase construction that begins at state NP; but it fails because *did* cannot begin a noun phrase. The second arc, S-Q2, matches the grammatical category AUX that belongs to *did*, and so allows a transition to state Q2.

From Q2 there is only one arc, the NP arc leading to Q3. NP, unlike AUX, is not the name for a grammatical category found in the dictionary. Instead it is the name of a state in the grammar. Putting a state name as label on an arc is a way of saying that anything that can be analyzed by starting into the network at that state and going on until a POP is reached will match the arc and permit the transition. In other words, NP is equivalent to a label for a family of constructions, any one of which will allow the transition from Q2 to Q3, no matter how many or how few words it contains.

At this point we save a record of the path we have traversed up to the NP arc: S-Q2 trying Q3 summarizes where we are. We then turn our attention to the NP state. NP-Q6 does not work because *George* is not of the category DET. NP-Q8, however, will work because *George* is listed as NPR. The POP from Q8 tells us that we have finished

one of the constructions that can be a complete NP, so we return to the arc that told us to try to match an NP and make the transition it allows to Q3.

RECURSION

The way in which one part of a transition network like Q2-Q3 can call on another part of the network like NP to see whether a transition is allowed is important for two reasons. First, it means that the same complex construction can be recognized at different places without having to be duplicated. There are NP matches on four different arcs in the network: S-Q1, Q2-Q3, Q4-Q5, and Q9-Q10. A single diagram of the NP subassembly is enough to permit its recognition in any of these places.

Second, consider what would happen if instead of treating NP as a construction by itself, we were to write it into the main network wherever it appears. We could, for example, redo the Q2-Q3 arc as follows: take one arc from Q2 to the NP state below it without performing any match on it, and two others from Q7 to Q3 and from Q8 to Q3, eliminating each POP. (We would also have to add in separate copies of the NP subassembly at the other three places where it appears.) The catch is that if we put the NP subassembly into the S network in this way, then we have to do the same thing for its PP subassembly Q7-Q7. But that contains an NP arc itself, which in turn contains a PP arc, and so on forever. In other words, any attempt to substitute that part of the network leads to an endless chain of substitutions within substitutions.

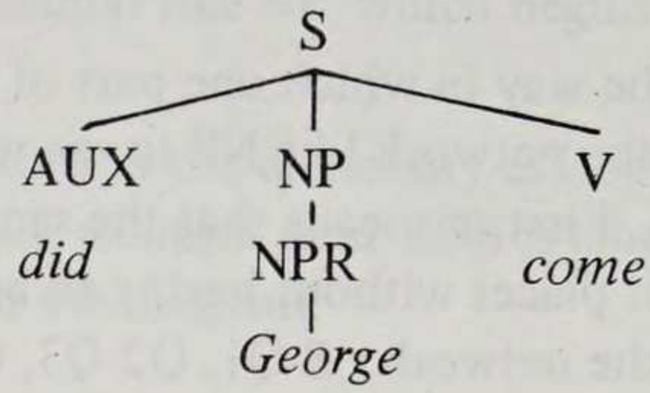
What we have come up against is the **recursive** property of natural languages. A transition network grammar copes with it by the matching procedure already sketched: wherever a construction is to be recognized, rather than a simple word category that can be looked up in the dictionary, it saves the information gained so far about the state of the network on a special list that in the theory of finite automata goes by the name of a **pushdown stack**. Then it starts a new analysis at the state of the network that represents the embedded construction. If that construction in turn calls for the analysis of an embedded construction, its state is added to the pushdown stack, and analysis starts again at the next level of embedding. When a POP arc is reached, it makes the information that was found in the network that led up to the POP available to the level above. Then it takes all the state information that had been found earlier for the level above off the pushdown stack and goes on from there. When a POP is attempted with a pushdown stack that has nothing on it, either because no embedded constructions were analyzed or because all the state information for embedded constructions has been removed, the outermost construction is finished; the network has been successfully traversed. This strategy for matching embedded constituents is quite naturally called a PUSH match rather than a CAT (for CATegory) match. Every successful PUSH leads to a corresponding POP.

Back to the example, following the PUSH to the state NP that recognizes *George* as a kind of NP and POPs back up, the arc leaving Q3 recognizes *come* as belonging to

category V. At Q4 there is no NP to be found by a PUSH; instead, we are at the end of the sentence, so that the POP arc is taken. With nothing on the pushdown stack this means we have successfully gone through the entire sentence. Our record of the states traversed gives a picture like this:

S AUX *did*
Q2 NP
NP NPR *George*
Q8 POP
Q3 V *come*
Q4 POP

or



The relationship between the record of states traversed and a grammatical analysis of the surface form of the sentence is not difficult to see. There are a large number of sentences that could be analyzed by the same network as is given in Figure 1. A record of the states traversed would be close in each case to the analysis a grammarian would give. For example, try each of the following on the network:

George ate his supper.
Has your wife found a place for her roses?
That nice old man in the village sends his greetings.

GRAPH FORM AND LISP FORM

There are two ways to write a transition network grammar. The first is the pictorial or graph representation used in Figure 1. It is easy to follow, but tends to get crowded as the grammar grows, especially after the next section where we add two new kinds of information, tests and actions, to each arc.

The second way of writing a transition network grammar can be done on a typewriter or a computer. It is not limited as to space; everything that needs to be said can be included without crowding. Furthermore, it follows a pattern that allows it to be tested directly by computer. This is a powerful help in developing a grammar from scratch because as the grammar is tested against data, it can be modified on the spot if the analysis it gives is faulty, then immediately retested. The non-pictorial way of writing a transition network grammar is known as the LISP form of the grammar, because it makes the grammar readable by the LISP computer programming language described in the appendix. It is shown in Figure 1a:

```
(S    (PUSH NP  (TO Q1))
      (CAT AUX (TO Q2)))

(Q1   (CAT V (TO Q4))
      (CAT AUX (TO Q3)))
```

```
(Q2   (PUSH NP      (TO Q3)))
(Q3   (CAT V        (TO Q4)))
(Q4   (PUSH NP      (TO Q5))
      (POP))
(Q5   (PUSH PP      (TO Q5))
      (POP))
(NP   (CAT DET      (TO Q6))
      (CAT NPR      (TO Q8)))
(Q6   (CAT ADJ (T   (TO Q6))
      (CAT N        (TO Q7)))
(Q7   (PUSH PP      (TO Q7))
      (POP))
(Q8   (POP))
(PP   (CAT PREP     (TO Q9)))
(Q9   (PUSH NP      (TO Q10)))
(Q10  (POP))
```

Figure 1a. The grammar of Figure 1 in LISP form.

The first thing one notices about the LISP form is that each expression is enclosed in a pair of parentheses. All groupings of terms in the grammar are thus completely explicit. Second, the information on each arc is presented in a concise but easy-to-remember shorthand. Each term of it will be explained as it comes up.

The basic unit of the LISP form of the grammar is the **arc set**. An arc set consists of the name of a state followed by the descriptions of all the arcs that leave that state, given in the order in which they are to be tried. In schematic form an arc set is (STATE (ARC) (ARC) . . . (ARC)).

Each arc is described by giving its component parts. So far each arc has two components: a **match** and a **terminal action**. The **match** can have one of two forms: PUSH STATE, where STATE stands for a state name like NP on the S-Q1 arc or PP on the Q5-Q5 arc, or else CAT CATEGORY, where CATEGORY stands for a word category like DET or V used in the dictionary. The **terminal action** is (TO STATE), in which STATE names the state to which transition is made if the match succeeds. Two more components of arcs, the test and the actions, will be added before the picture of the transition network grammar is complete.

AUGMENTED TRANSITION NETWORKS

Transition networks of the kind exemplified by Figures 1 and 1a, which include the recursive property, are nearly equivalent to context free grammars. (The main technical

difference is that transition networks can also handle unbounded branching.) There are, however, things that happen in natural language that context free grammars manage awkwardly or not at all. One of these is the agreement constraints that all languages have between one part of a sentence and another. For example, in *Has your wife found a place for her roses?* there is a limitation on the noun phrases that will match Q2-Q3 if the auxiliary is *has*: they must be third person and singular (compare that example with *Have your brothers found a place for their cabbages?*). Such ties between one part of a sentence and another make context free grammars, and hence simple recursive transition networks, not quite the right tool for large scale grammatical analysis.

The fact that some constituents of one construction can appear in a different place in a closely related construction is another difficulty for these simple grammars. Sentences like *Which camera did you ask me to bring?* in which *camera* is the object of *bring,* not of *did* or *ask,* overload the capacity of simple grammars like that of Figures 1 and 1a.

Furthermore, although a record of the path we took through the network does show the surface form of what was analyzed, it does not show it in the most useful way. It does not show, for example, that *Did George come?* is a question. It would be useful if its structure could be reported in the way a linguist would report it.

What we need is an **augmented** transition network grammar. A grammar of this kind can handle agreement, order displacement, and a more meaningful presentation of the results of analysis. It seems to be capable of doing everything that is ever required of a grammar.

An augmented transition network grammar is recursive, but has three added features: **registers**, **tests** on arcs, and structure building **actions**. **Registers** are like notes that are kept about items or constructions that match arcs. For example, a register named SUBJ might be used to contain the noun phrase found on arc Q2-Q3 of Figure 1.

Whenever a PUSH match is called for, all the registers that have been set up to that point in the network are saved with their contents on the pushdown stack together with a record of the path that led up to the PUSH. The analysis of the embedded construction then starts with empty registers and fills up its own set as it needs them. When the embedded construction ends in a POP and the original network resumes, the registers that were saved on the pushdown stack are taken off it along with the path information.

Tests on arcs look for agreement. They do this most often by examining the contents of registers that have been set by earlier arcs. They may also be used to find out whether effort would be wasted by trying for a match that could be shown ahead of time to be headed for failure. The match specified for an arc will be tried only if the test on the arc is successful first. If the arc does not pass the test, the match is not even attempted. For example, there is no point in looking for a noun phrase following a verb in English unless that verb is known to be of the kind that can take a noun phrase in that position. To keep things uniform, we stipulate that every arc in the grammar has a test on it. Arcs

that really require no test are given the test result T for "true" to show that the match is to be attempted under all circumstances.

The test is an expression enclosed in parentheses unless it is replaced by the universally true test result T. Test statements have to be able to take values of either T (true) or NIL (false) since those values determine whether the match will or will not be tried. The mechanics of specifying tests in detail will be taken up later. The test follows the matching specification on an arc and precedes the actions.

Actions are taken when a match is found. An arc can have any number of actions associated with it, to be performed if the match works and ignored if it does not. The action most frequently taken upon finding a match is to set or modify the contents of one or more registers. Sometimes the word that is found is stored in a register so that it can be picked up later. Sometimes the word is labeled as it is stored. Sometimes a register is set to an arbitrary value like T for true–indicating that the register is set, but to no particular value taken from the data–or NIL for false, which has the effect of removing the entire register from the list of registers. Registers set to such arbitrary values are called **flag** registers or flags.

Putting information into registers allows it to be reorganized later as the analysis proceeds. A transition network grammar can set up a tentative analysis, then change it as new evidence comes in.

Where a constituent is found out of its regular place in a construction, a special HOLD action puts it on a holding list from which it must later be removed and placed in its right structural position before a POP will work.

POP arcs normally have a fairly complex set of structure building actions associated with them. A POP arc collects all the bits and pieces of information about a construction that were put into various registers since that part of the network was activated, and places them into a representation of the construction that has been found. In building this representation it can change the order of elements and can add or subtract information. The structure that is built up by the POP is passed on to the arc that initiated the PUSH, which from there on handles it as a unit.

Each specific action in the list of actions is enclosed in a pair of parentheses. Actions follow tests on an arc, and are followed by a single terminal action. There is no limit on the number of actions an arc can take.

The most common action is (SETR REGISTER '*'), where REGISTER is the name of some register and '*' stands for the **current form**. The current form, represented here by an '*' for typographic reasons and by an asterisk in some other transition network grammars, is either the word that was matched by a CAT match, or else the entire structural description that was built up at a lower level by a POP arc. SETR takes the word or construction indicated and puts it into the register named. Instead of the '*', the information to be set in the register by SETR can be given on the spot in the form (QUOTE XYZ), which puts the character string XYZ into the register. For example,

(SETR TYPE (QUOTE DCL)) sets the value of a register called TYPE to the letters DCL. There is no restriction on how many registers may be set in going through a network, nor is there any predefined list of names appropriate for registers. They are created as needed by the linguist and labeled with one-word labels that serve as reminders of the kind of information they are being used for.

Another action is used most often with POP arcs, though it may be used with other kinds as well. (In a POP arc, however, actions are used to construct the form in the specification POP FORM, rather than appearing in the usual position after the test. The form given in the POP arc is the value that is returned to the higher level.) The name of the action is BUILDQ; it has the form (BUILDQ SKELETON REGISTERS). The skeleton part, enclosed in parentheses, is an empty structure that contains plus signs to indicate the points where information is to be filled in. Following the skeleton come the names of registers where the information is kept. The first register's contents replace the first plus sign, the contents of the second replace the second plus sign, and so forth.

Terminal actions are of two types. Normally the form is (TO STATE), giving the name of the state to which a transition is to be made after the action is performed if the match is successful. TO terminal actions cause the following state to examine the next word in the input sequence. There is also a (JUMP STATE) terminal action that makes the next state examine the current word again instead of moving ahead to the next word.

TRANSITION NETWORKS AND THEORIES OF LANGUAGE

Because the structure building actions can reshuffle information and label it in any way that is desired, it is possible to use a transition network grammar to simulate the implications of any linguistic theory. In itself it is independent of theories that are used to explain linguistic structures. Its results can be made to come out in the form specified by, say, a generative transformational grammar, a semantically based grammar, a tagmemic analysis, a dependency grammar, or even in forms implied by theoretical systems that are yet to be devised.

The reasons for this independence from particular theories is straightforward enough. The matching and conditional portions of a transition network grammar are a way of reporting what is and is not said in a language. The network reports the elements that are found and the arrangements in which they are found. This ticking off of possible sequences, even though it is accomplished recursively, does not constitute an explanation of them. The explanation is what a theory of language tries to provide. The model that expresses the relationships which the theory recognizes is what determines the way in which the structure building actions put together the pieces. As the matches and conditions are finding the surface form of a stretch of text on the one hand, the structure building actions are putting together an account of what lies behind that surface form on the other. The split between recognition and reporting therefore makes transition network grammars an important tool for testing the consequences of theoretical developments.

It is mainly the action parts of the network that need to be modified to reflect the development, while the recognition part stands relatively untouched.

Lexical categories entered in the dictionary:

AUX	auxiliary verb
V	verb
DET	determiner
ADJ	adjective
N	noun
NPR	proper noun
PREP	preposition

```
(S    (PUSH NP T   (SETR SUBJ '')  (SETR TYPE  (QUOTE DECL))    (TO Q1))
      (CAT AUX T   (SETR AUX '')   (SETR TYPE  (QUOTE Q))       (TO Q2)))

(Q1   (CAT V T     (SETR AUX NIL)  (SETR V '')   (TO Q4))
      (CAT AUX T   (SETR AUX QQ)   (TO Q3)))

(Q2   (PUSH NP T   (SETR SUBJ '')  (TO Q3)))

(Q3   (CAT V T     (SETR V '')     (TO Q4)))

(Q4   (PUSH NP T   (SETR VP (BUILDQ (VP (V +) '') V))  (TO Q5))
      (POP  (BUILDQ  (S + + + (VP +)) TYPE SUBJ AUX V) T))

(Q5   (PUSH PP T   (SETR VP  (APPEND  (GETR VP)  (LIST '')))  (TO Q5))
      (POP  (BUILDQ  (S + + + +)  TYPE SUBJ AUX VP)  T))

(NP   (CAT DET T   (SETR DET '')   (TO Q6))
      (CAT NPR T   (SETR N '')     (TO Q8)))

(Q6   (CAT ADJ T   (SETR ADJ  (APPEND  (GETR ADJ)  LIST '')))  (TO Q6))
      (CAT N T     (SETR N '')  (TO Q7)))

(Q7   (PUSH PP T   (SETR PP  (APPEND  (GETR PP)  (LIST '')))  (TO Q7))
      (POP  BUILDQ  (NP + +  (N +)  +)  DET ADJ N PP)  T))

(Q8   (POP  BUILDQ  (NP  (NPR +))  N)  T))

(PP   (CAT PREP T (SETR PREP '')  (TO Q9)))

(Q9   (PUSH NP T   (SETR OBJ '')  (TO Q10)))

(Q10 (POP  (BUILDQ  (PP + +)  PREP OBJ)  T))
```

Figure 2. Transition network grammar of Figure 1 in LISP form.

A FULLY SPECIFIED EXAMPLE

Figure 2 adds tests and actions to the grammar of Figure 1a. Each arc now has the fully specified form (MATCH (TEST) (ACTION) (ACTION) . . . (TERMINAL ACTION)). All tests in Figure 2 are of the form T (true), indicating that the match is to be tried unconditionally. Going back to Figure 1a, the fully specified arc set for its first state S is as follows:

```
(S (PUSH NP T   (SETR SUBJ '') (SETR TYPE  (QUOTE DECL))   (TO Q1))
   (CAT AUX T  (SETR AUX '')  (SETR TYPE  (QUOTE Q))       (TO Q2)))
```

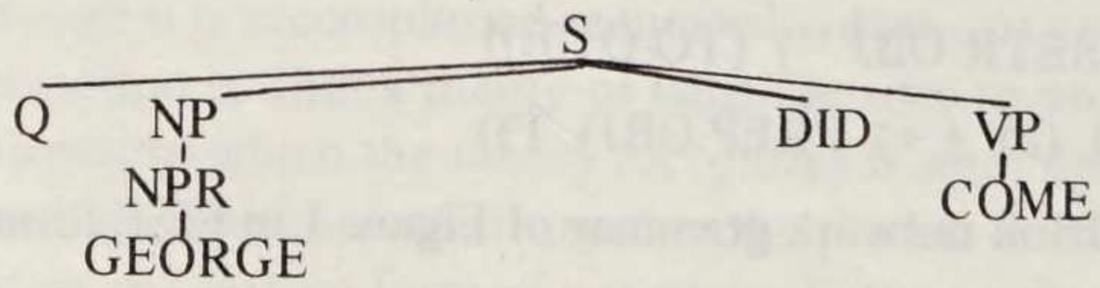
This says that the state S has two arcs leading out of it. The first tries to match a noun phrase by pushing to the NP state. It tries this without restriction; the test is always T. If the arc matches a noun phrase, the '' tells it to put the entire phrase into a register called SUBJ. The same arc also puts DECL (for 'declarative') into a register called TYPE, then makes the transition to state Q1. If the sentence does not begin with a noun phrase, the grammar looks to see if it can match any word tagged in the dictionary as belonging to the category AUX, again with no specific test. If it finds an auxiliary, it puts that word in a register called AUX, then sets the TYPE register to Q (for 'question') and goes to state Q2.

Figure 2 gives in its full LISP form the grammar that was presented in Figure 1 in pictorial or graph form and in Figure 1a in a LISP form that omitted tests and actions. The only terms in Figure 2 that are not explained yet are APPEND, GETR, and LIST, which will be taken up later.

The structure this grammar assigns to *Did George come?* is put together by the POP arcs from various registers. It comes out as

```
(S Q
     (NP  (NPR GEORGE))
     DID
     (VP COME))
```

The BUILDQ actions are written so that the representation they provide is the parenthesized equivalent of a tree in which the label of the parent node is given first inside each parenthesis pair, followed by the nodes that it dominates. The equivalent tree representation is



The analysis that results from parsing *That nice old man in the village sends his greetings* by means of the grammar of Figure 2 is

```
(S DECL
     (NP THAT
          (NICE OLD)
          (N MAN)
          (PP IN
               (NP THE
                    (N VILLAGE))))
     (VP  (V SENDS)
          (NP HIS
               (N GREETINGS))))
```

This overview of transition network grammars gives an idea of how they work and how they can be constructed. The next section goes over the same ground in detail, giving additional types of matches, tests, and actions, with examples taken from a large scale grammar of English.

LARGE SCALE GRAMMARS

Augmented transition networks for natural language analysis were developed by William A. Woods, who published a report with that title in 1969 (Aiken Computation Laboratory Report CS-1, Harvard University, available from the National Technical Information Service as Microfiche PB-203 527). Woods also developed the computer system that applies a transition network grammar to data in order to perform an automatic parsing or grammatical analysis.

The automatic parser was developed in connection with an information for lunar geology. By feeding the analysis performed by the transition network grammar into a semantic analyzer, it is possible to have a computer find or calculate answers to questions about moon rocks that are asked in ordinary English rather than in an artificially constructed computer language. The grammar of English that was developed for this purpose does not cover everything English speakers can say, but it does cover a surprisingly large proportion of the language in a small compass: 58 states and 172 arcs. Examples given in the next section to show the full range of matches, tests, and actions are taken from that grammar (Woods and Kaplan, 1971).

ARC TYPES

Each arc in a transition network grammar is identified by the label for the state it comes from and for the state it leads to. In a large grammar arbitrary labels like Q3 are fine for a computer but confusing to a grammar writer or reader. For this reason Woods adopts a two-component naming scheme for states. It helps to communicate to people the place where each arc belongs in the total scheme.

The first part of the label designates the construction to which the state belongs: S for sentence, NP for noun phrase, PP for prepositional phrase, and so on. The second part tells how far along in the network the analysis has progressed by the time that state is reached. For example, S/VP is a state in the sentence analysis that is reached after the end of the verb phrase (VP) has been found; NP/HEAD is a state in the analysis of noun phrases that is reached after the head of the phrase has been identified; PP/PREP is the state reached in prepositional phrase analysis after the preposition itself is found. This naming policy makes it easier to sort out and remember the different parts of a network.

The two terminal actions, as already mentioned, are (TO STATE) and (JUMP STATE), where STATE is the name of the state to which transition is made. TO is the normal end of an arc. It is used when a match has been found and the arcs leaving the next state are to operate on the next word after the one just matched. The initial state in the grammar, for example, has as one of the arcs leaving it

```
(S/ . . .
    (CAT ADV T
        (ADDL VMODS (BUILDQ (ADV '')))
        (TO S/) . . . )
```

This means that if an adverb is found at the beginning of a sentence, it is put into a register, then the analysis of the sentence is begun over agian with the next word, also at state S/.

JUMP, on the other hand, makes the following state look at the same word over again. It is used mainly in arcs that do not match anything in the text string but perform some action. If an 'is there?' question sentence has been tentatively identified, the grammar takes a DO arc: (S/THERE (DO (. . .actions. . .) (JUMP VP/V))), which performs the actions that are required but keeps the same input word (the one after *there*) to be looked at by the arcs that leave VP/V. A TO terminal action here would cause the word after *there* to be skipped.

The arcs that go out of a state are looked at in order. In some cases the order is determined by efficiency, in that the arc looked at first is the most likely, or the one looked at last has the most complicated condition to test. In other cases order may be used to forestall ambiguity by granting precedence to one of several alternate parsings. All the arcs in an arc set are disjoint; that is, only one of them can be followed. The first one that fits is normally the one followed, though the computer system for analyzing transition networks keeps track of the remaining arcs at each state in case some state is reached where no transition is possible, so that the analyzer can back up and try again on a new path.

Two kinds of matching were introduced earlier: CAT by word category and PUSH by recursion. In addition there are two other kinds of word match. WRD TO matches the specific word TO and no other. Either WRD (AND BUT HOWEVER) or MEM (AND BUT HOWEVER) matches any one of the three words named.

VIR STATE examines a special list called the HOLD list to see whether an element analyzed previously by a push to STATE but recognized as being out of its deep structure position when it was found can be moved into the arc at that point. (VP/AGT (VIR NP T (. . . actions . . .) (JUMP VP/VP))) will pick up the *who* in a sentence like *Who was it requested by?* from the HOLD list, where it was placed when *it* turned out to be the real subject of the sentence. The actions on the arc attach the noun phrase taken from the HOLD list to the analyzed structure in a way that is determined by its relation to *by*, not by its initial position in the sentence. This type of arc is called a **virtual** arc since, instead of matching something at that place in the text, it matches something that was found earlier as though it had come up at that point.

A comment on word matching: some words appear in more than one form. Morphological analysis must frequently be performed to identify the root of a word together with the inflectional categories that are represented in its text form. Woods has a morphological analyzer for English that makes use of morphological codes stored in the dictionary to guide in the decomposition of regularly formed words. Each form of a word whose main entry is tagged as morphologically irregular appears as an entry by itself. For languages like English that have a light inflectional apparatus this seems adequate; but for languages that have complex morphological structures there probably needs to be a different approach to finding out what is in each word. A shortcut solution for most languages is for the linguist to segment each word into its component morphemes by means of hyphens, then have appropriate networks to analyze word structures.

To return to arc types, three kinds of arcs involve no matching. POP arcs, which specify a form to be passed on to the level above subject to a test, have already been mentioned. There is also a JUMP arc, not to be confused with JUMP as a terminal state. The JUMP arc names a state to which transition is to be made without advancing to the next input word, a test, and a list of actions. In its unconditional form it is useful for spreading out part of a network that is getting crowded, in which case it has no actions and is nothing but a dummy arc. It is also used with special tests that examine a word but do nothing to it themselves, as for example

```
(S/ (JUMP S/Q (QSTART)
          (SETRQ TYPE  Q)) . . . )
```

in which the test QSTART recognizes that the current word is either a question word or an auxiliary verb, but leaves it to state S/Q to decide what kind it is and therefore what to do with it. The third arc type that involves no match is the DO arc, which allows actions to be carried out unconditionally before transfer is made (usually by a JUMP terminal action) to the next state. Just as the DO arc allows actions, a TST arc allows a specialized test to be performed, which may then lead to a transition or specific actions without affecting the input text.

Tests, as mentioned earlier, are LISP expressions that can take values of either true (T) or false (NIL). Tests other than the universally true T are built up of three components:

sources of information to be tested, **test conditions** to be applied to that information, and in the case of complex conditions, a **logic** for combining several tests into one.

The **source** of information to be tested is normally either a register, the input text, or the dictionary. (GETR REGISTER) makes the contents of the register that is named available. It can also constitute a test condition by itself; that is, if no further test condition is applied to the contents of the register, GETR is true if the register has anything at all in it and false if it is empty or nonexistent. For example, (VP/V . . . (JUMP VP/HEAD (GETR SUBJ) . . .)) moves to a new state only if a subject for the sentence has been found earlier and placed in SUBJ. With a second argument, (GETR REGISTER WHERE) allows the contents of a register on a higher level to be examined. WHERE can have the value T for the topmost level, NEAREST for the first occurrence of the register named on a higher level, or a number for a particular stack level.

The test (NULLR REGISTER) is the opposite of GETR used as a test. (S/AUX (CAT NEG (NULLR NEG) . . .)) will look to see if the current word is a negative only if no other negative has been placed in the NEG register already. (NULL EXP) tests any LISP expression EXP to see if it is false; here EXP represents an expression and is not the name of a register.

The word currently being looked at in the input text is made available in a test by the special sign '' (or asterisk in some versions). In tests '' always refers to the current input word, even though in actions it may refer to the complete construction found by a PUSH match. (NEXTWRD) by itself makes available the word after the current one. With an argument NEXTWRD acts as a test condition: (NEXTWRD (WITH BY)) is true if the word after the current word is either *with* or *by* and false otherwise. These information sources will be illustrated later in conjunction with specific test conditions.

(GETF FEATURE) gets the value of a feature from the dictionary. It is used only with CAT matches because it always refers to the current input word. It looks to see what value, if any, the current word has for the feature named in its dictionary entry. (S/IMP (CAT V (GETF UNTENSED) . . .)) will try to match a word of the verb category only if the current word has a value entered (most likely the value T for 'true') for the feature UNTENSED–that is, if the word is the uninflected or infinitive form of a verb, like *have* and *take*. If the form is inflected, like *has* or *took*, it will not even try for a match.

One other source of information is the test specification itself. A particular word can be included in a test condition by QUOTE, in a way we shall see when we discuss specific tests.

The **test conditions** themselves examine information from one or more sources to see whether it fits certain requirements. The test EQ is followed by two things, and is true if they are the same and false otherwise. (EQ '' (GETR CONJ)), for example, is true if the current word '' is the same as the word that is stored in a register named CONJ. (EQ (QUOTE TO) (NEXTWRD)) is true if the word after the current word is the word TO.

There is a short form of the last test condition that eliminates the QUOTE: it has the form (WRD TO (NEXTWRD)). It is used as part of a complex test for deciding whether to look for a complement construction that begins with a negative like Hamlet's *not to be.* WRD here is used as a test, not as a matching instruction. It can also look for one of several words, as in (WRD (FOR TO THAT) '!), where the first argument is a word or a list of words to be tried and the second argument is the word to be tested against the first.

In the same way CAT can be used to test whether the current word belongs to a particular category or a list of category names, not as a matching instruction. (S/ . . . (PUSH PP (CAT PREP) . . .)) will try to match a prepositional phrase only if the current word is a preposition.

(CHECKF CATEGORY FEATURE) is similar to GETF, except that it looks up the current word to see if it has the feature that is requested under the category that is named, while GETF gets the actual value of the feature named for the current input word. (S/ . . . (JUMP S/IMP (. . . (CHECKF V UNTENSED) . . .))) has as part of the test for a JUMP the requirement that the current word be listed in the dictionary as a V (verb) that has the feature UNTENSED as part of its dictionary information.

Other tests are put together from the more elementary ones and referred to in a packaged form that makes it easier to remember them. (QSTART), for example, is true if the current word is an auxiliary verb or one of the interrogative words that can begin a question: *who, what, where,* and so forth. (MODAL) is true if the current word is a modal like *can, ought,* or *might.* (PNCHECK PHRASE PNCODE) tests the noun phrase that is named to see if it is of the person-number category required by PNCODE. (RFEAT (FEATURE WORD)) is true if the word indicated has the feature named in its dictionary entry. If the word is not otherwise indicated, it is the current word. (S/ . . . (CAT ADV (RFEAT NEGADV) . . .)) is tried if the current word is a negative adverb, which may entail an inverted word order as in *barely had we arrived when . . .*

(SCOMP REGISTER) is true if the register named contains a verb that is capable of taking a sentence complement, like *want (to go)* or *feel (that it is time).* (VP/HEAD . . . (PUSH COMPL/NTYPE (SCOMP V) . . .)) looks for such a verb before trying to match a complement construction. (VPARTICLE (REGISTER PARTICLE)) is true if the word in the register named is a verb that can take the verb particle specified, like *look up* or *carry on.* (VPARTICLE REGISTER) is true if the register contains a verb that can take any verb particle at all. (VPASSIVE REGISTER) is true if the register contains a verb that can be passivized like *eat, be eaten* and false if it cannot be passivized like *cost* or *sleep.* (VTRANS REGISTER) is true if the register contains a verb that can take an object, and (INTRANS REGISTER) is true if it cannot.

```
(VP/HEAD . . .
        (PUSH S/Q (AND    (VTRANS V)
                          (WRD (WHICH WHO WHAT WHOSE)))
                (SENDRQ TYPE IDQ)
                (SETR OBJ (BUILDQ (NP '')))
                (TO VP/NP)) . . . )
```

looks for transitive verbs whose objects are indirect questions, as in *I know whose handwriting this is.*

The **logic** components of a test combine simple tests into complex ones. (NOT (TEST)) is true if the test is false and false if the test is true, as in (S/ . . . (JUMP S/DCL (NOT (QSTART)) . . .)), where the jump will be taken if the current word cannot begin a question. (OR (TEST 1) (TEST 2) . . . (TEST N)) is true if any test in its scope is true, while (AND (TEST 1) (TEST 2) . . . (TEST N)) is true only if all the tests within its scope are true. The complement arc has a complex test that combines both:

```
(S/DCL . . .
        (PUSH COMPL/  (OR   (WRD   (FOR TO THAT))
                            (AND   (WRD TO (NEXTWRD))
                                   (CAT NEG)))
        (SETR SUBJ '')
        (TO S/NP)) . . . )
```

This test will try to match a complement structure on one of two conditions: either the current word is one of the set *for, to,* or *that* which begin most complements in English, or the current word is a negative and the word that follows it is *to,* as in *not to speak would be a mistake.* NOR is a combination of NOT and OR, and is true only if all the conditions within its scope are false.

```
(NP/ . . .
        (JUMP NP/ART  (NOR    (CAT    (DET PRO NEG))
                              (WRD    (WHOSE WHO WHAT
                                       WHETHER IF)))
        ) . . . )
```

recognizes noun phrases that do not begin with certain categories or words.

In constructing a transition network grammar, linguists who do not have access to a computer for testing it may prefer to simply write out the test on each arc in prose. On the other hand, the form that the computer accepts is concise, testable without a computer, and completely explicit. It is less susceptible to concealing hidden assumptions than a prose expression that attempts to cover the same ground.

ACTIONS

The basic structure building action already mentioned is abbreviated SETR. It sets a register to some value. The value is a form which may be the same as some of the forms involved in tests. For example, '' has as its value the current word in a CAT arc, as in

```
(S/; (CAT CONJ (OR   (NULLR CONJ)
                     (EQ '' (GETR CONJ)))
        (SETR CONJ '')
        (TO S/;)))
```

Here the grammar specifies an attempt to match a conjunction in a coordinate sentence either if no conjunction has been put in the CONJ register yet as in *we came; we looked; and we left immediately;* or, if one has, it is identical with the current word as in *we came; and we looked; and we left immediately.* In either case the conjunction that is found (') is put into the CONJ register. As mentioned, however, if ' is part of the action on a PUSH arc, it stands for the entire construction that was matched by the PUSH, not just for a single word.

```
(S/DCL . . .
      (PUSH NP/ T
            (SETR SUBJ ')
            (TO S/NP)) . . . )
```

puts an entire noun phrase into the SUBJ register, no matter how complex it is.

A variant form of the command, SETRQ, saves the trouble of having to QUOTE a form to be put into a register.

```
(S/ . . .
      (JUMP S/Q (QSTART)
            (SETRQ TYPE Q)) . . . )
```

puts the letter Q into the register called TYPE if the sentence begins with a question word or auxiliary verb.

SENDR sets a register on the next level down rather than on the current level. This is done in preparation for a PUSH so that the information sent will be available to the network at the lower level. A PUSH arc automatically checks to see if there are any SENDR actions at the beginning of its own action list (together with any actions other than SENDR that begin with (! . . .), which are also done before the actual PUSH) and performs them before doing the PUSH, thereby making sure that the network at the next level of embedding has the information on it needs to work with. For example,

```
(VP/HEAD . . .
      (PUSH COMPL/NTYPE  (SCOMP V)
            (SENDR NTYPE  (QUOTE THAT))
            (SETR COMPL ')
            (TO VP/NP)) . . . )
```

tests to see if the main verb of the sentence can take a complement construction. If it can, it sends the word THAT into an NTYPE register which the complement network uses. If the appropriate construction is found, as it would be in *I think that someone is coming,* the entire complement *that someone is coming* is placed in a register called COMPL. There is an abbreviated form of the command that eliminates the QUOTE: SENDRQ as in (SENDRQ NTYPE TYAT).

LIFTR and LIFTRQ set a register on a higher level of embedding. They are used in reporting back the structure found by a PUSH arc, as in

```
(NP/HEAD . . .
    (JUMP NP/NP T
        (LIFTR NPFEATURES (RESUMETAG NP/HEAD))) . . . )
```

which passes certain flag information back up to a register in the upper level network called NPFEATURES. (LIFTR REGISTER FORM WHERE) can set a register higher up on the stack, either at the topmost level T or at a particular level indicated by a number.

(ADDR REGISTER FORM) puts the form specified at the right end of whatever may already be in a register, and (ADDL REGISTER FORM) puts it at the beginning or left end. ADDR and ADDL do not erase anything that is already stored in the register, as does SETR. Note

```
(S/ . . .
    (PUSH PP/ (CAT PREP)
        (ADDL VMODS '')
        (TO S/)) . . . )
```

which looks for a prepositional phrase at the beginning of a sentence. If it finds one, it adds it to a register called VMODS. The last phrase found appears first in VMODS.

(HOLD FORM) places the form named on a special list called the HOLD list, from which it is eventually taken by a VIR virtual arc. This is the mechanism for handling elements of a construction that are uttered ahead of the place in the structure where they normally fit, which usually happens for thematic reasons. The features that belong to the element also go on the HOLD list if specified by the full expression (HOLD FORM FEATURES). In analyzing a relative clause, for example, one arc puts the relative word *whom* on the HOLD list immediately, since it can never be the subject of the sentence embedded in the relative construction:

```
(R/ . . .
    (WRD WHOM T
        (HOLD (GETR WH))
        (SETR WH NIL)
        (TO R/WH)) . . . )
```

For constructions like *it pleases me that you thought so*, where part of a noun phrase *it* comes in normal syntactic order and the rest follows later in extraposed position, the HOLD mechanism is not the appropriate one. Instead a flag is set on constructions that might be only partially analyzed when a POP is reached, so that if the second part does come along later it will be associated with the first part.

```
(NP/HEAD . . .
    (JUMP NP/NP T
        (LIFTR NPFEATURES
            (RESUMETAG NP/HEAD))) . . . )
```

uses RESUMETAG to set a register in the construction that called the NP network. Later

a RESUME action continues the analysis of that network at the node NP/HEAD, where it left off. The action appears, for example, in

```
(VP/HEAD . . .
        (VIR NP (VTRANS V)
                (RESUME)
                (SETR OBJ '')
                (TO VP/NP)) . . . )
```

There the virtual arc VIR NP looks at the HOLD list to see whether a noun phrase might have been found earlier that could serve as object of the sentence. If it finds one, as it would find the *what* of *what did you see?*, it does not stop there. By means of the RESUME action it checks ahead to see if a second fragment of the noun phrase might follow from that state, as in *what did you see that might give us our bearings?* The flag set earlier when *what* was found directs the RESUME action to plunge back into the noun phrase analysis at NP/HEAD where it left off, thereby uniting both the fronted element *what* and the rest into a single object *what that might give us our bearings*, set into the OBJ register.

On some arcs more than one action is possible, depending on specific details of what was found during the match. These alternative actions are expressed by the LISP conditional expression, which has the form (COND (EL AL) (E2 A2) . . . (EN AN)). The E's and A's represent pairs of expressions and actions. Each expression, like a test on an arc, can be either true or false. Each pair is tried in turn. If the first expression is true, that action is taken and the whole condition is satisfied so that no other pairs are looked at. If the first expression is false, the second is tried, and so on. If all the conditions are false, no action is taken. A conditional action is found at the beginning of the noun phrase network:

```
(NP/  (CAT DET T
           (COND  ((GETF POSSPRO)
                              (ADDL ADJS (BUILDQ (POSS (NP (PRO '')))))
                              (SETRQ DET THE))
                  (T  (SETR DET '')))
      (TO NP/ART)) . . . )
```

This says that if a determiner is found, it is first of all examined to see if it has the feature POSSPRO as in *my* or *his*. If it is a possessive pronoun, it is added to a register called ADJS which eventually may contain other adjectival modifiers, and the DET register that contains the determiner is set to THE to show that it is definite. For any other kind of determiner (T denoting all other possible conditions), that determiner itself is placed in the DET register. This would be the action taken for *a, the, those,* or *some.*

When a POP is reached, and sometimes at other points, pieces of information that have been stored temporarily in various registers are brought together into a description

of a structure. The action that collects these pieces and puts them together is (BUILDQ SKELETON SOURCES). In its simplest form BUILDQ has plus signs in its skeleton that are filled in from the contents of registers named after the skeleton in the order in which they are to be drawn on. For example,

(PP/NP (POP (BUILDQ (PP (PREP +) +) PREP NP) T))

will take a preposition in the PREP register and a noun phrase in the NP register and combine them at the points indicated by the plus signs. The preposition *to* and the noun phrase *the store* would be handed back to the arc that called the prepositional phrase section of the network in the form (PP (PREP TO) (NP THE (N STORE))), indicating its internal structure.

BUILDQ can also draw on other sources of information. In the same way that + indicates where the contents of a register are to be put into the skeleton, ' ' shows where the current word is to be put. $ (or double cross in some versions) indicates an expression which is to be evaluated and the results put in place of the dollar sign. The *at* symbol in @ X1 X2 . . .) says that the different lists X1, X2, and so on are to be appended to each other to form a single list. For example, the structure for conjoined noun phrases is constructed by the following statement:

```
(NP/NP . . .
      (POP (COND     ( . . . )
                     (T (BUILDQ ( @  (NP $) +)
                          (COND ((GETR CONJ))
                                (T (QUOTE OR)))
                          BODY))) . . . )
```

The $ is replaced by one of two things: the contents of the CONJ register if any, or if nothing is in CONJ, by the word *or*. Then the register BODY, where a list of the component noun phrases in the conjunction has been stored, replaces the plus sign. The @ specifies that the list of lists in BODY is to be appended to the list (NP $) to form a single list. Thus an analysis of *long days and short nights* would put *and* in the CONJ register and ((*long days*) (*short nights*)) in the BODY register. Placing them into the skeleton as directed by $ and +, then combining them into a single list as shown by , results in a form (NP AND (LONG DAYS) (SHORT NIGHTS)) that is returned by the POP arc. (Note: in LISP notation a **list** is represented by zero or more elements enclosed in parentheses. The elements of a list are either other lists or **atoms,** which are single words separated by spaces.)

There is a variant action BUILD in which the skeleton, instead of being quoted directly as in BUILDQ, comes from the evaluation of some other expression. This might be used if there were several possible skeletons, the choice of which involved a COND conditional expression.

There are also three packaged structure building complexes used in the Woods grammar: SBUILD for complete sentences, NPBUILD for noun phrases, and DETBUILD

for determiner structures. They are basically BUILDQ actions; but since they are complex and may be called by more than one arc it is easier to handle them as units rather than to rewrite them each time they appear. They are specific to English.

BUILDQ can be simulated by more elementary actions. (QUOTE XYZ) has already been mentioned; its value is the character string XYZ. (LIST A B C) takes elements A, B, and C and combines them into a single list. (APPEND L M) takes a list L and a list M and combines them into a single list.

Miscellaneous actions which do not fit into any other category are given here. (ABORT) causes any arc it appears on to fail even though the test on it may have been met, the match found, and some actions performed. This forces the automatic parser to back up and try another arc, presumably one that will succeed on the basis of information tested on the way to the ABORT. For example, following a verb with no subject this situation applies:

```
(S/NO-SUBJ . . .
    (PUSH NP/ T
        (' (SETR HOLDNPFEATURES    (GETR NPFEATURES)))
        (COND ((NULL (PNCHECK ''    (GETR PNCODE)))
                (ABORT))) . . . ))
```

That is, before a match for a noun phrase is tried, as indicated by the ', a copy of the register NPFEATURES is made. Then if the match succeeds, a register called PNCODE that is set to the person and number of the noun phrase is tested against the phrase, and if there is a discrepancy as to person-number agreement, the entire arc is abandoned.

(SUSPEND W) suspends one computation in favor of another. It adds the weight W to the arc it is part of and puts that arc on the list of alternatives to be tried later if the next arcs fail. It is used in noun phrase analysis:

```
(NP/HEAD . . .
    (TST R/NIL (AND (GETR PPFLAG)
                    (NOR    (WRD (WHAT WHO WHOM
                                  WHICH THAT WHOSE))
                            (GETR QDET)
                            (WRD BE (GETROOT '' V))))
    (SUSPEND 1)
    (SENDRQ TYPE REL)
    (SENDR WH (BUILDQ (NP (DET WHR) + (NU +))
                      N NU))
    (PUSH R/NIL)
    (ADDL NMODS '')
    (TO NP/R)) . . . )
```

This rather complex arc looks for reduced relative clauses that follow a postnominal prepositional phrase. The actions listed after the SUSPEND (which here includes PUSH

as an action) will be performed only after a later arc in the same arc set has been tried and has failed. The resulting backup will then get past the SUSPEND. Its result will be, for example, to assign the reduced relative clause in *the man near the girl I see* to *girl* rather than to *man,* where it would be assigned on a straightforward pass through the part of the network that deals with postnominal modifiers.

SETQ assigns a value to a variable that does not go on the register list, and therefore never enters the pushdown stack. For example, when a verbal particle follows a noun phrase, as in *put the cat out,* SETQ tries to make a temporary copy of the information about what particles the verb can take. If the dictionary lists no particles, SETQ fails and the resulting false condition prevents the arc from being tried. If the verb can take a particle, however, and the CAT PREP match succeeds, the temporarily stored information is then put into a register:

```
(VP/NP . . .
      (CAT PREP  (SETQ TEMP  (VPARTICLE V))
            (SETR V TEMP)
            (SETR HEAD TEMP)
            (TO VP/VP)).. . . )
```

COPY is used to transfer the dictionary information about a word rather than the word itself, as in

```
(S/Q  (CAT QWORD T
            (COND  ((GETF SUBJ/OBJ)
                              (SETR WHQ  (COPY '')))
                        (T     (HOLD  (COPY'')))
            (TO S/NP)))
```

(REVERSE LIST) has as its value a list derived from the one named but with its elements in reverse order. (GETROOT SOURCE CATEGORY) has as its value the root given by the dictionary for the word indicated by SOURCE with respect to the category named. (WRD BE (GETROOT '' V)) succeds if the current word is any verbal form of *be.*

ANOTHER EXAMPLE

A more complex example, which uses the most common matches, tests, and actions, is given in Figures 3 and 4. We will trace the grammatical analysis of a sentence through this network. The example, like the preceding ones, is adapted from Woods's earlier paper. The NP and PP parts of the network are nearly the same as in Figure 1, 1a, and 2.

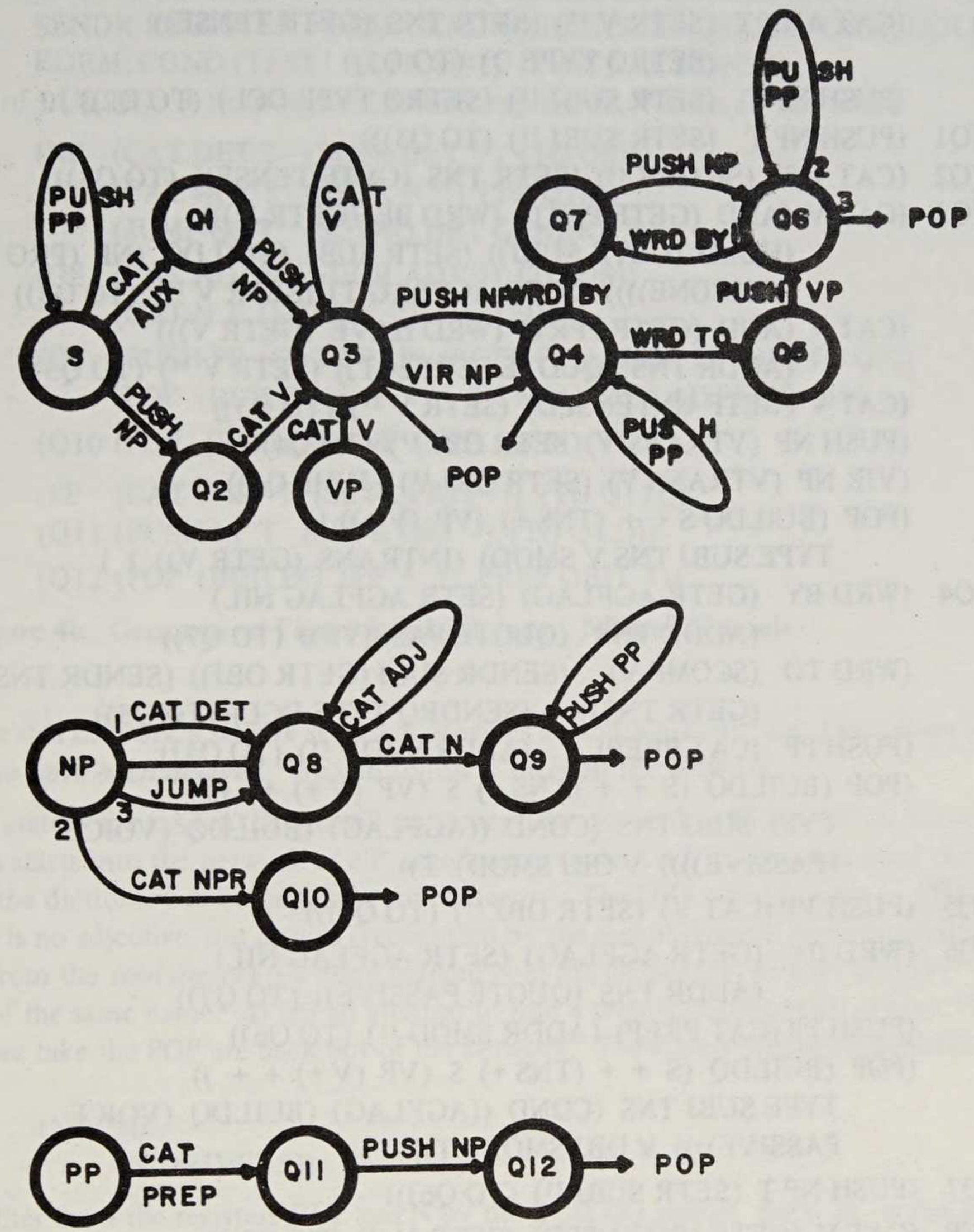


Figure 3. Enlarged transition network diagram for English grammar.

```
(S    (PUSH PP  (CAT PREP  (ADDR SMOD '') (TO S))
      (CAT AUX T  (SETR V '') (SETR TNS  (GETF TENSE))
                  (SETRQ TYPE Q)  (TO Q1))
      (PUSH NP T   (SETR SUBJ '') (SETRQ TYPE DCL)  (TO Q2)) )
(Q1   (PUSH NP T   (SETR SUBJ '') (TO Q3)))
(Q2   (CAT V T   (SETR V '') (SETR TNS  (GETF TENSE))  (TO Q3)))
(Q3   (CAT V (AND  (GETF PPRT)  (WRD BE  (GETR V)))
             (HOLD  (GETR SUBJ))  (SETR SUBJ  (BUILDQ  (NP  (PRO
             SOMEONE))))  (SETR AGFLAG T)  (SETR V '')  (TO Q3))
      (CAT V (AND  (GETF PPRT)  (WRD HAVE  (GETR V)))
             (ADDR TNS  (QUOTE PERFECT))  (SETR V '')  (TO Q3))
      (CAT V (GETF UNTENSED)  (SETR V '')  (TO Q3))
      (PUSH NP  (VTRANS V)  SETR OBJ '')  (TO Q4))
      (VIR NP  (VTRANS V)  (SETR OBJ '')  (JUMP Q4))
      (POP  (BUILDQ S +  +  (TNS +)  (VP  (V +)) +)
            TYPE SUBJ TNS V SMOD)  (INTRANS  (GETR V))  )  )
(Q4   (WRD BY   (GETR AGFLAG)  (SETR AGFLAG NIL)
                (ADDR TNS   (QUOTE PASSIVE))  (TO Q7))
      (WRD TO   (SCOMP V)    (SENDR SUBJ (GETR OBJ))  (SENDR TNS
                (GETR TNS))     (SENDRQ TYPE DCL)  (TO Q5))
      (PUSH PP   (CAT PREP)      (ADDR SMOD '')  (TO Q4))
      (POP  (BUILDQ  (S  +  +  (TNS +)  $  (VP  (V +)  +)  +)
            TYPE SUBJ TNS  (COND  ((AGFLAG)  (BUILDQ  (VOICE
            (PASSIVE))))  V OBJ SMOD)  T))
(Q5   (PUSH VP (CAT V)  (SETR OBJ '')  (TO Q6)))
(Q6   (WRD BY   (GETR AGFLAG)  (SETR AGFLAG NIL)
                (ADDR TNS  (QUOTE PASSIVE))  (TO Q7))
      (PUSH PP (CAT PREP)  (ADDR SMOD '')  (TO Q6))
      (POP  (BUILDQ  (S  +  +  (TNS +)  $  (VP  (V +)  +  +  ))
            TYPE SUBJ TNS  (COND  ((AGFLAG)  (BUILDQ  (VOICE
            PASSIVE))))  V OBJ SMOD)  T))
(Q7   (PUSH NP T  (SETR SUBJ '')  (TO Q6)))
(VP   (CAT V  (GETF UNTENSED)  (SETR V '')  (TO Q3)))
```

Figure 4a. Grammar of Figure 3 in LISP form. S level.

Additional arc types: WRD WORDLIST TESTWORD, VIR STATE
Tests: CAT CATEGORY, WRD WORDLIST TESTWORD,
GETF FEATURE, VTRANS REGISTER, INTRANS REGISTER,
SCOMP REGISTER, AND TEST TEST . . .

```
Actions:    ADDR REGISTER FORM, SETRQ REGISTER QUOTEFORM,
            SENDR REGISTER FORM, SENDRQ REGISTER QUOTEFORM, HOLD
            FORM, COND (TEST1 ACTIONS1)  (TEST2 ACTIONS2) . . .
Skeleton of BUILDQ:  + from register, $ from expression, '' from current word

            (NP   (CAT DET T   (SETR DET '')  (TO Q8))
                  (CAT NPR T   (SETR N '')  (TO Q10))
                  (JUMP Q8 T   (SETRQ DET GENERIC)))
            (Q8   (CAT ADJ T  (ADDR ADJ '')  (TO Q8))
                  (CAT N T  (SETR N '')  (TO Q9)))
            (Q9   (PUSH PP  (CAT PREP)  (SETR PP  (ADDR PP ''))  (TO Q9))
                  (POP  (BUILDQ  (NP  +  +  (N +)  +)  DET ADJ N PP)  T))
            (Q10 (POP   (BUILDQ  (NP  (NPR +))  N)  T))
            (PP    (CAT PREP T  (SETR PREP '')  (TO Q11)))
            (Q11 (PUSH NP T   (SETR OBJ '')  (TO Q12)))
            (Q12 (POP  (BUILDQ  (PP  +  +)  PREP OBJ)  T))
```

Figure 4b. Grammar of Figure 3 in LISP form. NP and PP levels.

For this network a passive sentence provides a good workout: say *The drivers were thought to have been delayed.* We will follow it through the network.

At state S the CAT AUX arc will not accept *the*, so we do a PUSH NP on the second arc. This starts into the network at NP, which accepts the CAT DET match since *the* is listed in the dictionary as belonging to that category. The DET register is set to THE. At Q8 there is no adjective, but *drivers* (recognized by the morphological analysis routines as built from the root *driver*) is a noun marked N in the dictionary, so goes into the register of the same name. At Q9 an attempt to find a prepositional phrase at state PP fails, so we take the POP arc back out of the network. The form it builds to send back up is

```
(NP THE
    (N DRIVERS))
```

put together from the registers DET and N by the BUILDQ action. Back up on the S-Q2 arc again, the first action puts the form that was built at the lower level into the SUBJ register, then setting the TYPE register to DCL to show that the sentence is declarative.

At Q2 the verb *were*, identified through the dictionary and morphological analyzer as a form of *be*, matches the CAT V arc. The word is therefore put into the V register, and its tense, recognized at the same time the root was found, is put into the TNS register. At Q3 the compound test on the first arc is met: the current word, *thought*, can take the feature PPRT, as indicated by the dictionary, and the verb in the V register is *be*. With this combination of conditions, the CAT V match succeeds. The actions, however, are here not the simple register setting actions like the ones performed up to this point.

Since the formation could be that of a passive, we change our minds about the status of the noun phrase found by S-Q2. Rather than continuing to call it the subject, we take it from the SUBJ register and put it on the HOLD list for reference later. Then we set up a dummy or impersonal subject (NP (PRO SOMEONE)) in the same register. The AGFLAG is set to T to keep the information that since this might be a passive sentence, the agent might appear later. With that information in AGFLAG, the verb *be* is redundant, and we can replace the contents of the V register by what is now the main verb *think.*

This time at Q3 the current word *to* is not a verb, nor does an attempt to analyze it as the beginning of a noun phrase work. This forces us along to the virtual arc VIR NP, which can be tried since *think* is listed as transitive (as in *he thinks important thoughts*). This virtual arc is matched not by finding a noun phrase in the text, but by being able to pull one off the HOLD list at this point. The noun phrase (NP THE (N DRIVERS)), which was at first thought to be the subject and was later put on HOLD, satisfies the match, and is therefore put now into the OBJ register.

At Q4 the AGFLAG is set, but the current word is *to,* not *by*, so the first arc fails. The test on the second arc is also met, in that *think* can take a sentence complement, so WRD TO matches. The result of this match is a series of actions in preparation for the forthcoming PUSH. The contents of the OBJ register, the noun phrase, are copied into the SUBJ register of the next level down, the tense is also sent down, and so is the type. At Q5, then, everything is ready for a PUSH VP.

The verb found at VP is *have*, which is in the untensed form required by the test. It is put into the V register (the lower level one corresponding to the PUSH, not the upper level one that contains think). At Q3 the CAT V match against *been* can be tried because the compound test is met: *been* is a past participle and V contains *have.* As a result PERFECT is added to the PAST that was sent down to this TNS register from the one above, *be* replaces *have* in the V register, and we are back at Q3 for the fourth time.

This time the test on the first arc succeeds because *delayed* has the feature PPRT and *be* is in the V register. CAT V matches, and so the HOLD action takes the noun phrase (NP THE (N DRIVERS)) out of the subject register, where it had been sent from the OBJ register of the upper level, puts it on the HOLD list for the lower level, and again substitutes the dummy (NP (PRO SOMEONE)) for it in the SUBJ register. The lower level AGFLAG goes on, and *delay* goes into the V register.

Back at Q3 for a fifth and final visit, the only arc that works is the VIR NP arc, made possible because *delay* can be a transitive verb. The VIR match takes the noun phrase off the HOLD list and puts it into the OBJ register. Q4 will not match either *by* or *to*, since there are no more words left. This forces the lower level network to POP with a value

```
(S DCL
     (NP (PRO SOMEONE))
     (TNS PAST PERFECT)
```

```
(VOICE PASSIVE)
(VP   (V DELAY)
      (NP THE
            (N DRIVERS))))
```

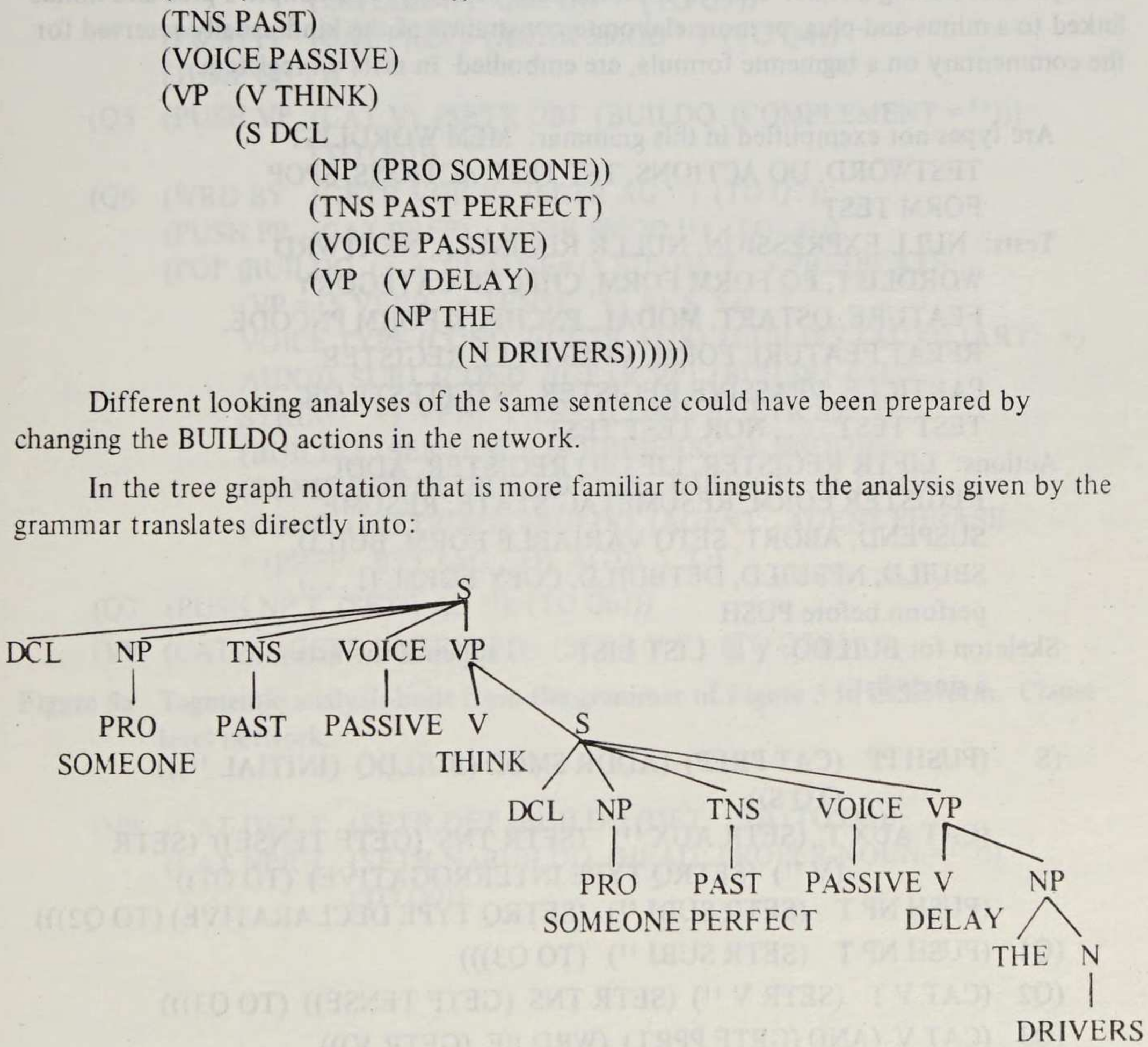
put together from various registers that were set either by information sent down from the main network or by what was found at this level.

The upper level returns to the arc Q5-Q6, which results in the entire lower level sentence being stored in its OBJ register. At Q6, since there are no more words to analyze, we take the final POP arc. This indicates that the analysis of the entire sentence by this grammar is

```
(S DCL
      (NP   (PRO SOMEONE))
      (TNS PAST)
      (VOICE PASSIVE)
      (VP   (V THINK)
            (S DCL
                  (NP  (PRO SOMEONE))
                  (TNS PAST PERFECT)
                  (VOICE PASSIVE)
                  (VP   (V DELAY)
                        (NP THE
                              (N DRIVERS))))))
```

Different looking analyses of the same sentence could have been prepared by changing the BUILDQ actions in the network.

In the tree graph notation that is more familiar to linguists the analysis given by the grammar translates directly into:



COMPARING MODELS OF LANGUAGE

In order to illustrate what happens when the model of linguistic structure is changed, Figure 5 gives the grammar of Figure 4 over again, but modified to give a tagmemic analysis as output. As is typical of tagmemic grammars, considerable information is put into the labels of tagmemes and construction types. The surface subject of passives is kept as the subject (eliminating the use of virtual arcs since nothing is out of place) instead of being treated as the logical object. In the report of the analysis there are none of the plus-and-minus signs usually associated with tagmemic formulas, and hence no plus signs. This is because the parse of a given utterance includes only those options that were in fact given a plus value in the particular reading of the generalized formula. Most tagmemic plus-and-minus signs are in fact translated into a network as multiple arcs leaving a state. Linked combinations of signs, for example a plus-and-minus linked to a minus-and-plus, or more elaborate constraints of the kind usually reserved for the commentary on a tagmemic formula, are embodied in tests on registers.

Arc types not exemplified in this grammar: MEM WORDLIST
TESTWORD, DO ACTIONS, TST/TEST ACTIONS, SPOP
FORM TEST
Tests: NULL EXPRESSION, NULLR REGISTER, NEXTWRD
WORDLIST, EQ FORM FORM, CHECKF CATEGORY
FEATURE, QSTART, MODAL, PNCHECK FORM PNCODE,
RFEAT FEATURE FORM, VPARTICLE REGISTER
PARTICLE, VPASSIVE REGISTER, NOT (TEST), OR
TEST TEST . . . , NOR TEST TEST . . .
Actions: LIFTR REGISTER, LIFTRQ REGISTER, ADDL
REGISTER FORM, RESUMETAG STATE, RESUME,
SUSPEND, ABORT, SETQ VARIABLE FORM, BUILD,
SBUILD, NPBUILD, DETBUILD, COPY FORM, (! . . .)
perform before PUSH
Skeleton for BUILDQ: (@ LIST LIST . . .) appends the lists into
a single list

```
(S    (PUSH PP  (CAT PREP) (ADDR SMOD (BUILDQ (INITIAL '')))
                (TO S))
      (CAT AUX T (SETR AUX '') (SETR TNS (GETF TENSE)) (SETR
                (V '') (SETRQ TYPE INTERROGATIVE) (TO Q1))
      (PUSH NP T  (SETR SUBJ '') (SETRQ TYPE DECLARATIVE) (TO Q2)))
(Q1   (PUSH NP T  (SETR SUBJ '') (TO Q3)))
(Q2   (CAT V T  (SETR V '') (SETR TNS (GETF TENSE)) (TO Q3)))
(Q3   (CAT V (AND (GETF PPRT) (WRD BE (GETR V)))
              (SETRQ VOICE PASSIVE) (ADDR VP (GETR V))
```

```
            (SETR V '') (TO Q3))
        (CAT V (AND (GETF PPRT) (WRD HAVE (GETR V)))
            (ADDR TRS (QUOTE PERFECT)) (ADDR VP (GETR V))
            (SETR V '') (TO Q3))
        (CAT V (GETF UNTENSED) (SETR V '') (TO Q3))
        (PUSH NP (VTRANS V) (SETR OBJ '') (TO Q4))
        (JUMP Q4 T) )
(Q4     (WRD BY (GETR VOICE) (SETR AG '') (TO Q7))
        (WRD TO (SCOMP V) (SENDR TNS (BUILDQ (IMPLIED +) TNS))
            (SENDR TYPE (BUILDQ IMPLIED DECLARATIVE))
            (COND ((GETR VOICE) (SENDR SUBJ (BUILDQ
            (IMPLIED +) SUBJ)))  (T (SENDR SUBJ (BUILDQ
            (IMPLIED +) OBJ ))))  (TO Q5))
        (PUSH PP (CAT PREP) (ADDR SMOD '') (TO Q4))
        (JUMP Q6 T))
(Q5     (PUSH VP (CAT V) (SETR OBJ (BUILDQ (COMPLEMENT = ''))))
            (TO Q6)))
(Q6     (WRD BY (GETR VOICE) (SETR AG '') (TO Q7))
        (PUSH PP (CAT PREP) (ADDR SMOD '') (TO Q6))
        (POP (BUILDQ (+ + CLAUSE= ($ SUBJECT: + PREDICATE:
            (VP = ($ VERB: + TENSE: +) $) $ $))
            VOICE TYPE (COND ((GETR AUX) (BUILDQ (AUXILIARY: +)
            AUX))) SUBJ (COND ((GETR VP) (BUILDQ (AUX
            STRING: +) VP))) V TNS (COND ((GETR SMOD)
            (BUILDQ (SENTENCE MODIFIERS: +) SMOD)))
            (COND ((GETR OBJ) (BUILDQ (OBJECT: +) OBJ)))
            (COND ((GETR AG) (BULDQ (AGENT: AGENT PHRASE
            = (PREP: BY) (AG: +)) AG))) ) T))
(Q7     (PUSH NP T (SETR AG '') (TO Q6)))
(VP     (CAT V (GETF UNTENSED) (SETR V '') (TO Q3)))
```

Figure 5a. Tagmemic analysis built from the grammar of Figure 3 in LISP form. Clause level network.

```
(NP     (CAT DET T  (SETR DET (BUILDQ (DET: ''))) (TO Q8))
        (CAT NPR T  (SETR N (BUILDQ (HEAD: PROPER-NOUN = ''))))
                    (TO Q9))
```

```
        (JUMP Q8 T   (SETRQ DET GENERIC))  )
(Q8   (CAT ADJ T   (ADDR ADJS (BUILDQ (MOD: '''))) (TO Q8))
        (CAT N T (SETR N (BUILDQ (HEAD: ''))) (TO Q9)))
(Q9   (PUSH PP   (CAT PREP) (ADDR MOSTMOD (BUILDQ (POSTMOD: '''))))
                    (TO Q9))
        (POP (BUILDQ (NP = + + + +) DET ADJS N POSTMOD) T))

(PP   (CAT PREP T (SETR REL (BUILDQ (RELATOR: ''))) (TO Q10)))
(Q10 (PUSH NP T   (SETR AX (BUILDQ (AXIS: ''))) (TO Q11)))
(Q11 (POP (BUILDQ (PP = +  + ) REL AS) T))
```

Figure 5b. Tagmemic analysis built from the grammar of Figure 3 in LISP form. NP and PP networks.

Taking our first test sentence, *Did George come?* through the grammar of Figure 5, we get the following analysis:

```
(INTERROGATIVE CLAUSE =
      (AUXILIARY: DID
      SUBJECT: (NP =
                  (HEAD: PROPER-NOUN = GEORGE))
      PREDICATE: (VP =
                        VERB: COME
                        TENSE: PAST)))
```

This presentation of the results differs from ordinary tagmemic analyses only in that all levels are analyzed in a single pass through the network, which is no disadvantage. The equals signs show expansions of the constructions whose labels are given as filler labels of tagmemes, so that the clause level analysis alone would be (AUXILIARY: DID SUBJECT: NP PREDICATE: VP).

A more complex analysis naturally results from a sentence like *The younger hikers were believed to have been delayed by the flood.* The initial S arc calls for a PUSH NP that that returns (NP = (DET: THE) (MOD: YOUNGER) (HEAD: HIKERS)), which is placed in the SUBJ register. TYPE is set to DECLARATIVE. At Q2 the V register is set to BE and TNS to PAST. The first time through Q3 VOICE is set to PASSIVE, VP to BE, and V to BELIEVE. Back at Q3 again, the JUMP arc is taken to Q4, which matches the word TO and therefore sends down preparatory information to registers for the next level of embedding: (IMPLIED PAST) to TNS, (IMPLIED DECLARATIVE) to TYPE, and (IMPLIED (NP = (DET: THE) (MOD: YOUNGER) (HEAD: HIKERS))) to SUBJ since the clause is in the passive voice. Q5 then begins the analysis of the embedded stretch at VP. VP finds untensed HAVE, which it places in the V register of the lower level. Q3 adds to the TNS register to give (IMPLIED PAST) PERFECT, sets VP to HAVE, and V to BE. The next time around Q3 sets VOICE to PASSIVE, adds BE to the HAVE already in VP, and sets V to DELAY. Q3 then gives a JUMP to Q4, which recognizes BY and puts it in the AG register. Q7 matches

(NP = (DET: THE) (HEAD: FLOOD)) and puts it in the AG register in place of BY, which is now redundant. The path of analysis returns to Q6, which is the end of the network. Its result is the entire complement part of the text. The upper level network picks it up at Q5 and puts it into the OBJ register. It in turn ends at Q6 with a POP that gives the complete analysis:

```
(PASSIVE DECLARATIVE CLAUSE =
     (SUBJECT:   NP =
                    (DET: THE)
                    (MOD: YOUNGER)
                    (HEAD: HIKERS))
   PREDICATE: VP =
                      ((AUX STRING: BE)
                      VERB: BELIEVE
                      TENSE: PAST))
(OBJECT: COMPLEMENT =
                  (PASSIVE (IMPLIED DECLARATIVE) CLAUSE =
                    SUBJECT: (IMPLIED (NP =
                                 (DET: THE)
                                 (MOD: YOUNGER)
                                 (HEAD: HIKERS)))
                    PREDICATE: (VP =
                                   ((AUX STRING: HAVE BE)
                                   VERB: DELAY
                                   TENSE: (IMPLIED PAST)
                                          PERFECT))
                    (AGENT: AGENT PHRASE =
                                 (PREP: BY
                                 AG: (NP =
                                        (DET: THE)
                                        (HEAD: FLOOD)))) )) ))
```

Even though the grammar of Figures 3, 4, and 5 is obviously incomplete as far as the English language is concerned, it does handle an involved subset of it gracefully, and would be a reasonable starting point for expanding the grammar.

THE DICTIONARY

Every grammar presupposes a dictionary and vice versa. The dictionary is where all the information that does not have a place in the grammar is kept.

A transition network grammar refers to the dictionary constantly. The most common reference is in the CAT match, which asks whether the word being examined belongs to a particular grammatical category or not. Words may belong to large open

categories like noun and verb, or to small closed categories like possessive pronoun and conjunction. The same entry may belong to more than one category: *permit* is both a noun and a verb, and so will satisfy either a CAT N match or a CAT V match.

Other references to the dictionary pick up more particular information. For example, the test in (VP/HEAD (JUMP VP/NP (RFEAT INTRANS V) . . .)) allows the JUMP arc to be taken if the word in the register called V is listed in the dictionary as INTRANS; if it is intransitive, there is no reason to look for a possible object for it. Another test looks for a feature of the current word:

```
(S/Q  (CAT QWORD T
          (COND  ((GETF SUBJ/OBJ)
                      (SETR WHQ  (COPY '')))
                 (T (HOLD  (COPY '')))))
          (TO S/NP)))
```

matches a question word on the basis of its grammatical category label. Before it decides what to do with it, however, it has to know whether it is a question word like *what* that can be either subject or object, or whether it is restricted in where it can come: *whom* can be an object but not a subject, and *where* cannot be either. In the first case the grammar proceeds on the assumption that *what* might be a subject, and in the second it knows right away that *whom* and *where* cannot be. This feature information comes from the dictionary.

The dictionary also contains information to guide the morphological analyzer for a language like English. In the Woods grammar it tells which of the regular plural formations a noun can take and which of the regular past formations a weak verb can take. Anything the morphological analyzer cannot handle on that basis is tagged as an irregular form with appropriate features.

Languages with a richer morphology than that of English can be analyzed the same way only at the cost of a huge dictionary that runs to one entry for each morphologically complex word. Instead it seems preferable to generalize the idea of features in another direction: to use them to specify what sets of affixes a root or stem can take in regular formation. With this information a generalized morphological routine can analyze the word itself by means of a network that recognizes possible morpheme sequences. As an initial measure for many languages, words can be divided into their constituent morphemes by the linguist, with the divisions indicated in the written form by hyphens. A preliminary action on the PUSH to a word analyzing network then removes the hyphens from the current word, so that the network operates in the usual way on it. This is preferable, at least in the early stages of analysis, to writing a new set of morphological analysis routines for each language. Another possibility is a word analyzing network that goes through words letter by letter.

Going back to the current form of the dictionary in the Woods implementation, some words are tagged as the initial words of compounds. The English morphological

analyzer makes an on the spot check of the words that follow such a word in the input to see if the compound is in fact present. If it is, then an artificially constructed word is substituted for the entire compound in the input. The dictionary contains an entry for that artificial word, and the grammar goes on as though that were what was actually in the text.

Where a language makes use of separable verb particles as in English *throw the other party out, close the factory down,* a similar strategy is followed, but only at the point where the particle itself is encountered. At that point the verb *throw,* for example, is replaced in the verb register by the artificial *throw-out* its features replace those of the original verb, and the analysis continues.

The inverse capability allows an abbreviation or a contraction to be expanded out to its normal form.

The LISP form of a dictionary entry (there seems to be no reason to have a diagrammatic form) consists of two parts: the **definiens** or word being defined and a list, enclosed in parentheses, of information **pairs**, giving the form (DEFINIENS (PAIRLISTS)). The first member of each pair is a key to the kind of information contained in the second member. For example, the entry for *give* might be (GIVE (V IRR FEATURES (PASSIVE TRANS INDOBJ))). *Give* is the definiens. V IRR is the first pair. The first member of that pair, V, is the **lexical category symbol** that a CAT V match would look for. The second member is a **morphology code** used by the English analyzer to indicate that since the past is formed irregularly, it must be looked up in another entry. The second pair consists of the word FEATURES followed by a list of features associated with the word. They tell us that it can form a passive, it is transitive, and it can take an indirect object.

The counterpart entry (GAVE (V (GIVE (PAST)))) has a single pair associated with the definiens: again the lexical category symbol V, followed this time by a **root feature list.** The first part of the root feature list gives the root under which the rest of the information for this word is stored, while the second part gives a list of inflectional features (of which there is only one, PAST, in this case) that are to be appended to the features under the main entry. If *gave* is encountered in a text it will therefore be pulled apart by the analyzer into the root *give* and the feature list (PASSIVE TRANS INDOBJ PAST).

Lexical category pairs like the ones illustrated can have four kinds of values paired with the category symbol. Morphology codes include not only IRR for irregular forms, but indications of how the regular plural is formed for words of the noun (N) category and how the third person singular and the regular past and past participle are formed for verbs. For words that involve no morphological analysis, '' directs the morphological analyzer to put the text form itself out as the word that is recognized. The root feature list has been illustrated in its simple form: a root followed by a list of inflectional features associated with it for the definiens. The fourth form is a list of root feature lists enclosed in parentheses, each giving a different interpretation of the word.

A feature pair has already been illustrated; it consists of the word FEATURES followed by a list of feature names. Features named are assumed to be present and features not named are assumed to be absent unless their presence can be deduced from the existing feature list by redundancy rules built (at present) into the morphological analyzer. **A particle pair** consists of the word PARTICLES followed by a list which in turn is composed of pairs: a verb particle and the artificial definiens to be substituted for the verb if the combination is found:

```
(TAKE  (V IRR
        FEATURES   (PASSIVE TRANS)
        PARTICLES  (UP TAKE-UP
                    OUT TAKE-OUT
                    AWAY TAKE-AWAY))).
```

Compounds are entered in a **compound pair**, the first member of which is the word COMPOUNDS. The second member is a list of **compound trees** enclosed in parentheses. The tree itself consists of a word, a **result**, and if necessary another tree, with a pair of parentheses enclosing the three. If the word that begins the compound tree is found next in the text after the current word, the result associated with it is looked at. If it is the last word of a compound, the result that goes with it is an artificial word that stands for the entire compound and which then replaces the compound in the text. If the word that begins the tree is not the last word of a compound, the result is NIL and the compound analyzer keeps going. Even if a word could end a compound, if there is another compound that could continue on and that includes all the words that have been recognized for the first compound, the compound tree following the result gives the next word, and so on. The compound analyzer gives the result that corresponds to the longest possible match with the text. In an entry like

```
(EUROPEAN  (ADJ ''
            COMPOUNDS     ((COMMON NIL
                                (MARKET EEC
                                 (SECRETARIAT
                                  EEC-SEC)
                                 (COMMISSIONER
                                  EEC-COM))
                            (COUNCIL EUROCOM))
                           (ECONOMIC NIL
                                (COMMUNITY EEC)))))
```

the compounds *European Common Market* and *European Economic Community* are both replaced by *EEC* (which must then be entered in the dictionary), *European Common Market Secretariat* is replaced by *EEC-SEC, European Common Market Commissioner* by *EEC-COM,* and *European Common Council* by *Eurocom.*

A substitute pair does the opposite of what a compound pair does. it takes a single form like an abbreviation and replaces it by a string of words. The first member of the

pair is SUBSTITUTE and the second a list of words to be put into the input string in place of the one that was found. (ETC. (SUBSTITUTE (AND SO FORTH))) replaces the abbreviation *etc.* in text by the words *and so forth.*

For grammars of languages other than English it is useful to have an English gloss or definition as part of the dictionary entry. In a dictionary for Huichol the following entry might appear:

```
(QEI    (V ''
         FEATURES   (TRANS DEVEL UNITARY)
         ENGLISH    (CARRY ONE OBJECT IN THE HAND)))
```

This type of pair is not implemented in the Woods system; the current alternative is to include information about English in the form of a comment placed within the entry. Comments, which can also be interspersed throughout the grammar as needed, have the form (* . . .). Further development is needed, however, to make it possible for the results of a grammatical analysis to be displayed in a way that includes both the forms found in the text with their grammatical interpretation and the corresponding glosses for each form in English. Furthermore, it should be possible to attach a different English gloss to each area of meaning of a word.

The dictionary component of transition network grammars is at present heavily oriented toward the English morphological analyzer built into the Woods system. As more experience is gained with languages that have different morphological structures it is likely that this component will be modified in the direction of greater generality.

DEVELOPING A GRAMMAR

The simplest way to write a transition network grammar is to begin with a pencil sketch of the most common constructions in the language in pictorial form. It is relatively easy to draw out the main constructions in terms of the necessary matches as in Figure 1; tests and actions can come later.

The second step is to look for parts of the network that are identical. These should be merged, either by looping the network back upon itself or by combining them into a single subnetwork that is reached from various places by PUSH arcs.

Parts of the network that are similar to each other but not identical are merged next, using tests or conditional actions to distinguish the parts that are not alike. For example, in English questions and relative clauses share many properties, and cleft sentences have features of both. This similarity is expressed by having them use common portions of the network.

As the network grows the dictionary should grow with it. Each time a test involving features is put into the network the dictionary should be reviewed so that the feature is added to the right words.

Each time a POP arc is reached the structure building actions put together a piece of the analysis that is to be reported back to the grammar writer. This analysis does not have to present its components in the order in which they were found. The structure building scheme that is reported back from the POP arcs reflects the theoretical orientation of the grammar writer far more closely than anything else in the network. His view of language prescribes a particular kind of underlying structure as appropriate for language, and the POP arcs are where he puts together a representation of that structure.

There are three important tradeoffs in the construction of a transition network grammar. The first is in the choice of a PUSH match in lieu of a longer, more complex set of arcs. There are cases, of course, in which the recursive nature of language forces the use of a PUSH match. As we saw in Figure 1, trying to build the NP network as part of the S network would not only result in duplication; it would be impossible. It may, however, be desirable to use a subnetwork reached by PUSH even when infinite regress is not the problem. This is especially so when the POP that matches the PUSH is an appropriate place to put together a significant fragment of structure that is relatively independent of any information that was not collected within that subnetwork.

The second tradeoff involves subnetworks that are nearly alike but not identical. Merging parts of the network with tests to determine what part is actually being traversed is desirable in most cases, since the common set of paths expresses some generalizations about the language better than could be done with separate networks involving more paths. There comes a point, however, when the tests that are used to distinguish constructions become so complex that they are no longer worth the trouble. As Woods points out, the logical end point of merging could be a network that consists of a single arc upon which are heaped a battery of conditional actions; but this would not tell very much about the language.

This point leads to the third tradeoff between clarity of presentation of linguistic structures involved on the one hand and logical efficiency or cuteness on the other. The reason for writing a transition network grammar, like any other grammar, is largely to let human beings know how a language is put together. Transition networks are a transparent means of doing this, since they show how the elements of language go together in a graphic form that is easy to follow. On the other hand, it is possible to cut corners in a grammar in a way that takes advantage of the logical possibilities of the system at the cost of linguistic clarity or perspicuity. This temptation is especially great when a computer test of the grammar is involved, since it may be easier for the grammar writer to write a section of a network that gives a correct analysis without making it clear what is going on from the linguistic point of view. This sacrifice of perspicuity for efficiency is justified in utilitarian grammars like the original Woods grammar, whose purpose is to put together commands for an information retrieval system, but would be a disadvantage in a grammar of a previously unstudied language. (Actually the Woods grammar is quite clear from the linguistic point of view.)

By the time the grammar has grown to any size at all, a point will be reached at which it takes longer than it is worth to redraw the network as each new modification is fitten in. Furthermore, as the size of the network increases, it becomes increasingly difficult to fit in information about tests and actions along the arcs in the diagram. The LISP form of the grammar gives a place for tests and actions, but it is awkward to add new arcs into it unless it is being stored in a computer as it is written and revised. For transition network grammars written by hand a card file is probably more useful than a manuscript.

Each card represents one arc of the grammar. The state at which the arc begins is written in the upper left hand corner of the card and the terminal action that transfers to the next state is written in the upper right. If more than one arc leaves a state, each card is numbered in the top middle to show the order in which it is to be tried.

There are three kinds of information on the body of the card: the match or arc type, the test that permits the match to be tried, and the actions to be taken. They can be written as they would be in LISP form, preferably in pencil to permit revision. It is helpful to fill in the lower right hand portion of the card with examples or references to examples in a test corpus that make use of that arc. Figure 6 gives a card that represents the first arc in Figure 4.

The dictionary can be kept on similar file cards, preferably of a different color. In the upper left hand corner of the dictionary card is the definiens. Each pair in the pair list goes on a line below it. Figure 7 shows a dictionary card.

S	1	TO Q1
CAT AUX		
T		
SETR V ''		has anyone come?
		did the motor work?
SETR TNS (GETF TENSE)		are you ready?
		will you marry me?
SETRQ TYPE Q		could it be raining?

Figure 6. File card representing one arc.

From a set of arc cards it is fairly simple to draw up a diagrammatic representation of the grammar. Sort the cards by initial state and by order of trial. Choose the cards that begin at the starting state and lay them out on a table. Then pick the cards that go from each state reached from the initial state and lay them out. Continue until the entire network is laid out on the table, then draw a condensed version from that layout.

GIVE

V IRR

FEATURES (PASSIVE TRANS INDOBJ)

Figure 7. Dictionary card representing one entry.

CHAPTER 4

VERB CLUSTERS OF LAMNSOK

Karl Grebe

This chapter describes some of the grammatical structures of Lamnsok[1] by the augmented transition network grammar model developed by W. A. Woods (1970, Woods and Kaplan 1971). The author had not previously done any extensive description of the grammar of Lamnsok, so that this work should help to evaluate the usefulness of the Woods grammar model at the initial stage of language analysis. The model has been applied to the phenomena of verb clusters and sequences of verbs without expressed subjects within a sentence.

The augmented transition network model was designed by Woods for computer understanding of English; but even though Woods intended to build a grammar which could be handled by machines the networks of his grammar are very close to a model of the actual structures of language. For an example of such a network see Figure 1.

Woods's grammar is a network consisting of a finite number of states connected by arcs. The arcs are labelled and serve as paths which can be followed from one state to another. However, a transition from one state to another on a certain arc is only permitted if the input can match the label on the arc. If an arc leaving the first finite state of the network is labeled PRON, for example, the transition can only be followed if the first word of the input sentence happens to be a pronoun.

Furthermore, the condition for following a state transition is not limited to examining a single input symbol. The label on the arc can be something like NP/, where NP/ is the state that begins a subnetwork. Beyond the idea of recursive transition networks, a Woods grammar calls for certain actions to be taken at every transition and to store the information of the action in registers from where it can be retrieved to build up parts of the structure or the final structure of the input string. Conditions are

[1] Lamnsok is a Benue-Congo language spoken in the northeastern part of Bamenda Province in the United Republic of Cameroun. The author, under the auspices of the Summer Institute of Linguistics, has been engaged in language research on Lamnsok since December 1971. This work is the outcome of a linguistic workshop held in Nigeria from October to December 1972 under the direction of Joseph Grimes, partially funded by National Science Foundation grant GS-3180ANo.1. A concordance of Lamnsok texts was arranged by computer under National Science Grant GS-1605, and was used in preparing this paper.

The phonemes of Lamnsok are represented here by orthographic symbols with phonetic realizations where necessary. All the consonants of the language may appear word initially: voiceless stops t, c, [tš], k, kp, voiced stops b, d, j [dž], g, gb, fricatives f, v, s, sh [š], gh [ɣ], nasals m, n, ny [ɲ], ng [ŋ], liquids l, r, semivowels y, w. Word final consonants are v, m, n, ŋ, r, y, k [?] only syllable final. There are six vowels: i, e, oe [ə], a, u, and o. All of these may be lengthened, shown by writing them double. Tone is significant but is written here only where minimal pairs are involved.

also added to the arcs, which must be satisfied in order to follow the arc and execute the structure building actions.

When describing a language for linguistic purposes the conditions on the arcs can be stated in such a way as to reflect many regularities of the language. The labels of the arcs correspond to parts of the input sentence and the names of states can be given in linguistic terms so that they show the place to which the input sentence has advanced.

1. CLAUSE SERIES

Sentences in Lamnsok may contain an indeterminate number of verbs with the same subject. The subject always precedes the first verb of such a cluster. The other verbs may occur within the sentence in five positions relative to another verb:

(1) Immediately following another verb: **wu du yun nyam wiy fer nyoo sho** (she go buy meat come make soup with) 'She goes and buys meat, then comes and makes soup with it.'

(2) Separated from other verbs by postverbal verb phrase constituents such as objects, instrumentals, locatives, and adverbs: **wu du way yun nyam wiy sho fo lak fer nyoo sho.** (She go market buy meat come with to home make soup with) 'She goes to the market and buys meat and comes home with it and makes soup with it.'

(3) Separated from other verbs by preverbal auxiliary constituents: **wu du a yun vifa** (she go indefinite-time buy things) 'She went and bought things.'

(4) Separated from other verbs by both preverbal auxiliary constituents and postverbal verb phrase constituents: **wu du way a yun vifa** (she go market indefinite-time buy things) 'She goes to the market and will be buying things.'

(5) Separated from other verbs by clause relators together with any of the other combinations presented in (1)–(4): **wu du boo wiy** (He go sequence-immediate come) 'He is going and then returns immediately.' **wu du bo a wiy** (he go sequence indefinite-time come) 'He is going and will return sometime.' **wu du way bo a wiy** (He go market sequence indefinite-time come) 'He is going to the market before he will return sometime.'

1.1. CLAUSE RELATORS.

Clause relators are particles which relate clauses to one another in respect to time and logic. They fall into two main groups, those that join only clauses with expressed subjects and those that may join clauses that do not have expressed subjects. The latter group is under consideration here in relation to clause series.

Clause relators of clause series are:

e which emphasizes result, includes immediacy, and may also join full clauses,

bo which emphasizes sequence and may also join full clauses,

boo which emphasizes sequence and includes immediacy, but occurs in clause series only,

fo which implies purpose and occurs in clause series only,

foo which emphasizes purpose and includes immediacy, and occurs in clause series only.

Some other clause relators that can be employed only with clauses that have expressed subjects are: **a** 'if', **bik fo** 'because', **a** 'coordinate; and, but ', and **ji** 'purpose'.

1.2. CLAUSE SERIES VS. VERB PHRASE STRING. The question arises whether sentences with one subject and many verbs consist of multiple verb phrases or a series of clauses. The latter interpretation has been adopted for the following reasons:

(1) Subject deletion. First, every verb in a sentence has a retrievable subject. If the subject is not expressed, it is identical with the nearest subject of another clause. In the semantic underlying form of a clause series each verb together with its subject forms the minimal clause structure of subject plus verb, whether that subject is expressed or not. For example, **m du yun kifa** (I go buy thing) 'I go and buy something' equals semantically ***m du m yun kifa** (I go I buy thing) even though it never appears that way on the surface because the subject of the second clause is deleted.

(2) Expansion. Second, each clause, even though it may appear in the sentence as only a verb, may be expanded by adding other clause constituents: **m du yun kifa** (I go buy thing) 'I go and buy something', **m du way yun kifa** (I go market buy something) 'I go to the market and buy something'. **m du way yii a yun vifa** (I go market future indefinite buy things) (I go to the market and will be buying things.'

(3) Third, a clause series may employ clause relators to show various relationships between the individual clauses. Some of these clause relators can occur only in a clause series (**boo, foo**); but the clause relators **bo** and **e** can also relate clauses that have expressed subjects. The fact that **bo** and **e** also join clause series shows that clauses in a series are really separate clauses with deleted subjects. For example, **bo** occurs in a clause series: **wu sanri menkak bo a ton veoy** (He split firewood before indefinite burn fire) 'He is splitting firewood before making fire.' **bo** also occurs with clauses that have expressed subjects: **wu sanri menkak bo yeelav wa ton voey** (He split firewood before housewife indefinite burn fire) 'He splits firewood before the housewife will be making fire.'

(4) Fourth, a clause series employs the same system of deletion and identification of anaphoric references to things other than the subject as is used in clauses that have expressed subjects. These anaphoric patterns include objects, instruments, associatives, and locatives.

The object of any verb is deleted within the sentence if it is identical with the closest object in the sentence. This takes place within a clause series: **wu sanri menkak ton** (He split firewood burn) 'He split firewood and burned it.' (*Firewood* is deleted). The same thing takes place across the boundaries of clauses with subjects: **wu sanri menkak e yeelav ton** (He split firewood then housewife burn) 'He splits the firewood and then the housewife burns it.' (*Firewood* is deleted).

There is a restriction on this rule: if the nearest object is not semantically compatible with the verb under consideration, then the object deletion rule will look further back to find a compatible object: **wu fer nyoo su vifa yi** (She make soup wash dishes eat) 'She makes soup then washes the dish then eats.' The deleted object of eat **yi** is **nyoo** 'soup', not **vifa** 'dishes', which is the nearest object before the verb.

A second anaphoric pattern involves **sho**, an anaphoric reference particle which functions in full clauses and in clause series. The referent of **sho** is always the nearest object or associative of a different clause from the one in which **sho** is found: **wu yun nyam fer nyoo sho** (she buy meat make soup with) 'She buys meat and makes soup with it.' In this example **sho** is the instrumental and 'meat' refers back to **nyam** in the clause before it. However, the meaning of **sho** may be either instrumental, associative, or locative. The meaning of **sho** is mostly determined by the verb it follows, sometimes also by its position within that verb phrase.

Sho is ordinarily instrumental if the verb is an action process verb, V(a-p). An action process verb implies the semantic role relationship of an agent acting out a process upon a patient, like **fer** 'make' in the example just given.

With a verb whose meaning involves changing the position of something, **sho** is less likely to be instrumental and more likely to be locative: **wu yun kibam way kifa sho** (He buy bag put thing there) 'He buys a bag and puts something into it.' Here **sho** is a locative referring to **kibam** 'bag' because **way** 'put' is not an action process verb. It has an object and is a transitive verb, in surface grammar, but it differs from **fer** 'make' in that its patient does not go through a process, but rather a change in physical position. Some other verbs that take locative **sho** are **way** 'put', **tir** 'place upright', **lem** 'keep', **liti** 'hide something', **neer** 'put down', **neem** 'lie down'.

With motion verbs **sho** is usually associative but can also be instrumental in a few cases: **wu ki yun alabak wiy sho folak** (He past buy shoes come with to-home) 'He bought shoes and came home with them.' In this example **sho** is associative with a motion verb in the usual pattern. The context may, however, make it clear that **sho** is instrumental with a motion verb: **wu ki yun alabak way kiwuu wiy sho fo lak** (He past buy shoes put foot come with to home) 'He bought shoes, put them on and came home with them.' The clause **way kiwuu** 'put foot' makes it clear that the agent is using the shoes as instrument when coming home rather than just bringing them, the usual meaning of 'come with'.

With some motion verbs that take a range element, it is the position of **sho** within the verb phrase that determines whether it is instrumental or associative: **wu ki yun alabak kok gwan sho** (He past buy shoes climb hill instrumental) 'He bought shoes and climbed a hill with them.' **wu ki yun alabak kok sho gwan** (He past buy shoes climb associative hill) 'He bought shoes and brought them up a hill.'

With verbs that have no patient, but an agent who is not in motion towards a goal, **sho** is a locative. Such verbs include pure action verbs such as **tong** 'cry', **kpu** 'die', **buune**

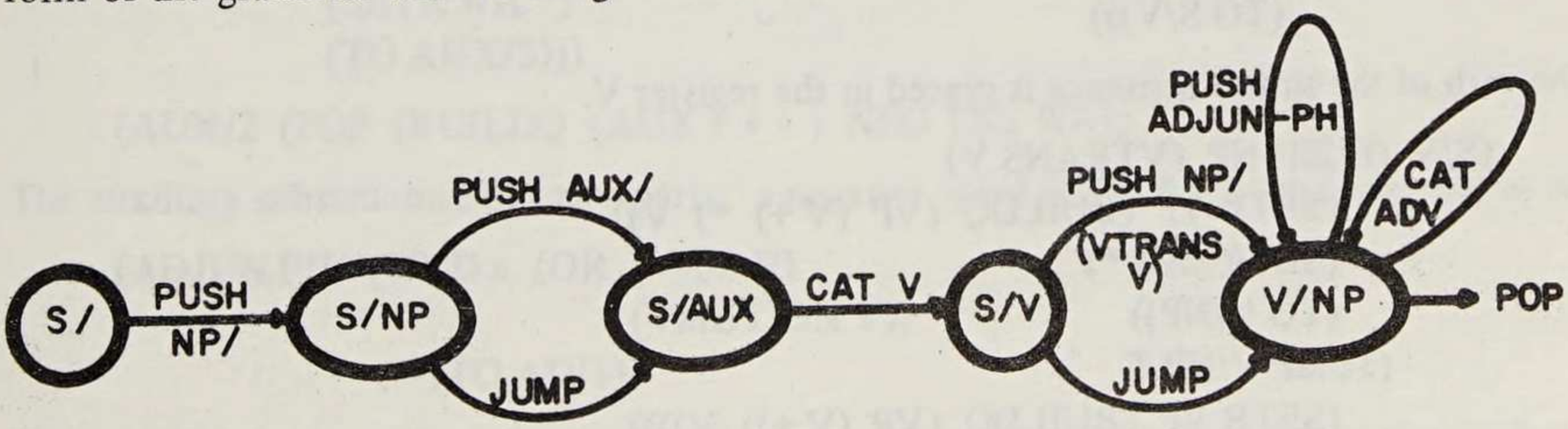
'sleep', **shiiy** 'stay', **tiy** 'stand', and **doe** 'sit'. **wu bakri kikun a buune sho** (He build bed indefinite sleep there) 'He builds a bed and will be sleeping on it.'

Anaphoric **fo** always refers back to the nearest locative and always means the same as its referent: **wu du sum lim lim fo** (He go farm work work there) 'He goes to the farm and works there.'

When the **sho** anaphoric rule and the object deletion rule apply to the same verb, the **sho** rule operates first, before we search for the object of the verb by the object deletion rule. The object cannot have the same referent as **sho**: **wu kiv ngwasang, wiy sho fo lak, nangsin ngwak, way sho** (He harvest maize, come with to home, prepare bin, put there) 'He harvests maize, brings it home, prepares a bin, and stores it there.' The **sho** rule looks back for the nearest object of a different clause, which is **ngwak** 'bin', and takes it as the locative. The object deletion rule then looks for the object which is expected by **way** 'put' because it is transitive; but **ngwak** 'bin' cannot be used again since it was used by the **sho** rule. The next object back is **ngwasang** 'maize', which is recognized as the deleted object of **way** 'put'.

1.3. NETWORK DESCRIPTION OF CLAUSE SERIES. To describe the clause series by Woods' transition network model we begin by charting a simple network for a one clause sentence, then revise that network to take in sentences that consist of clause series. The network in Figure 1 is kept simple to account only for a limited class of one clause sentences in order to facilitate reworking it to take in clause series. Most of the clause constituents shown in the diagram can be considerably more complex than this grammar implies, particularly the noun phrase; but they are left in rudimentary form throughout this paper. Their description will be expanded in later studies.

A sentence is followed through the diagram in Figure 1 by starting at the state S/ and proceeding from there to other states on arcs that permit a transition. The arcs labelled PUSH mean that one should enter the sub-network named with the PUSH, and follow through it, before returning to the higher level network at the point of departure with the construction analyzed by the lower level network. In the diagram the descriptions of the arcs are limited to arc labels plus a few conditions. The full description of conditions and actions associated with the arcs is given below in the LISP compatible form of the grammar with running comments.



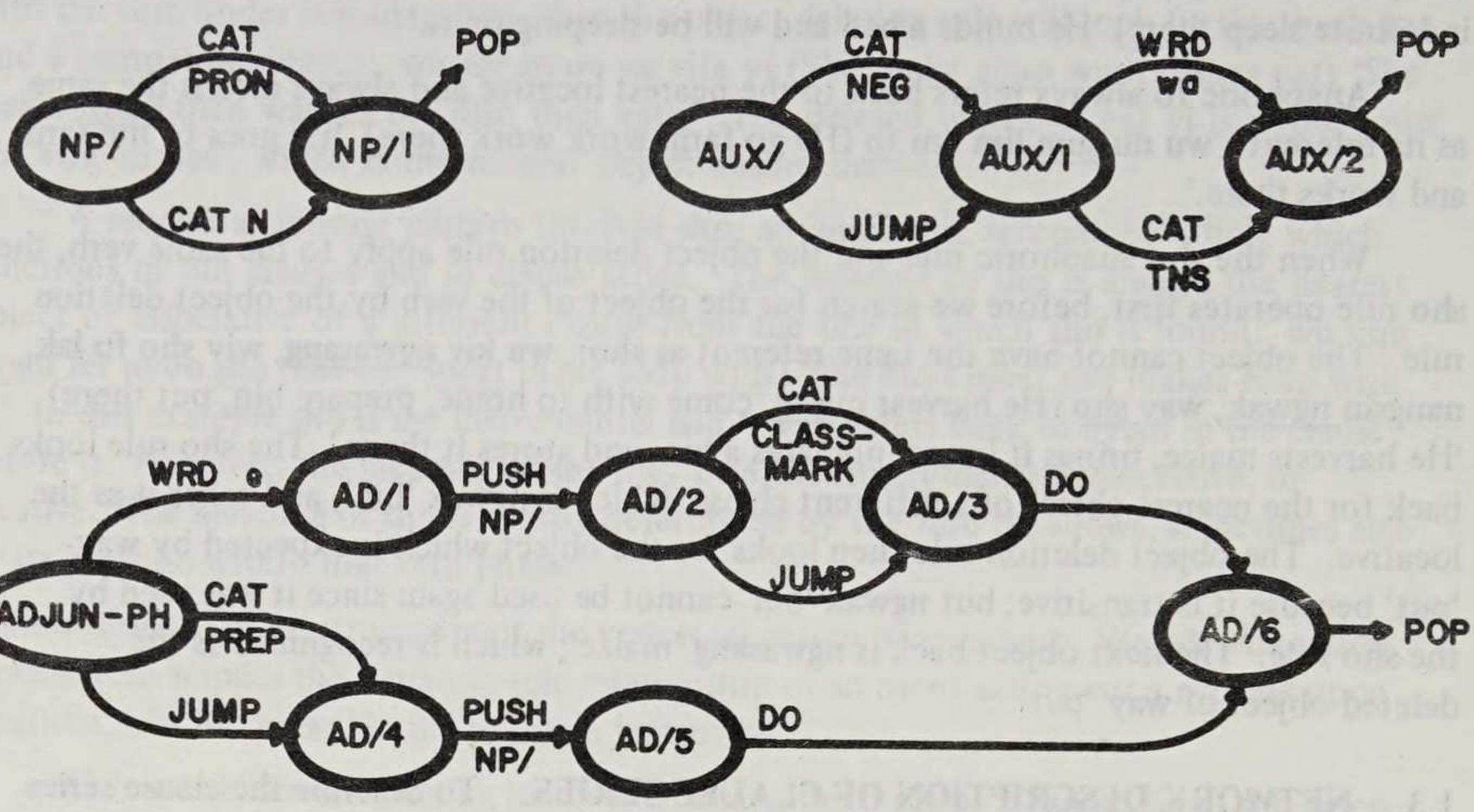


Figure 1. One clause sentences in Lamnsok.

```
(S/  (PUSH NP/  T
       (SETR SUBJ *)
       (TO S/NP)))
```

This arc finds the subject of the sentence and sets it into the SUBJ register.

```
(S/NP  (PUSH AUX/  T
         (SETR AUX *)
         (TO S/AUX))
       (JUMP S/AUX T))
```

The second arc looks for an auxiliary structure which may or may not be present (hence the JUMP arc).

```
(S/AUX  (CAT V T
           (SETR V *)
           (TO S/V)))
```

The verb of the simple sentence is placed in the register V.

```
(S/V  (PUSH NP  (VTRANS V)
        (SETR VP  (BUILDQ  (VP  (V +)  *)  V))
        (SETR OBJ *)
        (TO V/NP))
      (JUMP V/NP T
        (SETR VP  (BUILDQ  (VP  (V +))  V))))
```

If the verb in V is transitive, look for an object. If there is one, join verb and object in a VP structure. If not, report a VP that contains only the verb.

```
(V/NP   (PUSH ADJUN-PH T
            (SETR VP (APPEND (GETR VP) *))
            (TO V/NP))
        (CAT ADV T
            (SETR VP (APPEND (GETR VP)
                             (BUILDQ (ADV *))))
            (TO V/NP))
```

There may be several postverbal adjuncts but they are all added to the VP structure.

```
        (POP (BUILDQ (S + + +) SUBJ AUX VP) T))
```

At the end of the sentence return a picture of the constituents that were found, related in terms of the appropriate linguistic structure.

```
(NP/  (CAT PRON T
          (SETR N (BUILDQ (PRON *))
          (TO NP/1))
      (CAT N T
          (SETR N (BUILDQ (N *))
          (TO NP/1)))

(NP/1 (POP (BUILDQ (NP + ) N)) T)
```

This mini grammar of noun phrases reflects the emphasis of this paper on clause clusters; there is considerable complexity in noun phrases that is not mentioned here.

```
(AUX/   (CAT NEG T
            (SETR NEG *)
            (TO AUX/1))
        (JUMP AUX/1 T))

(AUX/1 (CAT TNS T
            (SETR TNS *)
            (TO AUX/2))
       (WRD wa
            (SETR WA *)
            (TO AUX/2)))

(AUX/2 (POP (BUILDQ (AUX + + + ) NEG TNS WA)) T)
```

The auxiliary substructure has two parts: a negative word and either a tense particle or **wa**.

```
(ADJUN-PH/ (WRD e (OR (VA-P V)
                      (VMOTION V))
           (TO AD/1))
```

With action-process and motion verbs **e** 'in' introduces a kind of prepositional phrase that may, however, take a final class marker not shared by ordinary prepositional phrases.

```
(CAT PREP T
     (SETR PREP *)
     (TO AD/4))
(JUMP AD/4 T))
```

A noun phrase may follow another as a locative and be equivalent to a prepositional phrase. There are other prepositional phrases which need to be distinguished in this network at a later study.

```
(AD/1   (PUSH NP T
             (SETR OBJ *)
             (TO AD/2))
(AD/2   (CAT CLASS-MARK
             (SETR CLASS-MARK *)
             (TO AD/3))
        (JUMP AD/3 T))
(AD/3   (DO   (SETR NP CLASS-MARK)) INST/ASSO (BUILDQ (INST/ASSO e + +)
              (TO AD/6)))
(AD/4   (PUSH NP T
             (SETR OBJ *)
             (TO AD/5)))
(AD/5   (DO  (SETR LOC  (BUILDQ (LOC # +)
                         (COND   ((GETR PREP)
                                  (BUILDQ (PREP +) PREP))
                                 (T NIL))
                         OBJ))))
(AD/6   (POP  (BUILDQ  (ADJUN-PH + +)  INST/ASSO LOC)   T))
```

This simple grammar analyzes sentences such as the following: (1) **wu ki du fo lak e kiwuu ki** (He past go to home on foot class-marker) 'He went home on foot.'

```
(S (NP (PRON wu))
   (AUX (TNS ki))
   (VP   (V du)
         (ADJUN-PH (LOC   (PREP fo)
                          (NP (N lak)))))
         (ADJUN-PH (INST/ASSO (e (NP (N kiwuu))
                                 (CLASS-MARK ki))))))
```

This is equivalent to the more familiar tree representation of grammatical structure given by Figure 2.

(2) **yeelav ton voey lav** (housewife burn fire house) 'The housewife is burning a fire in the house

```
(S (NP yeelav)
   (VP (V ton)
       (OBJ voey))
   (ADJUN-PH (LOC (NP (N lav))))  )
```

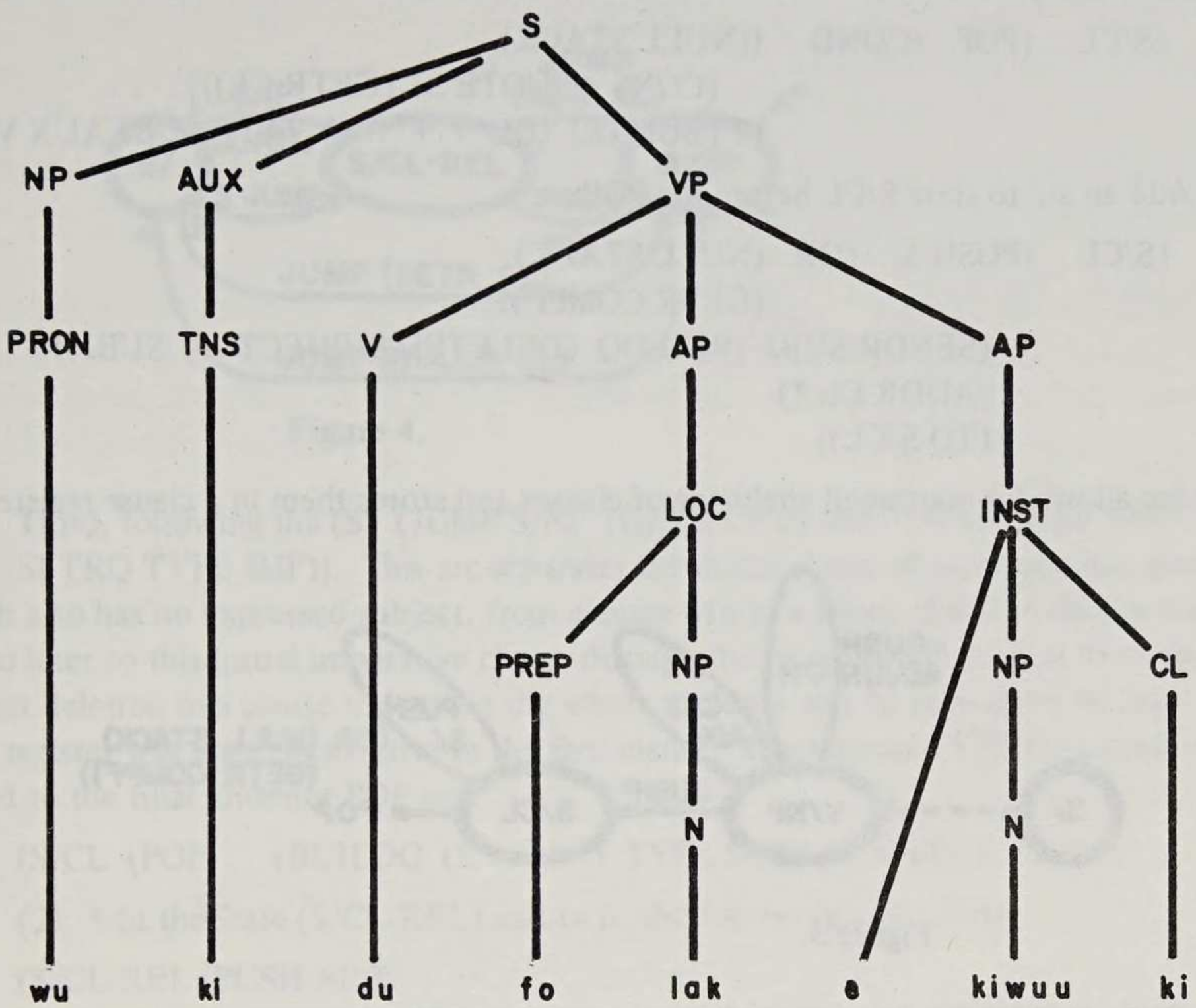


Figure 2.

2. EXPANSION OF ONE CLAUSE SENTENCE

2.1. EXPANSION TO GENERATE MULTIPLE CLAUSE SENTENCES. To describe multiple clause sentences the grammar needs an arc to build the first clause of the sentence and after that another arc to allow for repeated analyzing of clauses before the final building of the full sentence structure on the POP arc. This is accomplished as follows:

(1) Add a JUMP arc as the last one leaving state V/NP. It leads to a new state S/CL. The action on the arc builds the clause structure if we are on the sentence level, expressed by (NULL STACK).

```
(V/NP   (JUMP S/CL T
            (COND ((NULL STACK)
                        (SETR CL (BUILDQ (CL + + + +)
                                         (TYPE SUBJ AUX VP))))))
```

(2) Transfer the POP arc of the sentence network from state V/NP to S/CL and change the action on it so that it will pop any clause from a lower level to the sentence level as

well as building the final sentence structure

```
(S/CL  (POP  (COND  ((NULL STACK)
                     (CONS (QUOTE S) (GETR CL)))
                    (T (BUILDQ (CL + + + +) TYPE SUBJ AUX VP))))))
```

(3) Add an arc to state S/CL before the POP arc:

```
(S/CL  (PUSH S/  (OR  (NULL STACK)
                      (GETR COMPF))
                (SENDR SUBJ (BUILDQ (DELETED SUBJECT +) SUBJ))
                (ADDR CL *)
                (TO S/CL))
```

This arc allows for continued analyzing of clauses and stores them in a clause register.

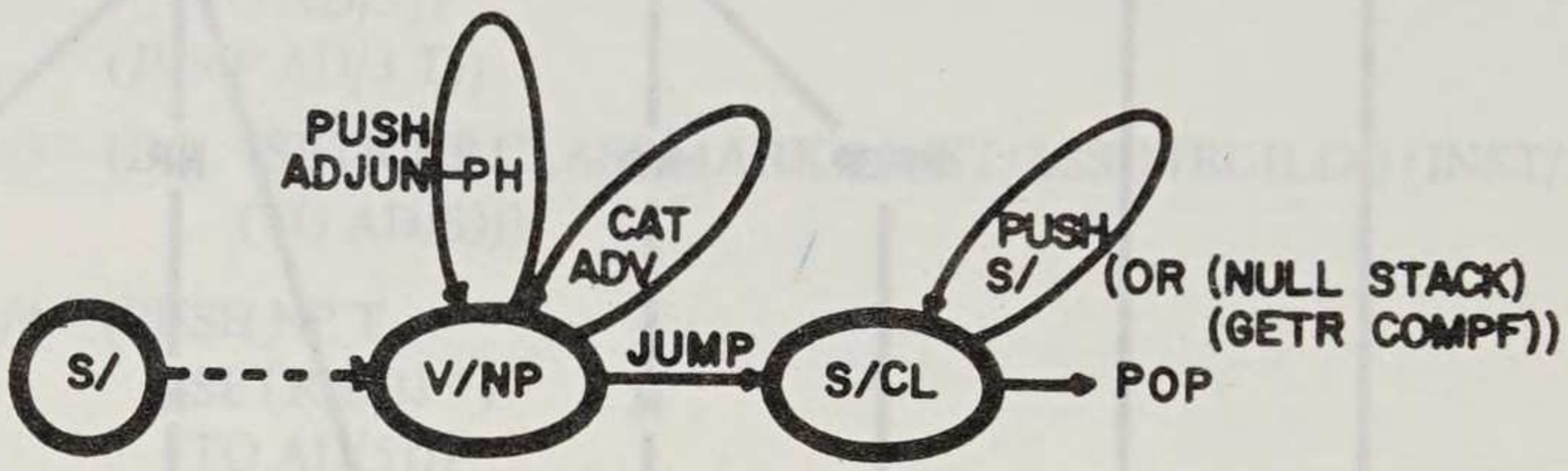


Figure 3.

2.2. **EXPANSION TO ALLOW FOR SUBJECT DELETION.** Subject deletion occurs when independent clauses are strung together, and also after certain clause relators. A mechanism is needed to add the clause relators and to bypass the subject for deletion under specific conditions.

(1) Add the following arcs to state S/. First before the arc (S/ (PUSH NP/ T)) add:

```
(CAT CL-REL T
   (SETR CL-REL *)
   (TO S/CL-REL))
```

This arc adds the new state S/CL-REL to which the PUSH NP/ arc is transferred:

```
(S/CL-REL (PUSH NP/  T
   (SETR SUBJ *) (SETRQ TYPE DECL)
   (TO S/NP)))
```

Second, following the (S/ (CAT CL-REL. . .)) add:

```
(S/   (JUMP S/NP (GETR CL))
      (JUMP S/CL-REL T))
```

The first of these two arcs will bypass the subject if a clause can be found in the register CL and there was no successful transition on the previous arc. In Figure 4 the arcs at

state S/ are numbered to show the order in which they should be tried for transition.

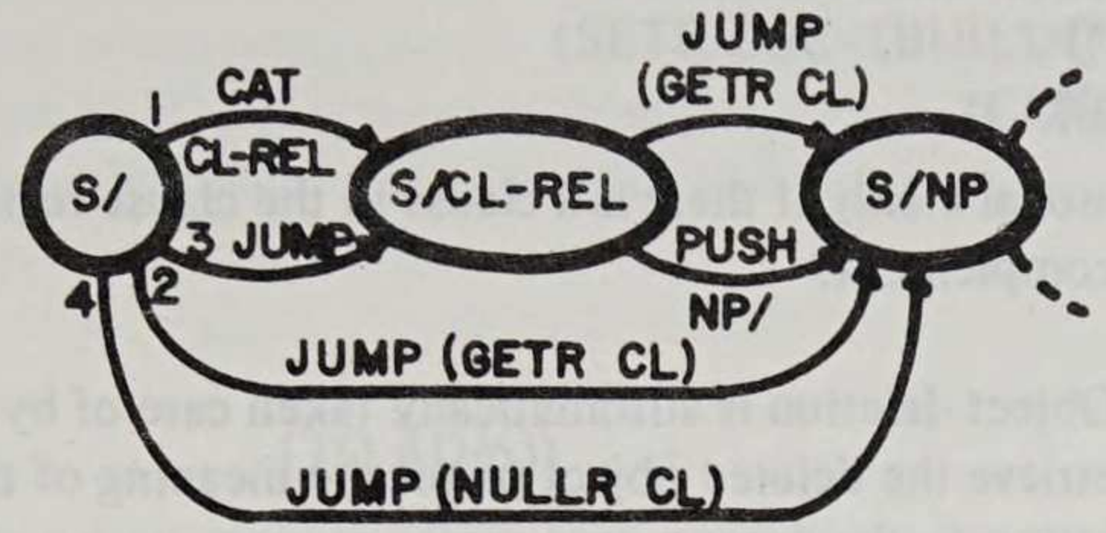


Figure 4.

Third, following the (S/ (JUMP S/NP (GETR CL))) add: (JUMP S/NP (NULLR CL) (SETRQ TYPE IMP)). This arc separates the initial clause of an imperative sentence, which also has no expressed subject, from clauses within a series. Further clauses may be added later to this initial imperative clause through the other JUMP arc that expresses subject deletion in a clause series; but the whole sentence will be imperative because the type register was set to imperative in the first clause. The register TYPE then needs to be added to the final sentence POP arc.

(S/CL (POP. . . (BUILDQ (S + + + +) TYPE SUBJ AUX VP)))

(2) Add the state (S/CL-REL) and to it, the following arcs. First,

(S/CL-REL (PUSH NP T
(SETR SUBJ *)
(TO S/NP))

This arc will allow for a subject after clause relators like **bik fo** 'because', **bo** 'sequence', **ji** 'purpose', **e** 'result', **ā** 'coordinate; and, but', and **à** 'if', which always stands sentence initial. Second, (JUMP S/NP (AND (GETR CL) (WRD (foo fo boo bo e) (GETR CL-REL)))). The condition on this arc allows subject deletion only if there is a clause in the clause register already, and if the clause relator is one of those listed in the condition.

2.3. MODAL a. The preverbal particle **a** occurs only in clauses with deleted subject. Its meaning has to do with an indefinite time limit between the action of the last clause and the action of the current clause. It is equivalent to the preverbal modal **wá** which occurs only with clauses that have expressed subjects: **wu sanri menkak fo a ton voey sho** (He split firewood purpose indefinite burn fire instrumental) 'He splits firewood for making fire (some day)', **wu sanri menkak ji yeelav wá ton voey sho** (He split firewood purpose housewife indefinite burn fire instrumental) 'He splits firewood for the wife to burn (whenever needed)'. To account for this modal **a** we add the following arc to the auxiliary network:

```
(AUX/1 (WRD a (OR (GETR CL)
                  (GETR COMPF))
              (SETR WA *)
              (TO AUX/2)))
```

This arc allows for the use of modal **a** only if there is a clause in the clause register or if the present clause is part of a complement.

2.4. OBJECT DELETION. Object deletion is automatically taken care of by the (S/V (JUMP)) arc; but in order to retrieve the deleted object to get the meaning of the sentence we have to add the following action to the arc:

```
(S/V (JUMP V/NP T
           (COND ((AND (VTRANS V)
                       (GETR OBJ NEAREST))
                  (SETR VP (BUILDQ (VP (V +) DELETED OBJECT: #))
                     V (GETR OBJ NEAREST))))
                     T (SETR VP (BUILDQ (VP (V +)) V)))) . . .).
```

The first part of this action builds a verb phrase consisting of verb plus deleted object. The second part of the action builds a verb phrase consisting of only a verb.

2.5. ANAPHORIC sho. Anaphoric **sho** always refers back to the nearest object other than the object of the clause in which **sho** is found. Its function may be either as an instrumental/associative or as locative depending on the verb of the clause (Section 1.2). **Sho** is entered into the ADJUN-PH network as the first arc leaving the first state and going to the last state of the network. The function of **sho** and its referent are determined by two complex actions. These actions also build the full instrumental/associate phrase or the full locative phrase which underlies **sho** and gives its full meaning.

```
(ADJUN-PH/ (WRD sho T
           (COND ((OR (VA-P V)
                      (VMOTION V))
                  (SETR INST/ASSO
                     (BUILDQ (* = DELETED INST/ASSO (e# #)
                        (COND ((NULLR OBJ)
                                  (CAR (GETR OBJLIST)))
                              (T (CADR (GETR OBJLIST)))))
                        (AND (SETQ U (CAR (GETR OBJLIST)))
                             (COND ((WRD N (CADR U))
                                    (BUILDQ (CLASS-MARKER#)
                                       (CDDR (ASSOC (CDDR U)
                                                    DICTIONARY)))
                                    ))))))
```

```
                    ((NOR  (VA-P  V)
                           (VMOTION  V))
                      (SETR LOC  (BUILDQ    (* = DELETED LOC# )
                                            (COND  ((NULLR OBJ)
                                                    (CAR (GETR OBJLIST)))
                                                   (GETR OBJLIST)))
                                            (T (CADR
                                                   (GETR OBJLIST)))))))
          (TO AD/6)) . . . ).
```

This assumes a dictionary in which each noun has a class marker associated with it:

(WORD (N CLASS)).

These actions are given in the LISP compatible language used by Woods and they explain everything concerning **sho** discussed in Section 1.2.4.

2.6. ANAPHORIC fo. Anaphoric **fo** is always a locative and always refers back to the nearest locative. It is entered in the ADJUN-PH network as the last arc leaving the first state of the network and going to the last state. It is entered as the last arc in order to test any **fo** in the input sentence first for being a preposition on the arc CAT PREP. **Fo** is anaphoric only if no successful transition can be completed on the CAT PREP arc series.

```
(ADJUN-PH/      (WRD fo T
                   (SETR LOC  (APPEND    (LIST *)
                                         (GETR LOC NEAREST)))))).
```

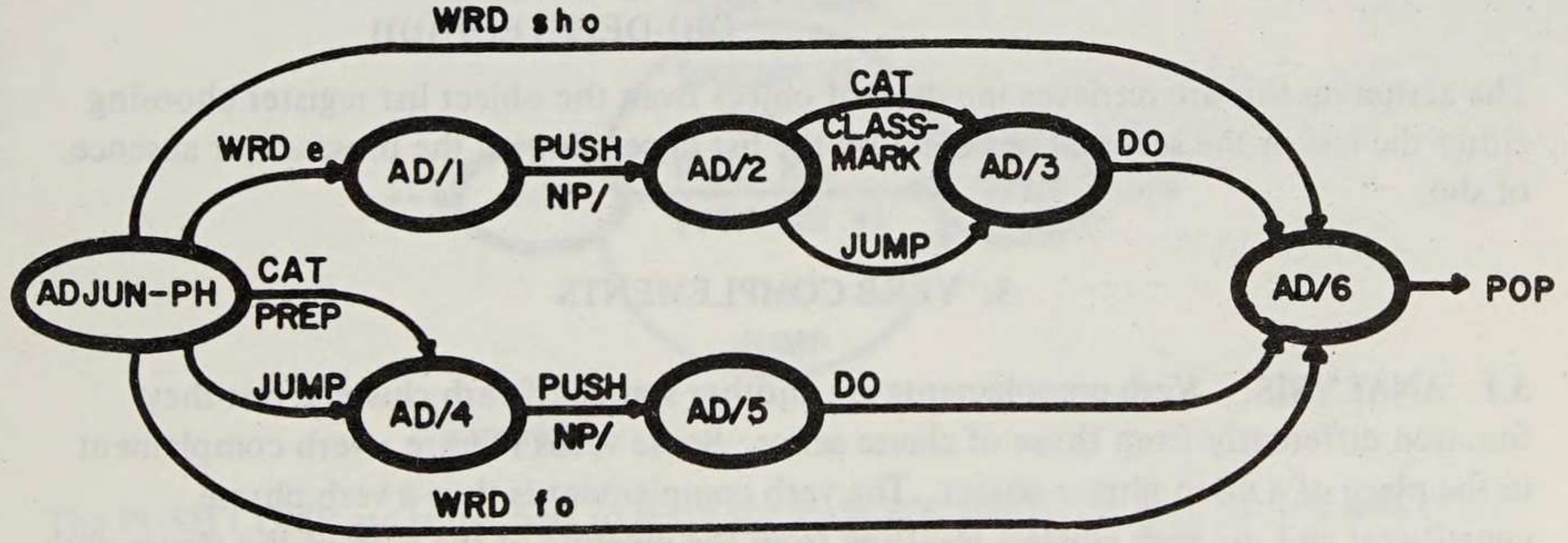


Figure 5. Showing sho and fo in the ADJUN-PH

2.7. **ORDERING OF ANAPHORIC sho AND OBJECT DELETION RULES.** Some mechanism is needed to allow the object deletion rule to look ahead for a possible **sho** next to the verb. If a **sho** is in that position the object deletion rule should jump over the nearest object and look for an object preceding the nearest one. This is accomplished by setting up a OBJLIST register on the (S/V PUSH NP/) arcs and adding a further action to the (S/V JUMP) arc.

```
(S/V  (PUSH NP/ (VTRANS V)
          (SETR VP (BUILDQ  (VP  (V +)  *)  V))
          (SETR OBJ *)
          (ADDL OBJLIST *)
          (TO V/NP))
```

In the object list register all objects of the clauses analyzed are stored.

```
(S/V  (JUMP V/NP  T
        (COND  ((AND   (VTRANS V)
                       (NEXTWRD sho))
                  (SETR OBJ-DELETED
                     (CADR  (GETR OBJLIST NEAREST))))
               (COND  ((AND   (GETR CL)
                              (VTRANS V))
                         (SETR OBJ-DELETED  (GETR OBJ NEAREST))))
               (SETR VP  (BUILDQ  (VP  (V +) #)
                            V  (COND  ((GETR OBJ-DELETED)
                                       (BUILDQ  (DELETED OBJECT +)
                                          OBJ-DELETED)))))))
```

The action on this arc retrieves the deleted object from the object list register choosing either the last or the second last object on the list depending on the presence or absence of **sho**.

3. VERB COMPLEMENTS

3.1. **ANALYSIS.** Verb complements are another source of verb clusters, but they function differently from those of clause series. Some verbs require a verb complement in the place of a noun phrase object. The verb complement is thus a verb phrase constituent and the verb clusters resulting from the meeting of the verb of the clause with the verb of the complement reach across two grammatical levels, clause level and phrase level. The verb complement has the basic construction of complementizer **ji** plus sentence: **wu tak ji à wá wiy** (He want complementizer you indefinite come) 'He wants you to come.' With an expressed subject in the complement the complementizer **ji** must always be used. However, when the subject of the complement is identical with the subject of the clause in which the complement is embedded the subject of the complement is deleted. In that case the complementizer cannot remain **ji** but must change to **fo** or **foo** or nothing, depending on the class of the complement verb.

With one class the complementizer changes to **fo** or **foo** or optionally nothing. The reduced clause of the complement has the same internal structure as the reduced clauses of clause series. It may employ the auxiliary **a** to show indefinite time limit or it may show immediacy with the complementizer **foo** similar to the marking of immediacy in clause series by the clause relator **boo**. **wu kong foo du way** (He like complementizer-immediate go market) 'He likes to go to market (now).' **wu kong fo a du way** (He like complementizer indefinite go market) 'He likes to go to market (sometimes).' **wu kong a du way** (He like indefinite go market) 'He likes to go to market (sometimes).' This complement has no complementizer. **wu kong du way** (He like go market) 'He likes to go to market.' This complement has no complementizer.

With the verb **tak** 'want' a complementizer is never used when the subject is deleted. **wu tak du way** (He want go market) 'He wants to go to market.'

With a third class of complement verbs a complementizer must always be used, **ji** if the subject is expressed and **fo** or **foo** if the subject is deleted. **wu ten ji à du way** (He refuse complementizer you go market) 'He refuses that you should go to the market.' **wu ten fo a du way** (He refuse complementizer indefinite go market) 'He refuses to go to the market (at any time).' Any one of these complements may itself consist of a clause series: **wu ten fo a du yun kifa** (He refuse complementizer indefinite go buy thing) 'He refuses to go and buy something (at any time).'

3.2. **DESCRIPTION.** In Figure 6 the complement is entered in the place of the object by adding an arc to the state S/V.

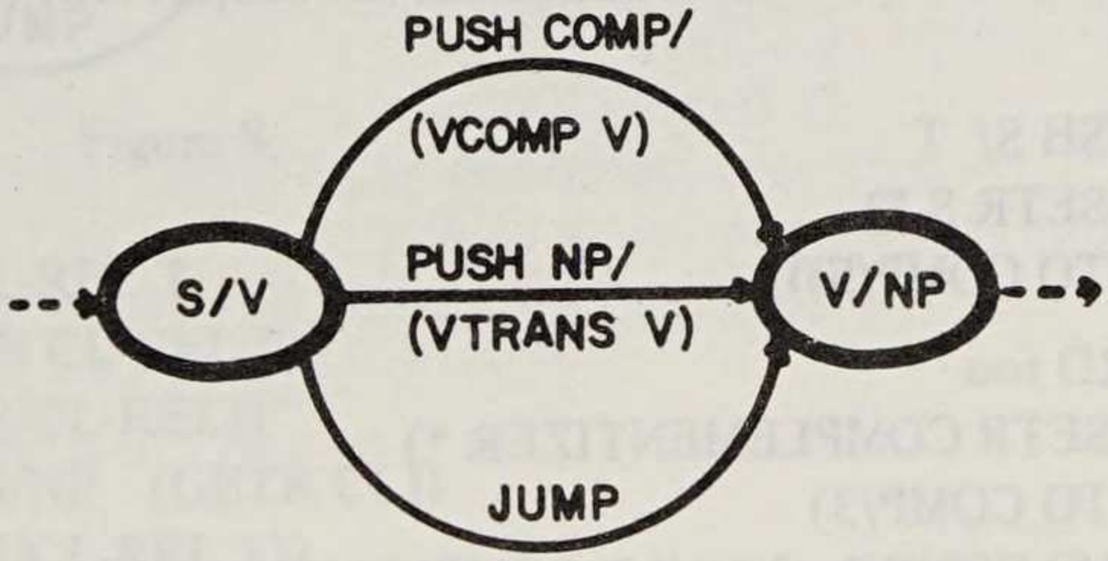


Figure 6.

The PUSH COMP arc is the first to leave the state S/V and it can only be followed if the verb preceding it has the feature COMP.

```
(S/V  (PUSH COMP  (VCOMP V)
        (! (COND  (NOT  (WRD ji)
                        (SENDR SUBJ  (GETR SUBJ))))
        (SENDRQ COMPF T)
        (SETR COMP *)
        (TO V/NP)))
```

The first action sends the subject of the sentence down to the next lower level if the next word in the input string is not **ji**. The second action also sends down the register complement flag (COMPF) for interrogation later on.

Figure 7 shows the diagram of the complement sub-network. The two arcs leaving the initial state describe the two basic kinds of complements, those that consist of a full sentence and those that consist of a sentence reduced by subject deletion.

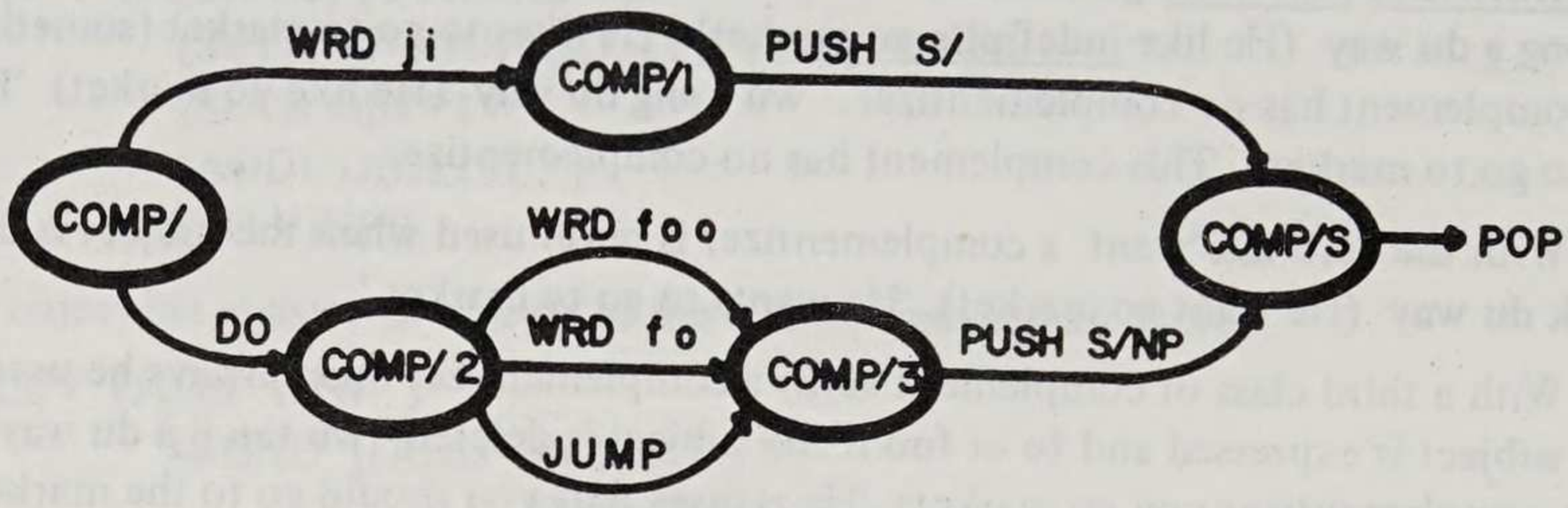


Figure 7.

```
(COMP/ (WRD ji T
            (SETR COMPLEMENTIZER *)
            (TO COMP/1))
       (DO  (SETR SUBJ  (BUILDQ  (DELETED SUBJECT +)  SUBJ))
            (TO COMP/2)))
```

The deleted subject of the complement is built from the subject sent down by the PUSH COMPL arc.

```
(COMP/1   (PUSH S/  T
               (SETR S *)
               (TO COMP/S)))
(COMP/2   (WRD foo
               (SETR COMPLEMENTIZER *)
               (TO COMP/3)
          (JUMP COMP/3  (NUULR COMPLEMENTIZER)))
(COMP/3   (PUSH S/NP  T
               (SENDR SUBJ  (GETR SUBJ))
               (SETR S *)
               (TO COMP/S)))
```

The PUSH S/NP arc pushes for a sentence starting at state S/NP. The subject of that reduced sentence is supplied by the action (SENDR SUBJ). When the sentence is completed the POP arc of the sentence network will POP up the full sentence into the complement network where it is stored in the register sentence (SETR S *). The arc PUSH S/ does the same except that it will push for a full sentence. The condition

(GETR COMPF) was added to the (S/CL PUSH S/) arc so that the push can be executed without returning first to the top level.

(COMP/S (POP (BUILDQ (COMP++) COMPLEMENTIZER S) T))

This arc returns the completed complement structure to the sentence network.

SUMMARY OF THE GRAMMAR

Figure 8 is the revised S/ network. It goes together with Figures 5 and 7 and the NP/ and AUX/ portions of Figure 1 to represent the complete grammar that has been worked out so far.

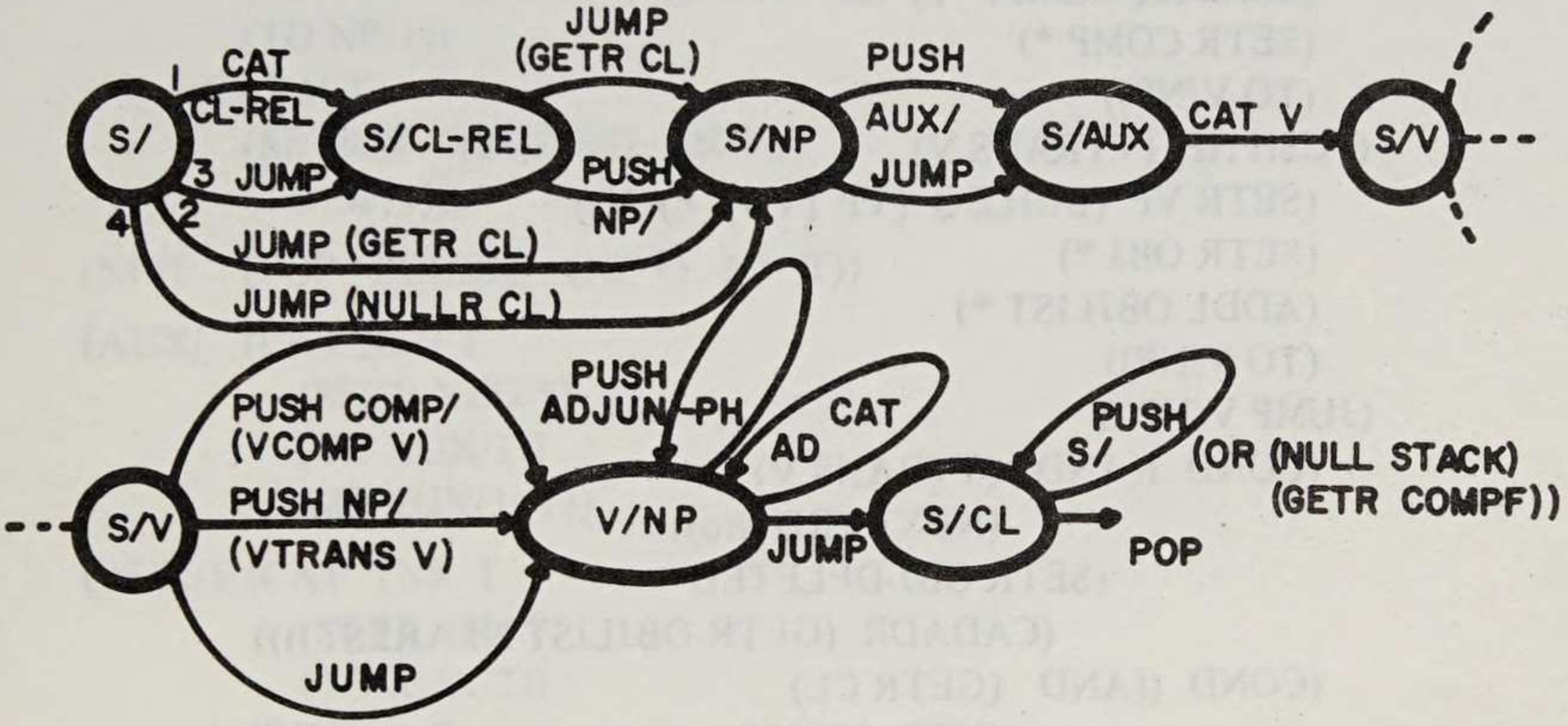


Figure 8.

```
(S/    (CAT CL-REL T
          (SETR CL-REL *)
          (TO S/CL-REL))
       (JUMP S/NP    (GETR CL))
       (JUMP S/CL-REL T))
       (JUMP S/NP    (NULLR CL)
          (SETRQ TYPE IMP)))
(S/CL-REL (JUMP S/NP  (AND (GETR CL)
                           (WRD (foo fo boo bo e))
                           (GETR CL-REL)))
             (PUSH NP/  T
                (SETR SUBJ *)
                (TO S/NP)))
```

```
(S/NP   (PUSH AUX/  T
            (SETR AUX *)
            (TO S/AUX))
        (JUMP S/AUX T))
(S/AUX (CAT V T
            (SETR V *)
            (TO S/V)))
(S/V  (PUSH COMP/    (VCOMP V)
        (! COND   ((NOT  (WRD ji))
                         (SENDR SUBJ  (GETR SUBJ))))
        (SENDRQ COMPF  T)
        (SETR COMP *)
        (TO V/NP))
    (PUSH NP/ (VTRANS V)
        (SETR VP  (BUILDQ  (VP  (V +)  *)  V))
        (SETR OBJ *)
        (ADDL OBJLIST *)
        (TO V/NP))
    (JUMP V/NP  T
        (COND  ((AND   (VTRANS V)
                       (NEXTWRD sho))
                (SETR OBJ-DELETED
                    (CADADR  (GETR OBJLIST NEAREST))))
        (COND  ((AND   (GETR CL)
                       (VTRANS V))
                (SETR OBJ-DELETED  (GETR OBJ NEAREST))))
        (SETR VP  (BUILDQ  (VP  (V +) #)
                    (V  (COND  ((GETR OBJ-DELETED)
                               (BUILDQ  (DELETED OBJECT +)
                                 OBJ-DELETED))) ))))
(V/NP   (PUSH ADJUN-PH  T
            (SETR VP  (APPEND  (GETR VP)  *))
            (TO V/NP))
        (CAT ADV T
            (SETR VP  (APPEND  (GETR VP)
                               (BUILDQ  (ADV *)))))
        (JUMP S/CL T
            (COND  ((NULL STACK)
                    (SETR CL (BUILDQ  (CL + + + +)
                                      (TYPE SUBJ AUX VP))))))
```

```
(S/CL   (PUSH S/  (OR  (NULL STACK)
                       (GETR COMPF))
          (SENDR SUBJ  (BUILDQ  (DELETED SUBJECT +)  SUBJ))
          (ADDR CL *)
          (TO S/CL))
        (POP  (COND  ((NULL STACK)
                      (CONS  (QUOTE S)  GETR CL)))
                     (T (BUILDQ  (CL + + + +)
                         TYPE SUBJ AUX VP))) T))

(NP/  (CAT PRON  T
        (SETR N   (BUILDQ   (PRON *)))
        (TO NP/1))
      (CAT N T
        (SETR N   (BUILDQ   (N *)))
        (TO NP/1)))
(NP/1   (POP  BUILDQ  (NP +)  N)  T))
(AUX/   (CAT NEG T
          (SETR NEG *)
          (TO AUX/1))
        (JUMP AUX/1  T))
(AUX/1 (CAT  TNS  T
          (SETR TNS *)
          (TO AUX/2))
        (WRD wa T
          (SETR TNS *)
          (TO AUX/2))
        (WRD a  (OR  (GETR CL)
                     (GETR COMPF))
          (SETR WA *)
          (TO AUX/2)))
(AUX/2  (POP  (BUILDQ  (AUX  +  +  +)  NEG TNS WA)  T))
(ADJUN-PH/  (WRD sho  T
              (COND  ((OR (VA-P V)
                          (VMOTION V))
                      (SETR INST/ASSO
                        (BUILDQ   (* = DELETED INST/ASSO  (e# #))
                          (COND  ((NULLR OBJ)
                                  (CAR  (GETR OBJLIST)))
                                 (T (CADR  (GETR OBJLIST))))
                          (AND    (SETQ U  (CAR  (GETR OBJLIST)))
```

```
                                (COND ((WRD N (CADR U))
                                       (BUILDQ (CLASS-MARKER #)
                                               (CDDR (ASSOC (CDDR U)
                                                            DICTIONARY
                                                            )))))))
                        ((NOR (VA-P V)
                              (VMOTION V))
                         (SETR LOC (BUILDQ (* = DELETED LOC #)
                                           (COND ((NULLR OBJ)
                                                  (CAR (GETR
                                                        OBJLIST)))
                                                 (T (CADR (GETR
                                                           OBJLIST)))))))
                            (TO AD/6))
                         (WRD e (OR (VA-P V)
                                    (VMOTION V))
                            (TO AD/1))
                         (CAT PREP T
                            (SETR PREP *)
                            (TO AD/4))
                         (JUMP AD/4 T)
                         (WRD fo T
                            (SETR LOC (APPEND (LIST *)
                                              (GETR LOC NEAREST)))
                            (TO AD/6)))
(AD/1 (PUSH NP/T
         (SETR OBJ *)
         (TO AD/2)))
(AD/2 (CAT CLASS-MARK T
         (SETR CLASS-MARK *)
         (TO AD/3))
      (JUMP AD/3 T))
(AD/3 (DO (SETR INST/ASSO (BUILDQ (INST/ASSO (e + +))
                                  NP CLASS-MARK))
          (TO AD/6)))
(AD/4 (PUSH NP/ T
         (SETR OBJ *)
         (TO AD/5)))
(AD/5 (DO (SETR LOC (BUILDQ (LOC # + )
                            (COND ((GETR PREP)
```

```
                                        (BUILDQ (PREP +) PREP))
                                    (T NIL))
                                OBJ))
                    (TO AD/6)))
(AD/6   (POP (BUILDQ  (ADJUN-PH + +) INST/ASSO LOC) T))
(COMP/ (WRD ji T
            (SETR COMPLEMENTIZER *)
            (TO COMP/1))
        (DO (SETR SUBJ (BUILDQ (DELETED SUBJECT +) SUBJ))
            (TO COMP/2)))
(COMP/1   (PUSH S/ T
            (SETR S *)
            (TO COMP/S)))
(COMP/2   (WRD foo T
            (SETR COMPLEMENTIZER *)
            (TO COMP/3))
          (WRD fo T
            (SETR COMPLEMENTIZER *)
            (TO COMP/3))
          (JUMP COMP/3 (NULLR COMPLEMENTIZER)))
(COMP/3   (PUSH S/NP T
            (SENDR SUBJ (GETR SUBJ))
            (SETR S *)
            (TO COMP/S)))
(COMP/S   (POP (BUILDQ (COMP + +) COMPLEMENTIZER S) T))
```

CHAPTER 5

TRANSITION NETWORK GRAMMARS AS A NOTATION SCHEME FOR TAGMEMICS

William M. Leal

1. INTRODUCTION. Tagmemic theory is not defined in terms of any particular notational scheme. Although one scheme which has certain pedagogical and heuristic advantages has been almost universally used, there are in principle a large number of notational schemes that are compatible with the basic premises of tagmemics. This chapter explores the use of a system developed by W. A. Woods (1970) as a notational scheme for tagmemics. I intend to show that networks can be advantageously used to represent the display grammar system ("9 box tagmemics") developed recently by Pike and Hale (Hale 1973a,b). The formalism used in this paper for networks differs somewhat from Woods's formalism, but the underlying systems are compatible.[1]

---

[1] I am indebted to Woods for the underlying concepts of transition network grammars. I have attempted, however, to improve on the formalism he uses. Here is a brief sketch of the differences between Woods's system and the one presented here.

One of Woods's main goals was to provide a means of communicating with a computer via natural English. Since this entailed developing a computer processable grammar of English, TNG was the result. Woods (1970) presents a number of ways in which his effort is, from a theoretical standpoint, preferable to what has been done previously. Despite this theoretical superiority, however, the main goal was still pragmatic, and this is reflected in the formalism which he adopted.

Woods's TNG formalism is based on LISP, a computer programming language designed for natural language research. Without going into all the details, LISP uses prefix operators. My formalism uses infix operators, with which we might represent a form as (–.A & .B) / .C, Woods's system would represent the same form as (OR (AND (NOT A) B) C), using prefix operators; but embeddings increase, so do the number of parentheses until it may become unclear to a reader just what is being done. The infix system with delimiters (colon, semicolon, comma, etc.) helps eliminate many parentheses to yield more readable grammars.

One main advantage of LISP notation is that it can be interpreted by a computer without the translation required by an infix system. This is not, however, relevant to TNG as such. Avoiding the translation simply makes it easier for a person to implement the system on a particular computer. It does not affect what can be done in TNG, nor does it make it easier for the linguist, to whom ease of use is a major consideration.

Woods's TNG for English includes a large number of primitive forms. Woods (1970:594 footnote) in fact states that no attempt has been made to reduce primitives to a minimum, but rather primitives were added to the system as needed. Since these primitives are specified in LISP, a linguist wishing to add his own primitives to a system tailored to some other natural language would ahve to do so in LISP. By having a subroutine component defined in TNG, the linguist need not master the details of both TNG and LISP.

Certain shortcomings in the traditional tagmemic scheme (Elson and Pickett) point to faults in the notation rather than to faults in the basic premises. In the tranditional scheme it is difficult, for example, to represent optional ordering of constituents, especially where the possible set of permutations is complex. In such cases, statements on ordering are usually omitted from the formulae and stated in prose. Since the prose is not formally constrained, it can be inexact. Similarly, where there is a set of optional tagmemes within a syntagmeme, with each tagmeme optional but with the condition that at least one of the tagmemes of the set is to be chosen, the statement of this relation is generally committed to prose. Prose is also resorted to when there are complex dependencies among optional items that do not form parts of a single continuous constituent in the syntagmeme.

Consider the matter of ordering in Sherpa. Schöttelndreyer makes the following statements: 'The topicalization of almost any tagmeme of a clause in Sherpa can be done simply by fronting.' (1972:43) 'The normal order of roles in Sherpa . . . can be changed according to the following rule: *animate* roles generally will *precede inanimate* roles.' (1972:44) These statements are unclear because it is not explicitly stated what the order is in the case of topicalization (fronting for emphasis) when there are three roles. That is, if to show topicalization, a particular role were fronted from its normal order, would the succeeding roles also be in the animate-inanimate order, or would they be in normal order, or would they follow any particular order at all? Then, in the case of animate fronting in which there are either two animate roles or two inanimate roles, are these roles ordered in any way or not? These questions cannot be answered by the information given, and hence the specification, though superficially complete, does not quite work. One purpose of this paper is to present a notation scheme which forces the linguist to answer questions of this sort.[2]

2. SOME CHARACTERISTICS OF TAGMEMICS.

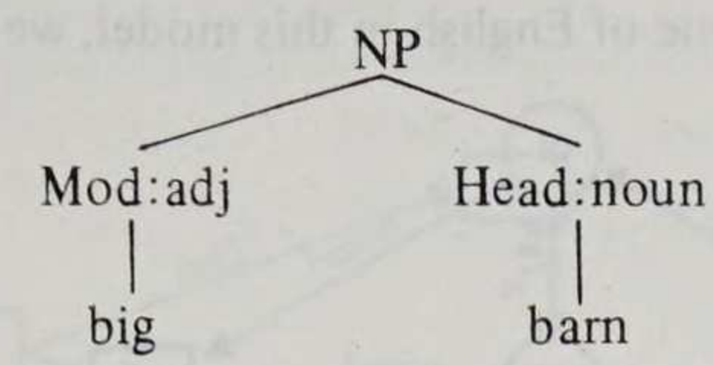
2.1. TAGMEMICS AS TRANSDUCER AND ANALYZER. Consider the following partial tagmemic representation for the English noun phrase:

NP = ± Mod:adj + Head:noun.

This syntagmeme states that a noun phrase consists of a noun preceded by an optional adjective. The formula is thus a transducer, showing what forms a noun phrase may take. The Mod and Head labels provide an analysis of the noun phrase. For the noun phrase *big barn*, the resulting analysis is:

Finally, related to the above point, Woods's morphological analyzer is implemented in LISP, and is very specific to English. The morphological analysis system presented here attempts to be more general. Although there are a number of as yet unresolved problems in morphological analysis, particularly those related to morphophonemic reduction, the mechanism presented here at least makes an attempt to face the problems.

[2]For details regarding this notational system, see Hale 1973.



Tagmemics thus transduces as well as analyzes.

2.2. TAGMEMIC ORDERING. In Spanish a noun phrase can have a modifier either preceding or following the head. Then in the traditional notation, we would write:

$$NP = \pm \overset{\overset{\pm}{\frown}}{\text{Mod:adj}} + \text{Head:noun} \mp \text{Mod:adj}$$

Here the line above the syntagmeme shows that the items so marked are related. That is, when the first is present, the second must be absent and vice versa; or else neither need be present as indicated by the ± on the line itself.

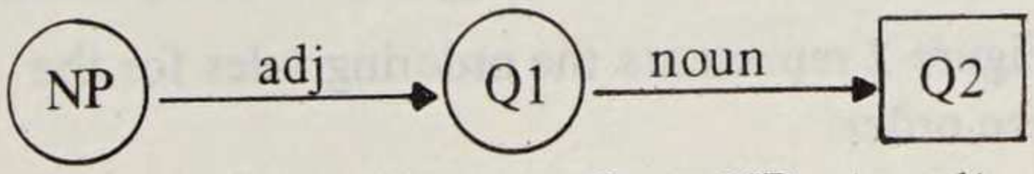
The NP given for English is by no means complete. There may, for instance, be an optional determiner:

1) NP = ± Det:art ± Mod:adj + Head:noun

Now suppose for some other language that the determiner, modifier, and head could appear in any order, as long as the determiner preceded the modifier if the two were contiguous. Traditional tagmemic notation has no way of representing this rule conveniently, and it would probably be stated in prose. The alternative is to give three syntagmemes showing the possible orderings. This obscures the regularity which has been noted. (It is perhaps unlikely that any natural language would have such a noun phrase, but such phenomena as ordering of clause constituents are often more complex even than this.)

3. TRANSITION GRAPHS AS A TAGMEMIC REPRESENTATION

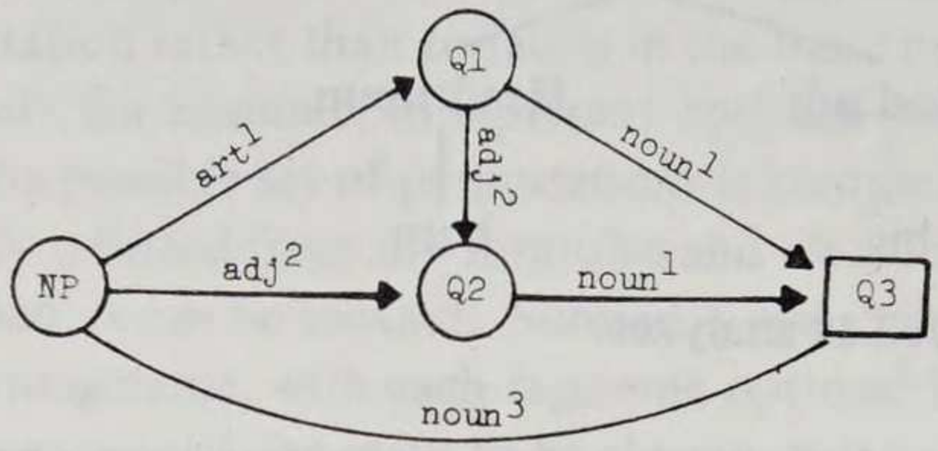
One of the earlier models for natural language grammars was the finite state transition graph (Chomsky 1957). This consists of a set of labeled **states** connected by sets of labeled **arcs**, which permit the grammar to move from state to state. Each arc is labeled with a symbol which permits the grammar to move between the two states it connects. Suppose that, in English, a modifier were obligatory in a noun phrase. Then a finite state transition graph for the noun phrase would be:



To interpret this, we begin at State NP. An adjective in the phrase being transduced allows a transition to State Q1. From here, a noun in the phrase allows a transition to State Q2. The square box around State Q2 means that that node is **terminal**. That is, when we reach that node, the phrase is finished, and the transduction is successful.

To represent a syntagmeme of English in this model, we may write it as follows:

2)



The arcs are numbered to permit reference to them and to show ordering. The graph may be interpreted as follows: If the phrase begins with an article, then transfer to State Q1 via Arc NP:1. If it begins with an adjective, then transfer to State Q2 via Arc NP:2. If it begins with a noun, then transfer to State Q3 via Arc NP:3.

Suppose it begins with an article. Then transfer to State Q1, via Arc NP:1. From State Q1, either a noun or an adjective may follow. This is shown by Arcs Q1:1 and Q1:2. Suppose an adjective follows. Then transfer to State Q2 via Arc Q1:2. From State Q2, only one arc leads out. This specifies a noun. By this arc, a noun must appear next. This being the case, State Q3 is reached. The square box around State Q3 indicates that this is a final state. If there are no more words in the phrase, then the transduction was successful. We can also say that the graph succeeds.

For example, consider the noun phrase *the big elephant;* Begin at State NP. The first word is *the*, an article. Transfer therefore to State Q1. The next word is *big*, an adjective. Transfer to State Q2. The next word is *elephant*, a noun. Transfer to State Q3. State Q3 is a final state and there are no more words in the phrase, so the graph succeeds. When transferring from one state to another via an arc, we say that the arc was **travelled**, or **traversed**. The arcs travelled in the course of transducing a particular input comprise the **path of transduction**; in the preceding example, the path of transduction consisted of Arcs NP:1, Q1:2, Q2:2. For a noun phrase such as *the boy*, the path consists of Arcs NP:1, Q1:1. Finally, the **evaluation** of a graph for a given input means that transduction is attempted for that input. If the graph fails, the input is ungrammatical and we say that the evaluation was **unsuccessful**. If, however, the graph accepts the input, then the evaluation was **successful**. We may also speak of the path described by an evaluation.

The reader may wish to evaluate the following phrases for English syntagmeme 1, noting that it fails on the fourth and sixth examples: man, carniverous animals, that article, that article of the linguist, that horrible article, the big red nose, my last example.

The network diagrammed in Figure 2 represents the ordering rules for the hypothetical language with nearly free order:

3)

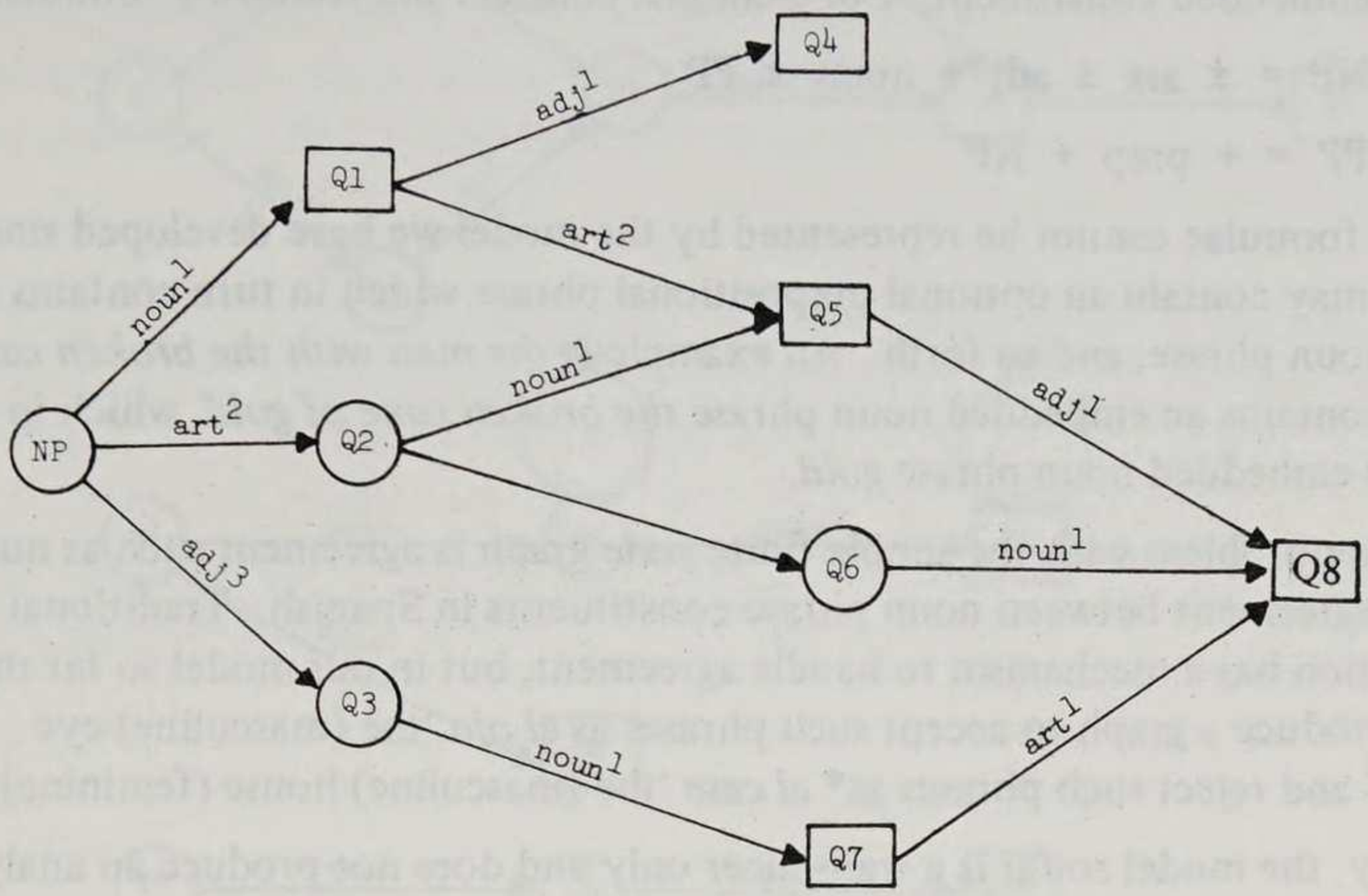


Figure 2

The kinds of phrases transduced by this graph are listed below with the paths followed:

4)	(a)	barn	NP:1
	(b)	big barn	NP:3, Q3:1
	(c)	barn big	NP:1, Q1:1
	(d)	the barn	NP:2, Q2:2
	(e)	barn the	NP:1, Q1:2
	(f)	the big barn	NP:2, Q2:2, Q6:1
	(g)	the barn big	NP:2, Q2:1, Q5:1
	(h)	barn the big	NP:1, Q1:2, Q5:1
	(i)	big barn the	NP:3, Q3:1, Q7:1

Note that such phrases as *big the barn* are not transduced by this graph.

This finite state model is, however, not enough, because it fails to capture other regularities of natural language. For one thing, there is no mechanism which permits the

evaluation of the graph to be suspended while the same (or another) graph is used to evaluate an embedded constituent. For example, consider the following formulae:

NP = ± art ± adj + noun ± PP

PP = + prep + NP

These formulae cannot be represented by the model we have developed since a noun phrase here may contain an optional prepositional phrase which in turn contains an embedded noun phrase, and so forth. An example is *the man with the broken cane of gold*. This contains an embedded noun phrase *the broken cane of gold*, which in turn contains the embedded noun phrase *gold*.

Another problem with the simple finite state graph is agreement such as number and gender agreement between noun phrase constituents in Spanish. Traditional tagmemic notation has a mechanism to handle agreement, but in our model so far there is no way to produce a graph to accept such phrases as *el ojo* 'the (masculine) eye (masculine)' and reject such phrases as* *el casa* 'the (masculine) house (feminine)'.

Finally, the model so far is a transducer only and does not produce an analysis as is produced by tranditional tagmemic notation.

4. RECURSIVE TRANSITION NETWORKS AS A TAGMEMIC REPRESENTATION

The transition graph was shelved for many years as a model for natural language for the objections cited above, among others. Chomsky (1957) shows specifically why it is inadequate. More recently, Woods (1970) has resurrected the model by adding features to it which increase its power so that such things as embedding and agreement may be included. In addition, the revised model can produce an analysis.

4.1. RECURSION. To begin with, Woods added recursion. Suppose we have a collection of **transition networks** (the term Woods uses; hereafter, 'graph' will have a more limited usage which will be explained later). Each network has a **name**. For Figure 2 the name NP. We permit as labels on the arcs not merely terminal category symbols such as **noun**, **art**, or **adj**, but also other network names.

When, in the course of evaluation, a network name is encountered on an arc, then that network is evaluated, while the evaluation of the first network is suspended. When the second network has finished being evaluated, evaluation resumes on the original network. For example, Figure 3 is adapted from Woods (1970):

5)

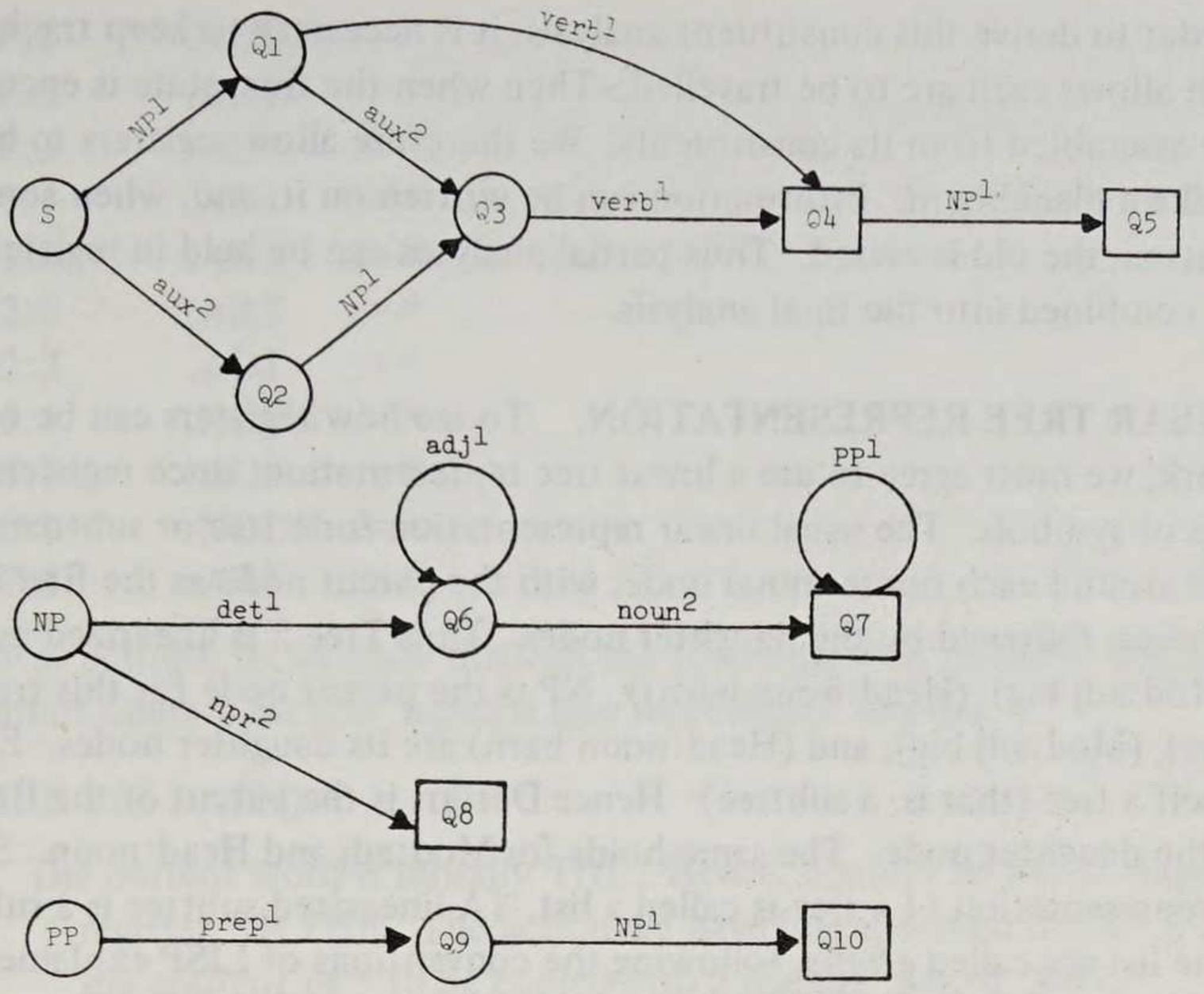


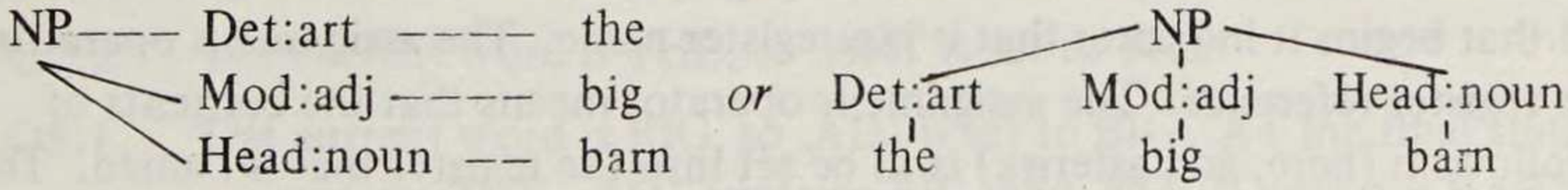
Figure 3

This network transduces a small subset of English. Some examples, with the paths followed, are given below. Parentheses are used to indicate embeddings.

Did John go?
S:2 Q2:1 (NP:2) Q3:1

John did break the big expensive window in the red Cadillac.
S:1 (NP:2) Q1:2 Q3:1 Q4:1 (NP:1 Q6:1 Q6:1 Q6:2 Q7:1
(PP:1 Q9:1 (NP:1 Q6:1 Q6:2)))

4.2. REGISTERS. Suppose that, no matter what the actual order in a particular noun phrase, we wished for Figure 2 to produce a tree in which the article, if present, would always be presented as the first node of the tree, followed by an adjective (if present), followed by a noun. Then the constituent analyses for Examples 4f through 4i would all be the same, being the tree shown below.



(The kind of graphic representation for a tree given on the left is more convenient to write and corresponds exactly to the more traditional representation shown to the right; Frantz (1970) and Grimes (1972)).

In order to derive this constituent analysis, it is necessary to keep track of the actual word which allows each arc to be travelled. Then when the final state is encountered, the tree can be assembled from its constituents. We therefore allow registers to be used. A **register** is like a blackboard. Information can be written on it, and, when something new is to be written, the old is erased. Thus partial analyses can be held in registers, and, at a final state, combined into the final analysis.

4.2.1. LINEAR TREE REPRESENTATION. To see how registers can be operated on by a network, we must agree to use a linear tree representation, since registers can hold only strings of symbols. The usual linear representation for a tree or subtree uses parentheses around each nonterminal node, with the parent node as the first item between the parentheses, followed by the daughter nodes. Thus Tree 5 is linearized as (NP (Det: art the) (Mod:adj big) (Head:noun barn)). NP is the parent node for this tree, and (Det:art the), (Mod:adj big), and (Head:noun barn) are its daughter nodes. Each daughter of NP is itself a tree (that is, a subtree). Hence Det:art is the parent of the first subtree and the is the daughter node. The same holds for Mod:adj and Head:noun. Such a linearized representation of a tree is called a **list**. (A linearized subtree is a **sublist**.) The words in the list are called *atoms*, following the conventions of LISP explained in the appendix. NP, Det:art, the, Mod:adj, big, Head:noun, barn are each atoms.

4.2.2. OPERATIONS ON REGISTERS. As we have seen above, a register can hold a list. It can also hold just an atom; that is, a word with no parentheses around it. Registers can be **set**. That is, their contents can be changed. This is equivalent to erasing a blackboard and writing something new on it. One register can also be **copied** into another. This is equivalent to erasing a blackboard and copying onto it the contents of another blackboard, thus giving two blackboards with the same information on each.

Registers and the manipulations performed on them are associated with the arcs of a network. When a particular arc is travelled in a network, various registers can be set or copied to produce a partial record of the analysis. Consider again Figure 2. If in the course of evaluation, Arc NP:1 is travelled, then we might wish to set some register to contain the word which allowed the arc to be travelled. We can do this by writing as follows:

NP:1 .NOUN := *

The register we have called .NOUN. We might have used any other combination of letters and numbers to make up a name for the register, but .NOUN is easy to remember. The period that begins it indicates that it is a register name. The **assignment operator** := follows the **register reference**. The assignment operator means that the contents of whatever follows it (here, and asterisk) is to be set into the register that is named. The asterisk is a special kind of register called a **distinguished register**. Its value (that is, its contents) is always whatever the current word happens to be. So the **action** shown above causes register .NOUN to be set to the current word. Let us fully specify the actions associated with each arc in Figure 2:

```
NP:1      .NOUN  := *
NP:2      .ART   := *
NP:3      .ADJ   := *
Q1:1      .ADJ   := *
Q1:2      .ART   := *
Q2:1      .ART   := *
Q2:2      .ADJ   := *
Q3:1      .NOUN  := *
Q5:1      .ADJ   := *
Q6:1      .NOUN  := *
Q7:1      .ART   := *
```

Given these actions, let us trace through the following four phrases. Since transition network grammars operate on lists, input is also represented as a list.

7) (a) (THE BIG BARN)

NP:2 The current word is initially THE. Hence, content of the distinguished register * is THE. The action we have just specified for NP:2 causes the content of * to be copied into a register named .ART.

Q2:2 The current word is now BIG, so * contains BIG. The action on this arc causes BIG to be put into .ADJ. Now 'ART contains THE and .ADJ contains BIG.

Q6:1 The current word is BARN. Hence .NOUN is set to BARN. At the final state, Q8, .ART contains THE, 'ADJ contains BIG, and .NOUN contains BARN.

(b) (THE BARN BIG)

NP:2 The current word is THE, so .ART is set to THE.

Q2:1 The current word is BARN, so .NOUN is set to BARN.

Q5:1 The current word is BIG, so 'ADJ is set to BIG. At the final state, Q8, .ART contains THE, .ADJ contains BIG, and .NOUN contains BARN, just as in the first example.

(c) (BARN THE BIG)

NP:1 The current word is BARN, so .NOUN is set to BARN.

Q1:2 The current word is THE, so .ART is set to THE.

Q5:1 The current word is BIG, so .ADJ is set to BIG. AT the final state .ART contains THE, .ADJ contains BIG, and .NOUN contains BARN as before.

(d) (BIG BARN THE)

NP:3 The current word is BIG, so .ADJ is set to BIG.

Q3:1 The current word is BARN, so .NOUN is set to BARN.

Q7:1 The current word is THE, so .ART is set to THE. At the final state .ART contains THE, .ADJ contains BIG, and .NOUN contains BARN.

Even with four permitted orders of BARN, THE, and BIG, the registers .NOUN, .ART and .ADJ are set identically by the time State Q8 is reached. The reader may wish to trace through Examples 4 a-e to see how the registers are set in evaluating the network.

4.2.3. REPORTING THE ANALYSIS. We now need some way to report the information that has been collected in the registers. There are several ways of doing this. The most powerful is the function BUILD. Here is an example:

```
BUILD ('(NP (DET:ART +) (MOD:ADJ +) (HEAD:NOUN +))',
       .ART, .ADJ, .NOUN)
```

BUILD takes several **arguments**, each set off by commas. The arguments for the instance of BUILD shown here are first the skeleton tree '(NP (DET:ART +) (MOD: ADJ +) (HEAD:NOUN +))', second the register name .ART, third .ADJ, and fourth .NOUN.

BUILD operates on its arguments as follows: tracing through the value of the first argument– '(NP (DET:ART +) (MOD:ADJ +) (HEAD:NOUN +))' –it locates each instance of the atom +. The first instance of + is deleted and the value of the second argument is substituted for it. The same thing happens for the second instance of + and the third argument, and so forth.

The first argument is called the **skeleton** of BUILD, since evaluation of BUILD results in its being fleshed out with the various register values.

In the example the first instance of + (following DET:ART) is replaced by the value of the second argument, .ART. For the inputs shown in Example 7, the value of .ART at the final state is THE. So THE is substituted for +. The value of BUILD at this point is (NP (DET:ART THE) (MOD:ADJ +) (HEAD:NOUN +)).

Continuing, the value of the third argument is substituted for the second instance of +. .ADJ is the third argument and its value is BIG. The value of the fourth argument is then substituted for the third instance of +. The fourth argument is .NOUN and its value is BARN. The value of BUILD at this point is (NP (DET:ART THE) (MOD:ADJ BIG) (HEAD:NOUN BARN)), corresponding to Tree 6.

Notice again that it does not matter which of the four possible paths leading to State Q8 were followed: the resultant tree defined by BUILD is the same.

The final states for Figure 2 are shown below along with the BUILDs associated with each. The reader may wish to use the results of tracing through the network for Example 4 a-e to see what trees are specified by these BUILDs.

```
Q1    BUILD (' (NP  (HEAD:NOUN +))', .NOUN)
Q4    BUILD (' (NP  (DET:ART +)    (HEAD:NOUN +))', .ADJ, .NOUN)
Q5    BUILD (' (NP  (DET:ART +)    (HEAD:NOUN +))', .ART, .NOUN)
Q7    BUILD (' (NP  (MOD:ADJ +)    (HEAD:NOUN +))', .ADJ, .NOUN)
Q8    BUILD (' (NP  (DET:ART +)    (MOD:ADJ +)  (HEAD:NOUN +))',
                .ART, .ADJ, .NOUN)
```

Note that these final state BUILDs have numerous common elements (such as (HEAD:NOUN +) which appears in each BUILD), and in fact Q4 and Q7 are identical. It is possible to combine these in the full model we are developing, but at this stage they remain separate.

4.3. DICTIONARIES. Up to now we have assumed that a dictionary exists in which words such as BIG and BARN can be looked up to determine their grammatical categories of adjective and noun. The dictionary also gives us the basis for handling agreement.

Let us first establish a small, simple model dictionary for Spanish. Each entry will be in the following form: (definiens (category FEATURES (NUM number GEN gender))). The dictionary is:

```
(CASA     (NOUN FEATURES  (NUM SG GEN FEM)))
(CASAS    (NOUN FEATURES  (NUM PL GEN FEM)))
(EL     (ART FEATURES  (NUM SG GEN MASC)))
(LA     (ART FEATURES  (NUM SG GEN MASC)))
(LAS    (ART FEATURES  (NUM PL GEN FEM)))
(LOS    (ART FEATURES   (NUM PL GEN MASC)))
(OJO    (NOUN FEATURES  (NUM SG GEN MASC)))
(OJOS   (NOUN FEATURES  (NUM PL GEN MASC)))
(ROJA     (ADJ FEATURES  (NUM SG GEN FEM)))
(ROJAS    (ADJ FEATURES  (NUM PL GEN FEM)))
(ROJO     (ADJ FEATURES  (NUM SG GEN MASC)))
(ROJOS    (ADJ FEATURES  (NUM PL GEN MASC)))
```

Now consider the following network for Spanish (assuming for simplicity at this stage that adjectives only follow the noun head):

9)

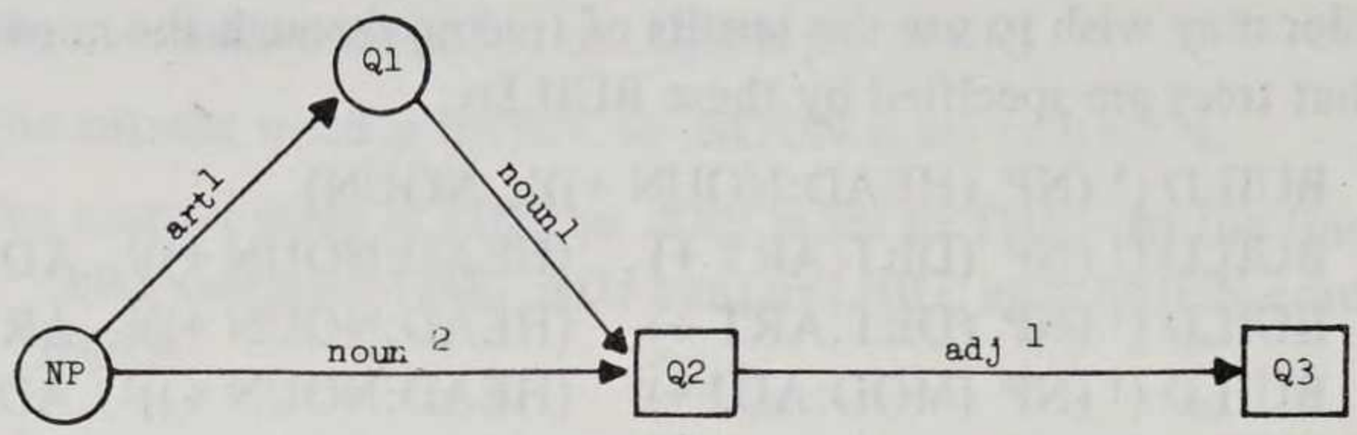


Figure 4

This network will transduce such noun phrases as (LA CASA ROJA), (OJOS ROJOS), etc. It will also transduce such noun phrases as (EL CASA ROJOS), which exhibits disagreement between grammatical number and gender. To avoid such erroneous transductions as this, we will add conditions to final states. If the condition is satisfied, and if the noun phrase is at an end, then the final state is accepted. Otherwise the network fails. These conditions for Figure 4 will be shown, but first let us add the following arc actions:

```
NP:1    .ART := BUILD(' (DET:ART *) ')
        .ARTGEN := GETF ('GEN')
        .ARTNUM := GETF ('NUM')

NP:2    .NOUN := *
        .NOUNGEN := GETF ('GEN')
        .NOUNNUM := GETF ('NUM')
        .ARTGEN := .NOUNGEN
        .ARTNUM := .NOUNNUM

Q1:1    .NOUN := *
        .NOUNGEN := GETF ('GEN')
        .NOUNNUM := GETF ('NUM')

Q2:1    .ADJ := *
        .ADJNUM := GETF ('NUM')
        .ADJGEN := GETF ('GEN')
```

Note that BUILD is used as part of an assignment action for Arc NP:1. When this action is evaluated, the BUILD is evaluated in just the same way as it is when it appears in connection with a final state. However, the value of this evaluation, rather than being returned as the analysis of the network, is set as the value of the register referenced, .ART.

This instance of BUILD is somewhat different from the instances shown earlier. It has only one argument, the skeleton, and there is an asterisk in the skeleton. As usual, the asterisk stands for the current word, so that for each instance of asterisk, the current word

is substituted in the skeleton. Suppose for example that the current word were EL, an article in Spanish. Then the value of BUILD (' (DET:ART *) ') would be (DET:ART EL). The effect of the entire action would be to cause register .ART to take on the value (DET:ART EL).

The next action, .ARTGEN := GETF ('GEN'), also has some thing new. GETF is a **subroutine**, details of which are given in Section 9. GETF checks the feature list entered in the dictionary entry under the current word. If the current word were EL, then the feature list shown in the dictionary would be (NUM SG GEN MASC). The argument of GETF specifies the particular feature in which we are interested. Hence the value of GETF ('GEN') for EL is MASC. The action then sets register .ARTGEN to MASC. In a similar way, the action .ARTNUM := GETF ('NUM') would, for this same word, cause register .ARTNUM to be set to SG.

Finally, consider the fourth action of Arc NP:2, .ARTGEN := .NOUNGEN. This is an instance in which we simply copy from one register to another. Register .ARTGEN is set to the value of register .NOUNGEN.

We see that there are a number of things which can appear to the right of the assignment operator := . The feature common to all of these is that they have a value associated with them when they are evaluated. We will call these entities **forms**. There are four kind of forms which have been introduced so far: the distinguished register *, ordinary registers such as .NOUNGEN, the function BUILD, and the subroutine GETF. In fact, any form may be used in connection with a final state, although BUILD is by far the most commonly used.

To make Figure 4 complete, let us add the following final state conditions and forms:

```
Q2      cond:   .ARTGEN = .NOUNGEN & .NOUNNUM = .ARTNUM
        form:   BUILD (' (NP +  (HEAD:NOUN +))', .ART, .NOUN)
Q3      cond:   .ARTGEN = .NOUNGEN = .ADJGEN &
                .ARTNUM = .NOUNNUM = .ADJNUM
        form:   BUILD (' (NP +  (HEAD:NOUN +)  (MOD:ADJ +))',
                .ART, .NOUN, .ADJ)
```

The condition for Final State Q2 is satisfied if and only if the value of .ARTGEN is the same as the value of .NOUNGEN and the value of .NOUNNUM equals that of .ARTNUM. If this condition is satisfied and if the phrase is at an end, then the BUILD form is evaluated and the resultant value is taken as the analysis of the noun phrase. The situation is similar for Final State Q3.

Let us trace through an example, (LAS CASAS).

(a) The first word is LAS. Referring to Dictionary 8, we see that LAS is of the grammatical category ART. Consequently, Arc NP:1 can be travelled. Now the actions for Arc NP:1 are evaluated. The first action sets .ART to the value of BUILD (' (DET: ART *) '). Since the current word is LAS, the value of the BUILD is (DET:ART LAS).

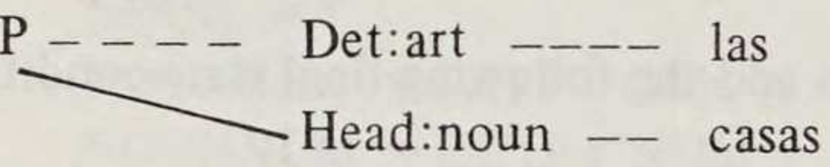
The second and third actions make use of GETF as shown before. The feature list for LAS is (NUM PL GEN FEM). The second and third actions thus set .ARTGEN to FEM and .ARTNUM to PL. Upon reaching State Q1, we have the following values for the registers:

.ART	(DET:ART LAS)
.ARTGEN	FEM
.ARTNUM	PL

(b) Beginning at State Q1, the current word is CASAS. We again consult the dictionary. Since CASAS is a noun, Arc Q1:1 is travelled. The feature list for CASAS is (NUM PL GEN FEM). On completing the actions we have the following values for the registers:

.NOUN	CASAS
.NOUNGEN	FEM
.NOUNNUM	PL

(c) Having travelled Arc Q1:1, the network is at State Q2. The phrase is finished, so the final state condition and form for Q2 are evaluated. Note now that the values of both .ARTGEN and .NOUNGEN are FEM and that the values of both .NOUNNUM and .ARTNUM are PL. The condition is therefore satisfied and the associated form can be evaluated. The value of .ART as shown above is (DET:ART LAS) and the value of .NOUN is CASAS. So the value of the form is (NP (DET:ART LAS) (HEAD: NOUN CASAS)). This corresponds to the tree shown below:



Let us trace through another example, (CASAS):

(a) Since the current word is CASAS, which in the dictionary is a noun, Arc NP:2 is travelled. After the actions have been evaluated, we have the following values for the registers:

.NOUN	CASAS
.NOUNGEN	FEM
.NOUNNUM	PL
.ARTGEN	FEM
.ARTNUM	PL

(b) Reaching State Q2, the phrase is finished so the condition for a final state is evaluated. Note from above that .NOUNGEN and .ARTGEN have the same value, as do .NOUNNUM and .ARTNUM. So the condition is satisfied and the form is evaluated. Now, Arc NP:1 has not been travelled. Hence no value has been assigned register .ART. Since .ART has no value, no value is substituted in the BUILD skeleton for .ART. Hence the value of the BUILD is (NP (HEAD:NOUN CASAS)), which corresponds to the following tree:

NP ———— Head:noun ———— casas

Finally, consider (LAS CASAS ROJAS):

(a) First, Arc NP:1 is travelled, causing the network to reach State Q1. The actions on Arc NP:1 result in the following register values:

.ART	(DET:ART LAS)
.ARTGEN	FEM
.ARTNUM	PL

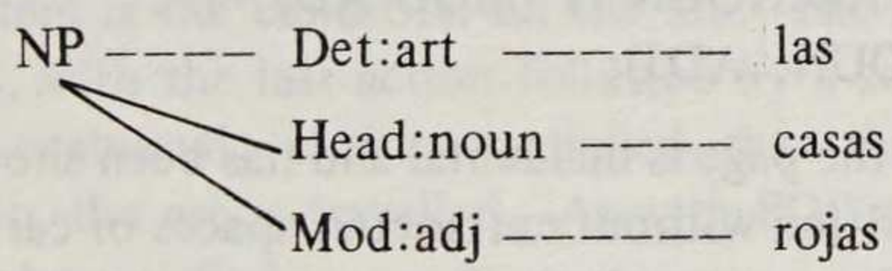
(b) Then Arc Q1:1 is travelled, causing the network to reach State Q2. The actions on this arc result in the following register values:

.NOUN	CASAS
.NOUNGEN	FEM
.NOUNNUM	PL

(c) Since the phrase is not finished, State Q2 is not considered a final state. Arc Q2:1 is travelled. The actions specified result in the following register values:

.ADJ	ROJAS
.ADJNUM	PL
.ADJGEN	FEM

(d) At State Q3, the phrase is finished and the final state condition is evaluated. Since .ARTGEN, .NOUNGEN and .ADJGEN all have the same value and since .ARTNUM and .NOUNNUM and .ADJNUM all have the same value, the condition is satisfied. The value of the form therefore is (NP (DET:ART LAS) (HEAD:NOUN CASAS) (MOD: ADJ ROJAS)), which corresponds to the following tree:



It is also convenient to permit conditions of the same sort on arcs. The condition specified must be satisfied before the primary condition (written on the arc) is tried. If that then is successful, the arc is travelled and the actions specified are performed. However, in the case of CAT arcs (see Section 5), the primary condition is evaluated before the additonal specified condition is evaluated.

5. A LINEAR FORM FOR TRANSITION NETWORKS

There are two ways in which a transition network can be written. One is the graph form which we have been using up to this time. It is often convenient, and it is very easy to follow; but it can become awkward as a network grows in complexity, since it involves frequent rewriting of the whole network. There is another form, which can be done with a typewriter or on file cards. This is called the **linear** form. This linear form is also easier

for certain explanatory purposes. Let us convert Figure 4 to linear form.

```
10)  NP:  CAT ('ART'), TO Q1, ,
                .ART := BUILD (' (DET:ART *) '),
                .ARTGEN := GETF ('GEN'),
                .ARTNUM := GETF ('NUM');
          CAT ('NOUN'), TO Q2, ,
                .NOUN := *,
                .NOUNGEN := GETF ('GEN'),
                .NOUNNUM := GETF ('NUM'),
                .ARTGEN := .NOUNGEN,
                .ARTNUM := .NOUNNUM;

     Q1:  CAT ('NOUN'), TO  Q2, ,
                .NOUN := *,
                .NOUNNUM := GETF ('NUM'),
                .NOUNGEN := GETF ('GEN');

     Q2:  CAT ('ADJ'), TO  Q3, ,
                .ADJ := *,
                .ADJNUM := GETF ('NUM'),
                .ADJGEN := GETF ('GEN');
          POP   .ARTGEN=.NOUNGEN & .NOUNNUM=.ARTNUM,
                BUILD (' (NP +  (HEAD:NOUN +))', .ART, .NOUN);

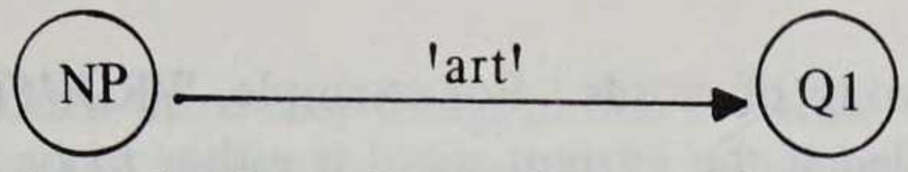
     Q3:  POP   .ARTGEN=.NOUNGEN=.ADJGEN & .ARTNUM=.NOUNNUM=
                .ADJNUM,
                BUILD (' (NP +  (HEAD:NOUN +)  (MOD:ADJ +))',
                          .ART, .NOUN, .ADJ);
```

The layout of the linear form on the page is incidental and has been shown this way only to help readability. It could be written without extraneous spaces or carriage returns (except for the spaces needed to set off atoms in a list). For instance, we could have

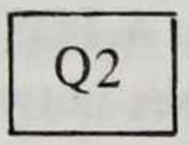
```
NP:CAT('ART'),TOQ1,,.ART:=BUILD('(DET:ART &)'),.ARTG
EN:=GETF('GEN'),.ARTNUM:=GETF('NUM');CAT('NOUN'),TOQ
```

Notice the relationship between the graph and linear forms. Each state name is followed by a colon, as in NP: or Q3:. The arcs leading from each state follow the name, with each arc being terminated by a semicolon. For State NP, we have two arcs, one beginning with CAT('ART'),TO Q1,, and ending with GETF('NUM');, and the other beginning with CAT('NOUN'),TO Q2,, and ending with .NOUNNUM;.

The first item in an arc is the arc type. The first arc for State NP is a CAT (for *category*) arc. Its argument, enclosed in parentheses, indicates that the current word must be of category 'ART' in the dictionary. Graphically, this is represented by an arrow with the argument written in lower case letters on the arrow:



The second arc for State Q2 is of type POP. This shows the final state condition and form. POP is really a dummy arc; that is, it does not cause the network to move to another state but instead causes the network to terminate. Syntactically, however, it is treated as an arc to allow ordering with respect to other arcs. That is, for State Q2, first the CAT arc will be attempted. If this is not successful, only then will the POP arc be attempted. Graphically, a POP arc is represented by a square box around the state to which it is affixed:



For a POP arc, the second item (after the word POP) is the condition. In Arc Q2:2, the POP condition is .ARTGEN = .NOUNGEN & .ARTNUM = .NOUNNUM. In general, the POP condition is optional and if it is omitted, it is assumed to be satisfied by all cases. After the condition (whether it is actually present or not) is a comma, after which is the form, terminated with a semicolon. For example, suppose there were no condition placed on the final state for Q2. Then it would be written

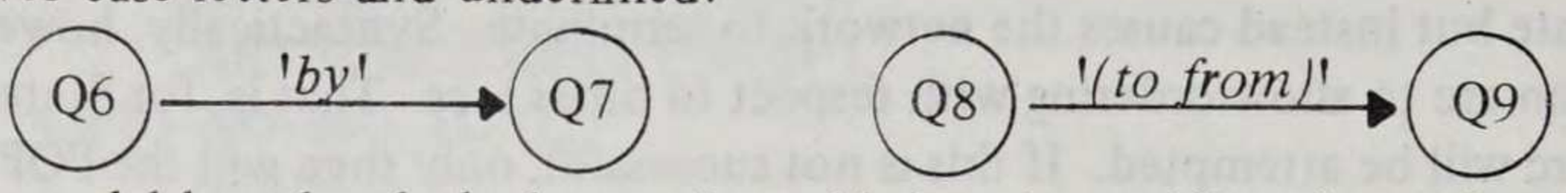
```
POP ,BUILD('(NP + (HEAD:NOUN +))',.ART,.NOUN);
```

For CAT arcs, the second item (such as TO Q2) indicates the state to which the network will transfer after the arc is travelled. The TO preceding the state name indicates that arcs from the next state will consider the next word in the input. If JUMP is written instead of TO (such as JUMP Q2), then the arcs from the next state will consider the same word again. That is, TO causes the current word to be advanced, JUMP does not. Some applications of the use of JUMP will be shown later. The third item is the condition on the arc. The actions follow the condition, set off by commas, with the last action followed by a semicolon. In evaluating an arc, first the CAT is evaluated. If this is satisfied, then the condition is evaluated. If this is satisfied also, the arc is travelled. As with POP arcs, if the condition is omitted, it is assumed to be satisfied.

There are several other arc types, including VIR, TST, PUSH and WRD, which are explained below. These all have the same format as CAT (arc type, destination state, condition, actions) though their functions are different. One difference is that the condition for these arc types is evaluated first; if that is successful, then the arc type is evaluated. Then if both are satisfied, the arc is travelled.

Frequently one wishes to test a particular word or group of words against the current word, rather than testing for a particular category. For instance, in *John was seen by Joe, by Joe* is not an ordinary prepositional phrase, but indicates an agent, *Joe.* A network for English would then test for the word *by* after transducing the subject and the verb phrase, having noted that the verb phrase is passive. To make such a test, a WRD arc is provided. For example, WRD('BY') tests the current word, and if it is *by*, then the arc is travelled (assuming the condition has first been met). Or, one can

test the current word against a group of words. For example, WRD('(TO FROM)'). An arc of this type will be travelled if the current word is either *to* or *from*. Graphically, this arc type is represented by an arrow with the argument written on it in the lower case letters and underlined:



Our model has already had recursion added to it, enabling the evaluation of one network to be suspended while another (or the same) network is evaluated. This is actually accomplished by means of a PUSH arc. A PUSH arc names a network which is to be evaluated. For example, PUSH(PP) causes network PP to be evaluated. (PP is also called a **subnetwork** since its evaluation is embedded in the evaluation of another network.) The network named is evaluated when a PUSH is encountered (assuming the condition is first satisfied), and its evaluation terminates with a POP arc. (The subnetwork may, of course, fail, in which case the PUSH arc which calls for the subnetwork also fails.) This POP arc has a form associated with it which represents a partial analysis of the total input. It is necessary therefore for the referencing network to have access to this partial analysis so that it can be incorporated into the overall analysis. The convention we adopt, following Woods, is that the entire result of the subnetwork evaluation referenced by a PUSH arc is set into the distinguished register *, rather than the current word. Thus the actions on the arc can make reference to this partial evaluation. As an example, let us linearize the networks NP and PP of Figure 3, adding actions and final state conditions and forms:

```
11)  NP:  CAT('DET'),TO Q6,,
                .DET := *;
          CAT('NPR'),TO Q8,,
                .NPR := *;

     Q6:  CAT('ADJ'),TO Q6,,
                .MODS := .MODS // BUILD('(*)');
          CAT('NOUN'),TO Q7,,
                .NOUN := *;

     Q7:  PUSH(PP),TO Q7,,
                .PP := *;
          POP ,BUILD('(NP (DET:ART +) (# MOD:ADJ +) (HEAD:NOUN
                      +) (# & PHMOD: +)',.DET,.MODS,.NOUN,.PP);

     Q8:  POP ,BUILD('(NP (HEAD:NPR +))',.NPR);

     PP:  CAT('PREP'),TO Q9,,
                .PREP := *;

     Q9:  PUSH(NP),TO Q10,,
                .NP := *;

    Q10: POP ,BUILD('(PP (LOC:PREP +) (# &HEAD: +))',.PREP,.NP);
```

We have added three new things in this network. The first is found in the action of Arc Q6:1. This is the **catenation operator** //. It combines the value of the form on the left with the value of the form on the right into a single list. For example, suppose that register .R1 had the value (A B C) and that register .R2 had the value (D E F G). Then .R1 // .R2 would have the value (A B C D E F G).

The second thing that has been added is found in the BUILD skeleton of Arc Q7:2. The atom # has a special meaning here. It is similar to the catenation operator in that all the items which follow are combined into a single list. Suppose .MODS contained (BIG TALL RED). Then BUILD('(#MOD:ADJ +)',.MODS) would have as its value (MOD:ADJ BIG TALL RED). By contrast, BUILD('(MOD:ADJ +)',.MODS) would have as its value (MOD:ADJ (BIG TALL RED)). The effect of # takes place after all substitutions have been made; that is, the skeleton is rescanned for instances of # and the catenation is performed.

The third item which has been added is found in the BUILDs of Arcs Q7:2 and Q10:1. It is the use of the atom &. The effect of & is to squash a string of atoms into a single atom at the beginning of a list. Suppose register .CHAR contained the atom C. Then in BUILD('(&A B +)',.CHAR), first the value of .CHAR is substituted for the value of +. The value of BUILD at this point is (& A B C). Then the & causes the atoms to be combined so that the result of the BUILD is a list (ABC) that contains the single atom ABC.

The effect of & in a skeleton takes place after substitution of argument values for instances of + and after instances of # have been evaluated. Suppose register .NP contained the list (NP (DET:ART THE) (MOD:ADJ SWEET RED) (HEAD:NOUN ROSES)). Then BUILD('(# & HEAD: +)',.NP) would be evaluated as follows: First the value of .NP would be substituted for + in the skeleton. The value of BUILD at this point would thus be (# & HEAD: (NP (DET:ART THE) (MOD:ADJ SWEET RED) (HEAD:NOUN ROSES))). Then # would take effect, yielding (& HEAD: NP (DET:ART THE) (MOD:ADJ SWEET RED) (HEAD:NOUN ROSES)). Then & would take effect. First, HEAD: and NP would be combined into a single atom, HEAD:NP. Since what follows is a list, the squashing stops. So the value of the BUILD finally is (HEAD:NP (DET:ART THE) (MOD:ADJ SWEET RED) (HEAD:NOUN ROSES)).

Let us trace through this network for the sample noun phrase (THAT NICE MAN WITH THE SWEET RED ROSES). The paths for embedded evaluations are indented. Thus steps (e) through (1) are embedded in step (d). Similarly, steps (g) through (k) are embedded in (f).

(a) NP:1, current word is THAT.
.DET THAT

(b) Q6:1, current word is NICE.
.MODS (NICE)
To see how .MODS was set, we proceed as follows. .MODS initially has no

value (we say it is *null*). BUILD('(*)') yields the value (NICE) since the current word, represented by *, is NICE. So .MODS / / BUILD('(*)') has the value (NICE), which is set as the new value of .MODS.

(c) Q6:2, current word is MAN.
.NOUN MAN

(d) Q7:1, current word is WITH.
.PP (PP (LOC:PREP WITH)
(HEAD:NP (DET:ART THE
(MOD:ADJ SWEET RED)
(HEAD:NOUN ROSES)))

Note here that register .PP, according to the action on Arc Q7:1, is set to the contents of *. But * has been set to the result of the evaluation of subnetwork PP (which we will trace below). The important thing to notice here is that for Arc Q7:1, the value of * when the PUSH arc is travelled is not the current word, but is the result of the embedded evaluation.

(e) PP:1, current word is WITH.
.PREP WITH

(f) Q9:1, current word is THE.
.NP (NP (DET:ART THE)
(MOD:ADJ SWEET RED)
(HEAD:NOUN ROSES))

Note again that the result of the embedded evaluation is set into register .NP, rather than the current word.

(g) NP:1, current word is THE.
.DET THE

(h) Q6:1, current word is SWEET.
.MODS (SWEET)

(i) Q6:1, current word is RED.
.MODS (SWEET RED)

(j) Q6:2, current word is ROSES.
.NOUN ROSES

(k) Q7:2, final state.
form (NP (DET:ART THE)
(MOD:ADJ SWEET RED)
(HEAD:NOUN ROSES))

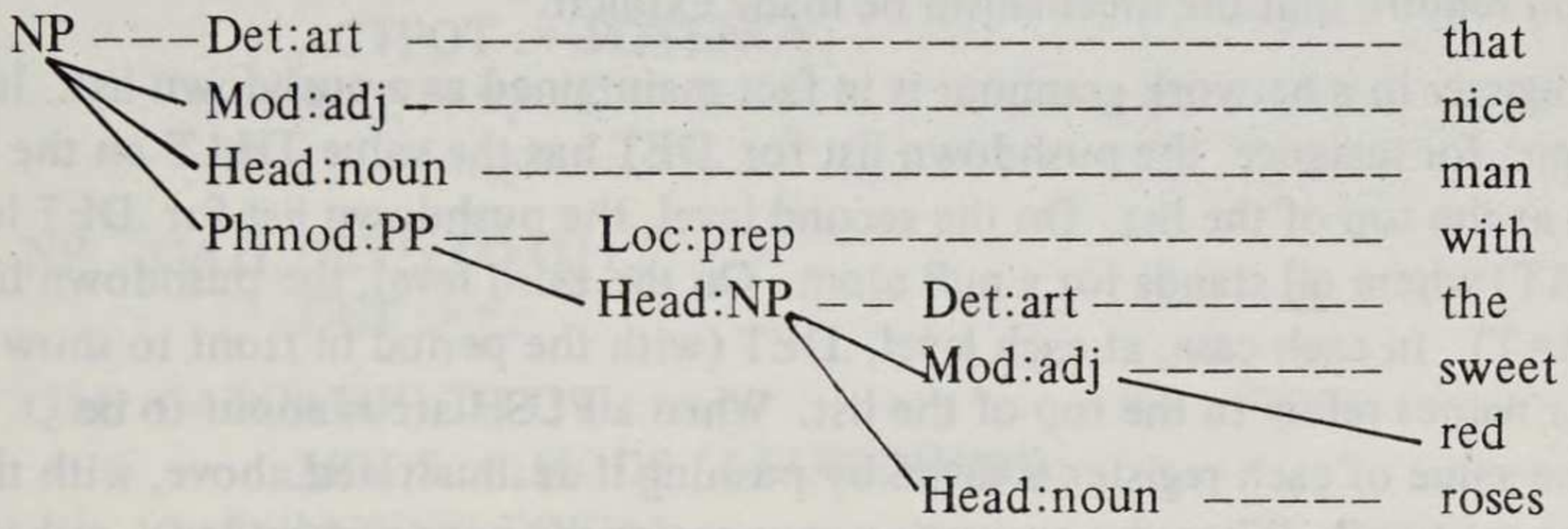
Note how this form value is that which is set into * for Step (f).

(l) Q10:1, final state.
form (PP (LOC:PREP WITH)
(HEAD:NP (DET:ART THE)

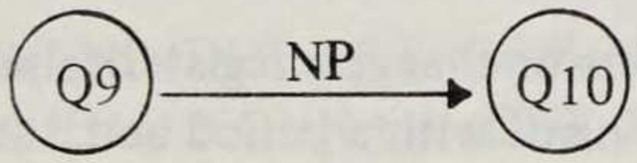
(MOD:ADJ SWEET RED)
(HEAD:NOUN ROSES)))

(m) Q7:2, final state.
form (NP (DET:ART THAT)
(MOD:ADJ NICE)
(HEAD:NOUN MAN)
(PHMOD:PP (LOC:PREP WITH)
(HEAD:NP (DET:ART THE)
(MOD:ADJ SWEET RED)
(HEAD:NOUN ROSES))))

This last form represents the final analysis for the noun phrase and corresponds to the following tree:



A PUSH arc is written graphically as an arrow with the name of the subnetwork written on it in upper case letters:



There was another mechanism operating in this example which has not been discussed yet: the pushdown list. Notice in Step (a) of the example that .DET is set to THAT. This takes place on the highest level. The NP network which is being evaluated references the PP subnetwork with a PUSH arc, sending evaluation down to the second level. The PP subnetwork in turn references the NP network as a subnetwork, sending evaluation down to the third level. On this level, .DET is set to THE in Step (g). The final state form is built in Step (k) and evaluation returns to the second level. Step (1) builds the final state form for the second level and evaluation resumes at the highest level. Note now that back on this highest level, .DET retains its original value, THAT, despite the fact that a register called .DET was set to THE on the third level. This is made possible by the pushdown list. A **pushdown list** is a special kind of list, also called a last in-first out list. That is, items are added sequentially to the list, and are removed sequentially with the condition that the most recently added item is the first which can be removed.

Suppose for example that such a list is initially null. Add the atom A. The value of the list is then (A). Add the atom B. The value of the list is then (B A). Add the atom C.

The value of the list is then (C B A). Remove an item. The atom C, being the most recently added item, is the one removed, leaving the list at (B A). Remove another item. The list now has the value (A). Add the atom D. The list now has the value (D A). The operation of adding items is called **pushing** onto the list. The operation of removing items is called **popping** from the list. There are the source of the names PUSH and POP used on arcs.

It should be pointed out that the mechanism of a pushdown list is implicit both in the TNG notation scheme as well as in the traditional one, and in fact is required of any notation scheme which permits recursion. The difference is that in the traditional notation, the concept of a pushdown list is taken for granted. The linguist in tracing the syntagmemes in an example similar to the one above simply remembers the word or construction associated with each tagmeme. The increased power and flexibility of the TNG notation require that the mechanism be made explicit.

Each register in a network grammar is in fact maintained as a pushdown list. In the above example, for instance, the pushdown list for .DET has the value THAT on the highest level at the top of the list. On the second level, the pushdown list for .DET looks like (nil THAT) where nil stands for a null atom. On the third level, the pushdown list is (THE nil THAT). In each case, at each level, .DET (with the period in front to show that it is a register name) refers to the top of the list. When a PUSH arc is about to be evaluated, the value of each register is saved by pushing it as illustrated above, with the top of the list set to nil. When the evaluation resumes at the next highest level, either because of a POP arc or because the subnetwork failed, then the original register values are restored by popping them from the list.

It is sometimes necessary to access or change a register value on a higher level. This may be done by prefacing the register name with a period and a string of one or more dollar signs. In Step (g) of the preceding example, on the third level, .DET is set to THE. Hence the pushdown list for .DET is changed from (nil nil THAT) to (THE nil THAT). The accessible value of .DET is THE. To access the second level value, .$DET is written. .$DET has the value nil, and .$$DET has the value THAT. Suppose in Step (g) we were to add the action .$$DET := .DET. Then the pushdown list would be changed to (THE nil THE). On the other hand, .DET := .$$DET would cause the pushdown list to be changed to (THAT nil THAT). The kind of a pushdown list that permits values that are not accessible at the top to be read or changed is technically known as a **stack**.

This kind of access between levels allows intercommunication between levels, which is necessary because language is not strictly hierarchichal. For example, a clause level analysis often requires information from within the noun phrase analysis as to the potency (animate or inanimate) of the head of the phrase, morphological affixes, etc., which are available only on the phrase level. Conversely, it might be necessary for a lower level network to have access to information from a higher level to permit evaluation of the lower level network. For example, consider a language such as Spanish which was illustrated earlier. There must be concord between a noun phrase and its

associated adjective phrase. The adjective phrase may be written as a subnetwork, which would be reasonable since adjective phrases are also found separate from nouns as verbal complements. Then the subnetwork would access grammatical number and gender information from the noun phrase network, and would fail if the adjective phrase did not agree with the noun phrase network.

As an example of upward communication, consider the following network fragment for the clause level. Register .POTENCY will be used to communicate role potency (animate-inanimate) from the phrase to the clause level. A sample dictionary entry might be:

```
(MEN (NOUN (FEATURES (NU PL POT ANIM))))

CL:  ...
     PUSH (NP),TO Q7,,
          .ACTPOT := .POTENCY,
          .ACT := *;
     ...
NP:  CAT('DET'),TO Q11,,
          .DET := *;
Q11: CAT('ADJ'),TO Q11,,
          .MODS := .MODS / / BUILD('(*)');
     CAT('NOUN'),TO Q12,,
          .NOUN := *,
          .$POTENCY := GETF('POT');
Q12; POP ,BUILD('(NP (DET:ART +) (# MOD:ADJ +)
                    (HEAD:NOUN +))',.DET,.MODS,.NOUN);
```

The noun phrase shown is simpler than Figure 3 to eliminate detail not necessary for the example. Let us assume that in the course of clause transduction the arc PUSH (NP) is encountered and that the input at that point is (. . . THE GOOD MEN . . .). Then evaluation of that arc results in .ACTPOT being set to ANIM and .ACT to (NP (DET: ART THE) (MOD:ADJ GOOD) (HEAD:NOUN MEN)). To see how this was accomplished, let us trace through the NP evaluation.

(a) NP:1, current word is THE.
.DET THE

(b) Q11:1, current word is GOOD.
.MODS (GOOD)

(c) Q11.2, current word is MEN.
.NOUN MEN
.$POTENCY ANIM

Note that Register .POTENCY is set to ANIM on the higher level.

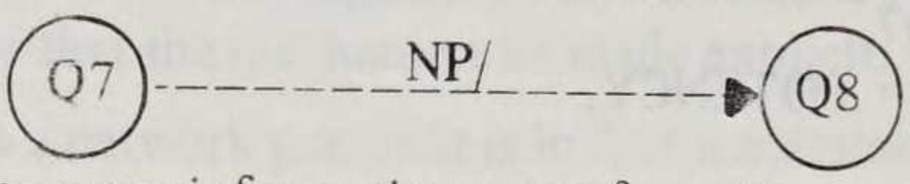
Hence the stack for .POTENCY takes on the value [nil ANIM].

(d) Q12, final state
form (NP (DET:ART THE) (MOD:ADJ GOOD) (HEAD:NOUN MEN))

Back on the higher level, the stack for .POTENCY has the value (ANIM), so that .POTENCY has the value ANIM. This value is set into .ACTPOT in the CL network.

There is yet another way of transmitting information between levels, but this will be covered in Section 9 after the discussion of subroutines.

In certain very rare cases it is necessary to evaluate a subnetwork as with a PUSH, but with the input string not advanced. This is accomplished with a TST arc which is identical to a PUSH arc with this restriction. This is graphically represented as for a PUSH arc, but the arrow is written with a broken rather than solid line:



Many languages move information away from its normal word order into clause initial position in order to topicalize it or for other reasons. In a constituent analysis of a sentence one may wish to show this information in normal rather than fronted order, while keeping track of the fact that it has been fronted in a register. To permit this conveniently, a pair of mechanisms is included in TNG. The first is called the HOLD list. Information may be entered on the HOLD list when it occurs, and may be extracted later by a special arc called a VIR arc (for *virtual*, since it extracts information from the HOLD list at the place where the information normally would occur even though it isn't actually there).

Information may be entered on the HOLD list by the HOLD action. Suppose that a noun phrase has been built up and set into a register called .SUBJ. Later it is found that that this noun phrase is not in fact the subject of the sentence, but rather has been fronted from its normal order, as in *The big man it was sent by*. Here *it* is the real subject of the sentence, whereas *the big man* is a marked topic. However, at first *the big man* is assumed to be the subject. So, until *it* is encountered, the register .SUBJ contains the analysis (NP (DET:ART THE) (MOD:ADJ BIG) (HEAD:NOUN MAN)). Then when *it* is encountered, the contents of .SUBJ are entered onto the HOLD list and (NP (HEAD:PRO IT)) is entered into .SUBJ instead. The arc actions would actually look like this:

```
Q3:  . . .
        PUSH(NP),TO Q6,,
              HOLD(.SUBJ),
              .SUBJ := *;
```

The HOLD list then has as its value (NP (DET:ART THE) (MOD:ADJ BIG) (HEAD: NOUN MAN)). More than one structure may be entered on the HOLD list: ((str1) (str2) . . .)), where each structure is a list whose first atom gives the kind of structure. For the above example, the first atom is NP, showing that the structure is a noun phrase.

Later on, we could have the following arcs:

```
Q7:  WRD('BY'),TO Q8;
Q8:  VIR('NP'),TO Q9,,
                .AGENT := *;
     PUSH(NP),TO Q9,,
                .AGENT := *;
```

Arc Q7:1 transduces the *by* in the sample sentence. Arc Q8:1 inspects the HOLD list. If an entry appears which begins with NP, then that entry is removed from the HOLD list and entered as the value of the distinguished register *. Thus the action of Q8:1 sets .AGENT to (NP (DET:ART THE) (MOD:ADJ BIG) (HEAD:NOUN MAN)), and the HOLD list to nil. Now suppose instead that the sentence were *It was sent by the big man.* Here *the big man* appears in its normal order, and nothing is entered on the HOLD list. Instead of Arc Q8:1 being taken, Arc Q8:2 is taken, resulting in register .AGENT being set to (NP (DET:ART THE) (MOD:ADJ BIG) (HEAD:NOUN MAN)). Thus the values of registers .SUBJ and .AGENT are the same, whether *the big man* is fronted or not.

We add a further condition now to final states. For the POP arc to be taken, the HOLD list for that level must be null. That is, each HOLD action must have a corresponding VIR arc that removes whatever structure was placed there by the HOLD action. Additional details about the HOLD action and the VIR arc are given in the appendix.

Additional actions and forms are available to manipulate information accumulated during transduction. These are summarized in the appendix.

6. DECLARATIONS

Punctuation and similar symbols are often included in input. However, if the input were something such as (WHAT IS THE MATTER, ANYWAY?), the fourth and fifth words would not be MATTER and ANYWAY but would be MATTER, and ANYWAY? instead. It would be useful to have this input converted to (WHAT IS THE MATTER , ANYWAY ?), so that punctuation symbols would be treated as separate atoms. To permit this, we allow the use of a **declaration**. A declaration gives the system some general information to allow it to operate properly. We can use the PUNCTUATION declaration:

```
PUNCTUATION , ;
PUNCTUATION ? ;
```

Whenever a comma or a question mark is encountered in the input text, it is automatically separated into a separate atom.

Another declaration allows a register to have an initial value when the network evaluation begins. This value may be changed, if desired, during evaluation, but this declaration specifies its initial value.

INITIAL .DICBLD '(+ FEATURES (NUM + GEN +))';

By this declaration, register .DICBLD will be set to the skeleton structure (+ FEATURES (NUM + GEN +)) when network evaluation begins. Several registers may be set to initial values at once:

INITIAL .TNS 'PRES!, .PERS 'FIRST';

It is also convenient to have a register which is **static**; that is, which is never entered on a pushdown list, even when a PUSH is evaluated. This means that it may be set at any level (say level 3), and it retains that value at any other level (such as level 5 or level 2) irrespective of PUSHes or POPs, until its value is reset at some level. The STATIC declaration is used here:

STATIC .DICBLD;

Register .DICBLD is declared to be static throughout the evaluation of the network. It is never pushed or popped. As a consequence, $ may never precede a static register name. A reference such as .$DICBLD is meaningless. Several registers may be declared static at once:

STATIC .DICBLD,.TEMP;

Two other declarations are shown below, GLOBAL and COMPLEX. GLOBAL is explained in Section 8, COMPLEX in Section 7.

GLOBAL register;
GLOBAL register,register, . . . ;

COMPLEX atom state-name [MORPH / LETTER] ;;

COMPLEX atom state-name [MORPH / LETTER] , atom state-name [MORPH / LETTER] , . . . ;

7. COMPLEX WORDS

Many languages are morphologically complex. With person, tense, aspect, number, gender, and other features marked in the verb, it quickly becomes awkward to have to have each form of each verb in the dictionary. When similar conditions hold for other parts of speech, it becomes clear that there must be a mechanism to allow morphologically complex words to be analyzed rather than stored. In addition, tagmemics permits word level syntagmemes, so if TNG is to be an adequate notational scheme for tagmemics, it must be able to analyze complex words.

There is a fundamental difference between word level analysis and analysis on higher levels: on the higher levels, there are graphic boundary markers (at least spaces) between words and often between certain kinds of clauses as well as sentences and paragraphs. Other boundaries are represented orthographically as commas, periods,

parentheses, or quotation marks. Usually, however, there are no morpheme boundary markers within a word. Either the transcription omits those phonological features which indicate boundaries, or there may be skewing between phonology and grammar so that morphological boundaries have no phonological indication.

The traditional notation for tagmemics does not recognize these problems. Instead, it gives word level syntagmemes in a manner which is analogous to higher level syntagmemes. The linguist, when interpreting the word level syntagmeme, parses the word in question according to some unspecified rule, and matches the morphemes so parsed against the tagmemes. In other words, the traditional notation assumes the linguist will make the right cuts (that is, will insert morpheme boundaries in the right places).

One of the goals of the TNG system is to eliminate this sort of unspecified assumption. Consequently we wish to make explicit the parsing technique used for handling complex words. However, this technique is complicated and not fully understood yet, particularly in the case of morphophonemic reduction. Therefore the mechanisms presented in this section are nowhere near final.

The COMPLEX declaration comes into play here. As in the preceding section, it has two formats:

```
COMPLEX atom state-name MORPH;
COMPLEX atom state-name LETTER;
```

The atom cited is a dictionary category like VERB. The state-name is the name of a subnetwork. The COMPLEX declaration operates as follows: when a CAT arc is evaluated which specifies a dictionary category that matches the atom, then the subnetwork is evaluated. Suppose we had a declaration

```
COMPLEX  N  NOUNW  MORPH;
```

Then whenever CAT('N') is evaluated in the network, the declaration COMPLEX N brings about an automatic push to a subnetwork called NOUNW. The input to that network is the current word, which MORPH directs to be parsed according to morpheme breaks provided by the linguist.

Consider the dictionary of Spanish given earlier as Example 8. If we take this limited corpus as representative, we may draw some general conclusions. First, gender is usually indicated by 0 for masculine and by A for feminine. Also, absence or presence of a final S or ES indicates singular or plural respectively. The dictionary shown is therefore redundant since the difference between, say, LAS and LOS is predictable. Note, however, that EL is irregular. Let us therefore establish the following alternate dictionary:

13)
```
(A    (GENM FEATURES  (GEN FEM)))
(CAS     (NOUNM FEATURES  (GEN FEM)))
(EL   (ART FEATURES  (NUM SG GEN MASC)))
(L    (ARTM))
(LO   (ART FEATURES  (IRR))
```

```
(O    (GENM FEATURES  (GEN MASC)))
(OJ   (NOUNM FEATURES  (GEN MASC)))
(ROJ   (ADJM))
```

Note the entry for LO. The network has been written so that a feature tag of IRR causes the word so tagged to be rejected. This is done via a condition on Final State of the example which follows Q7/. Another way to handle irregular forms is shown in Section 10, which gives a major example for English.

Note also in this dictionary that only morphemes have been entered, so that the dictionary size has been substantially reduced. What happens to these complex words is perhaps best explained by means of an example based on this dictionary.

First we must take the appropriate declarations:

```
COMPLEX ART ARTW MORPH,
        NOUN NOUNW MORPH,
        ADJ ADJW MORPH;
INITIAL .DICBLD '(+ FEATURES (NUM + GEN +))'.
        .STRS '(+ (NUC:ROOT +) (PER:GENSFX +))',
        .STRP '(+ (NUC:ROOT +) (PER1:GENSFX +)
            (PER2:PLSFX +))';
STATIC  .DICBLD,.STRS,.STRP;
```

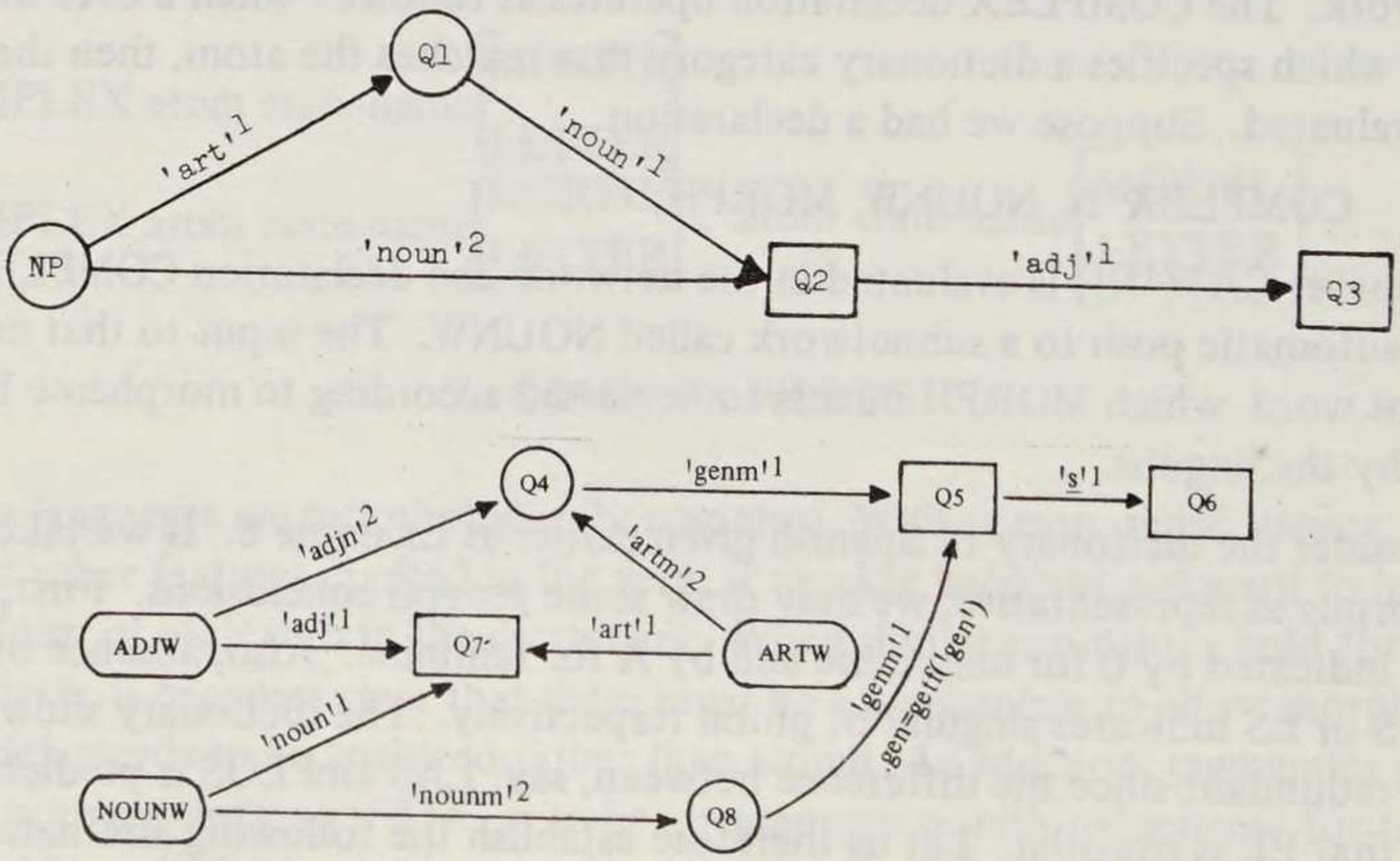


Figure 5

For this grammar, given below, note that Arc Q5:1 is a WRD arc whose argument is 'S'. This is not to be confused with an arc such as Q4:1 which is a CAT arc, written CAT ('GENM').

```
14)  NP:    CAT('ART'),TO Q1,,
                  .ART := BUILD(' (# & DET: *) '),
                  .ARTGEN := GETF('GEN'),
                  .ARTNUM := GETF('NUM');
             CAT('NOUN'),TO Q2,,
                  .NOUN := *,
                  .NOUNGEN := GETF('GEN'),
                  .NOUNNUM := GETF('NUM'),
                  .ARTGEN := .NOUNGEN,
                  .ARTNUM := .NOUNNUM;
Q1:          CAT('NOUN'),TO Q2,,
                  .NOUN := *,
                  .NOUNGEN := GETF('GEN'),
                  .NOUNNUM := GETF('NUM');
Q2:          CAT('ADJ'),TO Q3,,
                  .ADJ := *,
                  .ADJGEN := GETF('GEN'),
                  .ADJNUM := GETF('NUM');
             POP' .ARTGEN = .NOUNGEN & .NOUNNUM = .ARTNUM,
                  BUILD('(NP + (# & HEAD: +))',.ART,.NOUN);
Q3:          POP  .ARTGEN=.NOUNGEN=.ADJGEN & .ARTNUM=.NOUNNUM=
                            .ADJNUM,
                  BUILD('(NP + (# & HEAD: +) (# & MOD: +))',
                       .ART,.NOUN,.ADJ);
ADJQ:        CAT('ADJ'),TO Q7,,
                  .WORD := *,
                  .CATEGORY := 'ADJ';
             CAT('ADJM'),TO Q4,,
                  .PARTM := *,
                  .CATEGORY := 'ADJ';
ARTW:        CAT('ART'),TO Q7,,
                  .WORD := *,
                  .CATEGORY := 'ART';
             CAT('ARTM'),TO Q4,,
                  .PARTM := *,
                  .CATEGORY := 'ART';
```

```
Q4:     CAT('GENM'),TO Q5,,
            .GEN := GETF('GEN'),
            .GENM := *;

Q5:     WRD('S'),TO Q6;
        POP  ,BUILD(.STRS,CATEGORY,.PARTM,.GENM),
             BUILD(.DICBLD,.CATEGORY,'SG',.GEN);

Q6:     POP  ,BUILD(.STRP,.CATEGORY,.PARTM,.GENM),
             BUILD(.DICBLD,.CATEGORY,'PL',.GEN);

NOUNW: CAT('NOUN'),TO Q7,,
            .WORD := *,
            .CATEGORY := 'NOUN';
       CAT('NOUNM'),TO Q8,,
            .PARTM := *,
            .CATEGORY := 'NOUN';

Q7:     POP  GETF('IRR'),
             BUILD(' (+ +)',.CATEGORY,.WORD),
             $;

Q8:     CAT('GENM'),TO Q5,.GEN=GETF('GEN'),
            .GENM := *;
```

Now we can illustrate the use of the COMPLEX declarations. For instance, consider COMPLEX ART ARTW MORPH. This means that whenever a CAT('ART') arc, such as NP:1, is evaluated, then an automatic push to the subnetwork ARTW takes place. However, the assumed input string to the subnetwork is not the original input, but is a parsed version of the complex word under consideration. MORPH in the COMPLEX declaration indicates that morpheme breaks have been marked by the linguist. Suppose for example that the current word is written L-A-S, in which hyphens indicate morpheme breaks. Evaluation of CAT('ART') results in a push to ARTW with the assumed input being (L A S), the word with hyphens removed and enclosed in parentheses.

Note in the network that Final States Q5:2, Q6:1, and Q7:1 have the format

POP condition, form1, form2;

That is because these are final states for word level subnetworks. Form1, as is the case for normal subnetworks, represents an analysis of the input word. Form2 is a new dictionary entry which is built and returned to the NP network as the entry for the complex word under consideration. The evaluation traced below shows how this works.

First it is necessary to introduce two new entities, the distinguished register $ for features of the current word and the function STRING. Suppose we had the following dictionary entry for English:

```
(MAN    (N FEATURES   (NU SG))
        (V FEATURES   (TRANS PASSIVE))
```

Then if the current word is MAN, a CAT('N') arc causes $ to be set to (N FEATURES (NU SG)). A CAT('V') arc causes $ to be set to (V FEATURES (TRANS PASSIVE)). The tracing below for the Spanish network shows another way $ can be set. The subroutine GETF makes use of $ in getting its value.

The function STRING has as its value the remainder of the input string beginning with the current word. If input were (JON TWIRLED JENNY IN HER SWING) and if the current word were JENNY, then the value of STRING at that point would be (JENNY IN HER SWING).

Let us trace an evaluation for (EL OJ-O ROJ-O).

(a) NP:1
$ (ART FEATURES (NUM SG GEN MASC))
.ART (DET:ART EL)
.ARTGEN MASC
.ARTNUM SG
STRING (EL OJ-O ROJ-O)

Note: The current word is the first word in STRING, here EL. Since the declaration COMPLEX ART ARTW MORPH has been included, the Arc NP:1, a CAT('ART') arc, causes a push to subnetwork ARTW, which we will trace below. The assumed input to ARTW is (EL).

(b) ARTW:1
.WORD EL
.CATEGORY ART

Note: Arc ARTW:1 is a CAT('ART') arc. However, since we are in the range of a COMPLEX declaration, another push does not take place. Rather, the dictionary is inspected to see if the current word, EL, is of type ART. That is, the arc functions normally.

(c) Q7:1, final state
form1 (ART EL)
form2 (ART FEATURES (NUM SG GEN MASC))

Note: This final state has two forms. Form1 is the usual kind for final states, giving a structural representation of the input. The second form is a generated dictionary entry for the complex word. Here, EL is irregular so the value of form2 is the value of $.

(d) Q1:1
$ (NOUN FEATURES (NUM SG GEN MASC))
.NOUN (NOUN (NUC:ROOT OJ) (PER:GENSFX O))
.NOUNGEN MASC
.NOUNNUM SG
STRING (OJ-O ROJ-O)

Note: Since Arc Q1:1 is a CAT('NOUN') arc and since there is a declaration of COMPLEX NOUN NOUNW MORPH, a COMPLEX declaration push is made to subnetwork NOUNW, with the assumed input being (OJ O).

(e) NOUNW:2

$	(NOUNM FEATURES (NUM SG GEN MASC))
.PARTM	OJ
.CATEGORY	NOUN
STRING	(OJ O)

(f) Q8:1

$	(GENM FEATURES (NUM SG GEN MASC))
.GENM	O
STRING	(O)

(g) Q5:2, final state

form1	(NOUN (NUC:ROOT OJ) (PER:GENSFX O))
form2	(NOUN FEATURES (NUM SG GEN MASC))

Note: Form2 gives a dictionary-like representation for the word OJO. This was generated from the features of the word's morphemes.

(h) Q2:1

$	(ADJ FEATURES (NUM SG GEN MASC))
.ADJ	(ADJ (NUC:ROOT ROJ) (PER:GENSFX O))
.ADJGEN	MASC
.ADJNUM	SG
STRING	(ROJ-O)

(i) ADJW:2

$	(ADJM)
.PARTM	ROJ
.CATEGORY	ADJ
STRING	(ROJ O)

(j) Q4:1

$	(GENM FEATURES (GEN MASC))
.GEN	MASC
.GENM	O
STRING	(O)

(k) Q5:2, final state

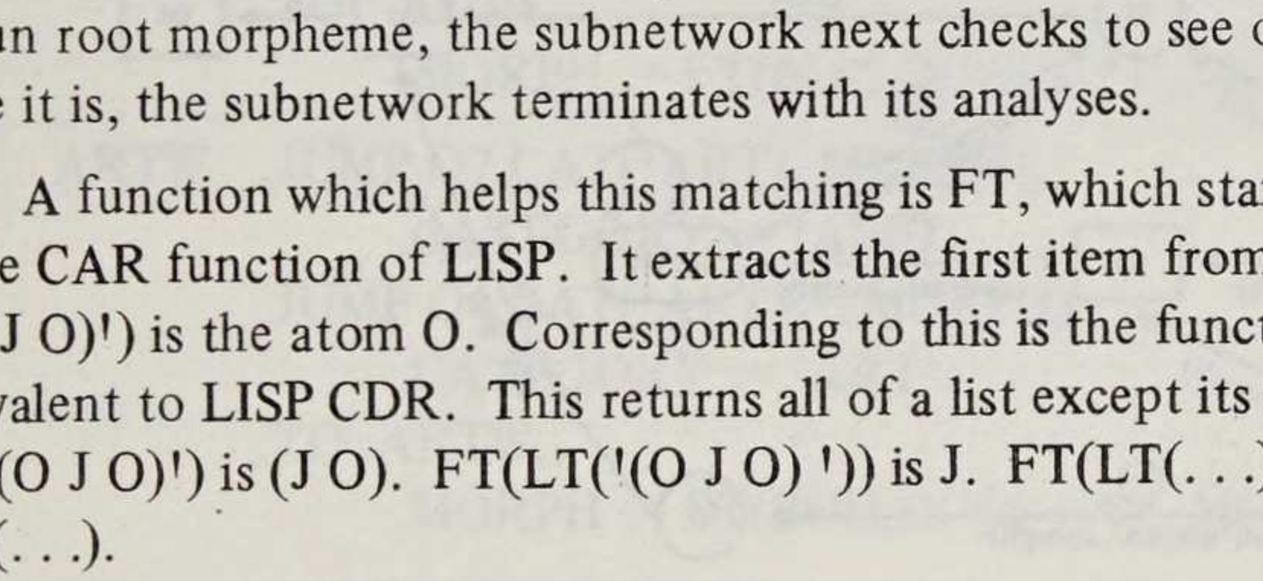
form1	(ADJ (NUC:ROOT ROJ) (PER:GENSFX O))
form2	(ADJ FEATURES (NUM SG GEN MASC))

(l) Q3:1, final state

form	(NP (DET:ART EL) (HEAD:NOUN (NUC:ROOT OJ) (PER:GENSFX O))

(MOD:ADJ (NUC:ROOT ROJ)
(PER:GENSFX O)))

This last form is the final one and corresponds to the following tree:



Another way to handle complex words is by changing the declaration from MORPH to LETTER. In this case, morpheme breaks are not indicated in the input. Instead, when a word level network is referenced, the input provided for it is a list consisting of the current word taken letter by letter. Thus, OJO is split into (O J O).

The letter strategy is to space through the split word from left to right building up a tentative morpheme until a match is found. Thus, for (O J O), the dictionary would be checked to see if O is a noun morpheme. Since it is not, OJ would be checked. Since it is a noun root morpheme, the subnetwork next checks to see of O is a gender morpheme. Since it is, the subnetwork terminates with its analyses.

A function which helps this matching is FT, which stands for 'first', and is equivalent to the CAR function of LISP. It extracts the first item from a list. Thus the value of FT ('(O J O)') is the atom O. Corresponding to this is the function LT which stands for 'last', equivalent to LISP CDR. This returns all of a list except its first element. The value of LT('(O J O)') is (J O). FT(LT('(O J O) ')) is J. FT(LT(. . .)) may be shortened to FLT(. . .).

One action used in the example which follows bears some explanation. It is:

.MORPH := FT(BUILD('(& + *)',.MORPH)

Suppose .MORPH contains RO and the current word is J. Then BUILD('(& + *)', .MORPH) is evaluated as follows. First, the value of .MORPH is substituted for + and the value of the current word for *. BUILD at this point has the value (& RO J). The & takes effect, yielding (ROJ) as BUILD's final value. Then FT is evaluated, yielding ROJ as its value. Thus the effect of the action is to append a letter to the morpheme which is being built up.

Finally, CAT can be used as a function as well as an arc type, and can be incorporated into a test. Given Dictionary 13, CAT('ADJM',.MORPH) will be successful if .MORPH has the value ROJ, and will fail otherwise. CAT as a function can also have a single argument, as with the arc type, in which case the second argument is assumed to be the current word.

Now for the example. Dictionary 13 will be retained. The declarations are:

```
COMPLEX ART ARTW LETTER,
        NOUN NOUNW LETTER,
        ADJ ADJW LETTER;
INITIAL .DICBLD '(+ FEATURES (NUM + GEN +))',
        .STRS  '(+  (NUC:ROOT +)  (PER:GENSFX +))',
        .STRP  '(+  (NUC:ROOT +)  (PER1:GENSFX +)
                    (PER2:PLSFX S))';
STATIC  .DICBLD,.STRS,.STRP;
```

The only difference between these declarations and those of the last example is that LETTER has been substituted for MORPH. The graph and the network itself follow:

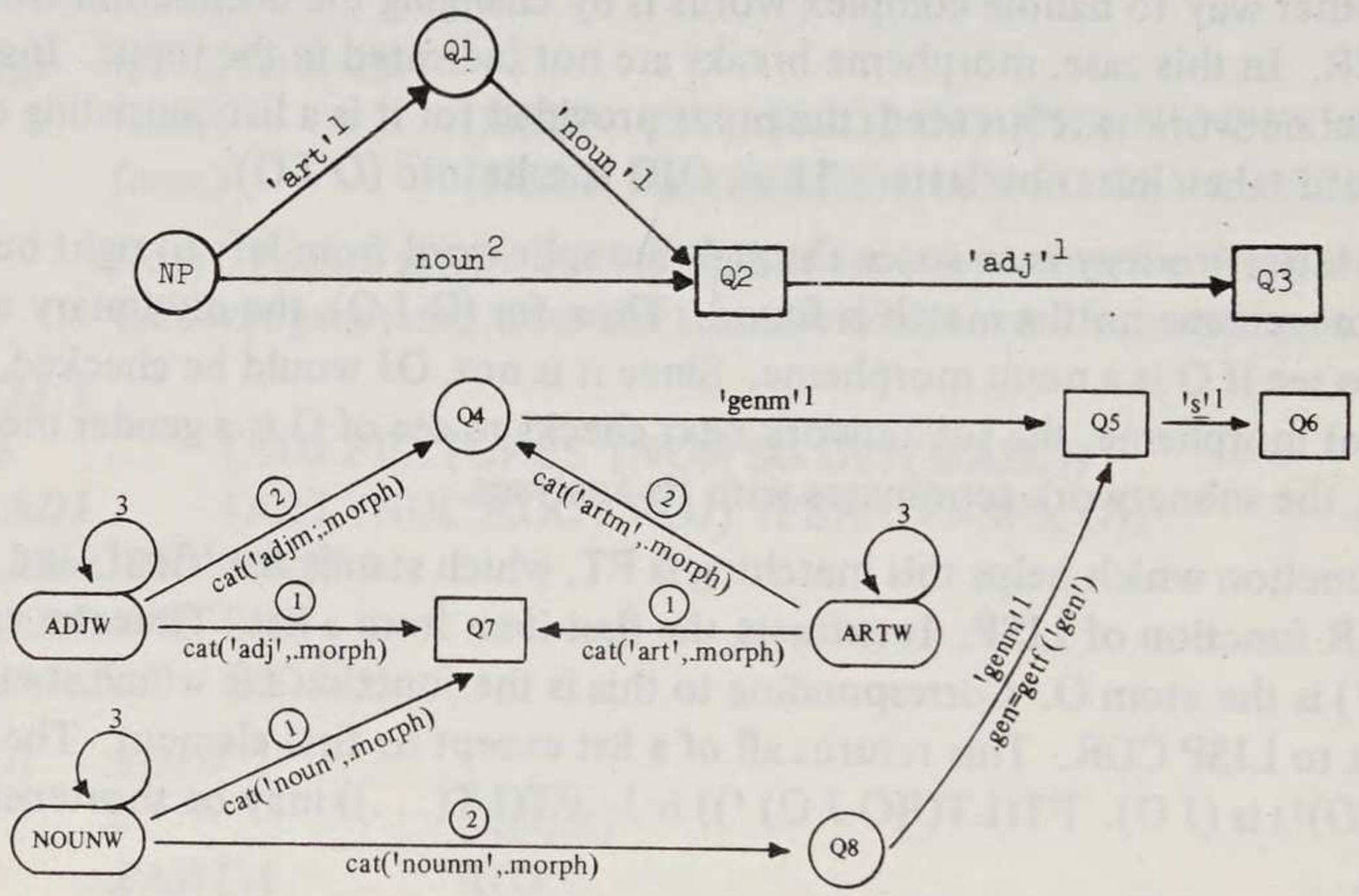


Figure 6

```
NP:  CAT('ART'),TO Q1,,
            .ART := BUILD('(# & DET: *)'),
            .ARTGEN := GETF('GEN'),
            .ARTNUM := GETF('NUM');
     CAT('NOUN'),TO Q2,,
            .NOUN :=*,
            .NOUNGEN := GETF('GEN'),
            .NOUNNUM := GETF('NUM'),
            .ARTGEN := .NOUNGEN,
            .ARTNUM := .NOUNNUM;
```

```
Q1:   CAT('NOUN'),TO Q2,,
            .NOUN   := *,
            .NOUNGEN := GETF('GEN'),
            .NOUNNUM := GETF('NUM');
Q2:   CAT('ADJ'),TO Q3,,
            .ADJ := *,
            .ADJGEN := GETF('GEN'),
            .ADJNUM := GETF('NUM');
      POP   .ARTGEN = .NOUNGEN & .NOUNNUM = .ARTNUM,
            BUILD('(NP + (# & HEAD: +))',.ART,.NOUN);
Q3:   POP   .ARTGEN=.NOUNGEN=.ADJGEN & .ARTNUM=.NOUNNUM=
            .ADJNUM,
            BUILD('(NP + (# & HEAD: +) (# & MOD: +))',
                  .ART,.NOUN,.ADJ);
ADJW:  ,JUMP Q7,CAT('ADJ',.MORPH),
            .CATEGORY := 'ADJ';
       ,JUMP Q4,CAT('ADJM',.MORPH),
            .CATEGORY := 'ADJ';
       ,TO   ADJW,,
            .MORPH := FT(BUILD('(& + *)',.MORPH);
ARTW:  ,JUMP Q7,CAT('ART',.MORPH),
            .CATEGORY := 'ART';
       ,JUMP Q4,CAT('ARTM',.MORPH),
            .CATEGORY := 'ART';
       ,TO  ARTW,,
            .MORPH := FT(BUILD('(& + *)',.MORPH);
Q4:    CAT('GENM'),TO Q5,,
            .GENM := *,
            .GEN := GETF('GEN');
Q5:    WRD('S'),TO Q6;
       POP  ,BUILD(.STRS,.CATEGORY,.MORPH,.GENM),
            BUILD(.DICBLD,.CATEGORY,'SG',.GEN);
Q6:    POP  ,BUILD(.STRP,.CATEGORY,.MORPH,.GENM),
            BUILD(.DICBLD,.CATEGORY,'PL',.GEN);
Q7:    POP  -GETF('IRR'),
            BUILD('(+ +)',.CATEGORY,.MORPH),
            $;
NOUNW:  ,JUMP Q7,CAT('NOUN',.MORPH),
            .CATEGORY := 'NOUN',
```

```
        ,JUMP Q8,CAT('NOUNM',.MORPH),
              .CATEGORY := 'NOUN',
              .GEN := GETF('GEN');
        ,TO NOUNW,
              .MORPH := FT(BUILD('(& + *) ',.MORPH);
Q8:     CAT('GENM'),TO Q5,.GEN = GETF('GEN'),
              .GENM := *;
```

There are a few points which should be noted about this network before progressing. The first is about Arcs ADJW:3, ARTW:3 and NOUNW:3. These arcs have no conditions whatsoever attached to them. These arcs will always be travelled if the preceding ones fail; that is, in the case of NOUNW:3, the arc will be taken if Arcs NOUNW:1 and NOUNW:2 fail. The purpose of this arc, as for ADJW:3 and ARTW:3, is to build up a morpheme.

The second point regards Arcs ADJW:1&2, ARTW:1&2 and NOUNW:1&2. These arcs have no arc type indicated. They will be travelled only if the condition is satisfied. Thus NOUNW:1 will be travelled only if the morpheme .MORPH, built up by NOUNW:3, is listed in the dictionary as a noun. NOUNW:2 will be travelled if it is listed in the dictionary as a noun morpheme.

Finally, regarding these same arcs (ADJW:1&2, ARTW:1&2, NOUNW:1&2), note that each is a JUMP arc rather than a TO arc. In other words, when the arc is travelled, the input string is not advanced to the next word, but remains at the same one. This is used here because these four arcs test the contents of certain registers only, and do not involve the use of the current word.

Now, let us take the same example as was used last time, (EL OJO ROJO). Note that morpheme breaks are not indicated. It now becomes the job of the network to determine these boundaries by stepping through the words letter by letter.

(a) NP:1

```
$          (ART FEATURES (NUM SG GEN MASC))
.ART       (DET:ART EL)
.ARTGEN    MASC
.ARTNUM    SG
STRING     (EL OJO ROJO)
```

Note: The COMPLEX declaration for ART indicates a push to subnetwork ARTW. The input presented to that subnetwork is (E L), that is, the word split into individual characters.

(b) ARTW:3

```
    .MORPH   E
    STRING   (E L)
```

(c) ARTW:3

```
    .MORPH   EL
    STRING   (L)
```

(d) ARTW:1
$ (ART FEATURES (NUM SG GEN MASC))
.CATEGORY ART

(e) Q7:1, final state
form1 (ART EL)
form2 (ART FEATURES (NUM SG GEN MASC))

(f) Q1:1
$ (NOUN FEATURES (NUM SG GEN MASC))
.NOUN (NOUN (NUC:ROOT OJ) (PER:GENSFX O))
.NOUNGEN MASC
.NOUNNUM SG
STRING (OJO ROJO)

(g) NOUNW:3
.MORPH O
STRING (O J O)

(h) NOUNW:3
.MORPH OJ
STRING (J O)

(i) NOUNW:2
.CATEGORY NOUN
.GEN MASC

(j) Q8:1
$ (GENM FEATURES (GEN MASC))
.GENM O
STRING (O)

(k) Q5:2, final state
form1 (NOUN (NUC:ROOT OJ) (PER:GENSFX O))
form2 (NOUN FEATURES (NU SG GEN MASC))

(l) Q2:1
$ (ADJ FEATURES (NUM SG GEN MASC))
.ADJ (ADJ (NUC:ROOT ROJ) (PER:GENSFX O))
.ADJGEN MASC
.ADJNUM SG
STRING (ROJO)

(m) ADJW:3
.MORPH R
STRING (R O J O)

(n) ADJW:3
.MORPH RO
STRING (O J O)

```
(o)  ADJW:3
     .MORPH      ROJ
     STRING      (J O)
(p)  ADJW:2
     .CATEGORY   ADJ
(q)  Q4:1
     $           (GENM FEATURES  (GEN MASC))
     .GENM       O
     .GEN        MASC
     STRING      (O)
(r)  Q5:2, final state
     form1       (ADJ  (NUC:ROOT ROJ)  (PER:GENSFX O))
     form2       (ADJ FEATURES  (NUM SG GEN MASC))
(s)  Q3:1, final state
     form        (NP  (DET:ART EL)  (HEAD:NOUN  (NUC:ROOT OJ)
                 (PER:GENSFX O))  (MOD:ADJ  (NUC:ROOT ROJ)
                 (PER:GENSFX O));
```

This last form is the final one. Note that it is exactly the same as the final form of the preceding example. The tree it defines is the same as the tree for that example. Note too that the NP level network remains unchanged, though the word level networks differ.

8. SUBROUTINES

A subroutine is very similar in general purpose to a subnetwork: both are designed to show up language regularities by eliminating redundancies and condensing similar portions of networks. For instance, go back to Network 10, the linear representation for Spanish noun phrases given in Figure 4. Arcs NP:2 and Q2:1 have the same sequence of actions:

```
.NOUN := *,
.NOUNGEN := GETF('GEN'),
.NOUNNUM := GETF('NUM')
```

We may condense these actions into a subroutine by giving them a name, NOUNSET:

```
NOUNSET =
        .NOUN := *,
        .NOUNGEN := GETF('GEN'),
        .NOUNNUM := GETF('NUM');
```

To use this subroutine, we merely write its name wherever the actions are to be evaluated. Thus we can rewrite Arcs NP:2 and Q1:1 as:

```
NP:  . . .
     CAT('NOUN'),TO Q2,,
          NOUNSET,
          .ARTGEN  := .NOUNGEN,
          .ARTNUM  := .NOUNNUM;
Q1:  CAT('NOUN'),TO Q2,,
          NOUNSET;
```

Thus, each time a reference to NOUNSET is encountered, the three corresponding actions are evaluated.

A subroutine can also be used to eliminate a redundant form. Referring agin to Figure 4, consider Arcs Q2:2 and Q3:1, which are final state POP arcs. These are:

```
Q2:  . . .
     POP  .ARTGEN=.NOUNGEN & .NOUNNUM=.ARTNUM,
          BUILD(' (NP + (HEAD:NOUN +))',.ART,.NOUN);
Q3:  POP  .ARTGEN=.NOUNGEN=.ADJGEN & .ARTNUM=.NOUNNUM=
            .ADJNUM,
          BUILD(' (NP + (HEAD:NOUN +)  (MOD:ADJ +))',.ART,
            .NOUN,.ADJ);
```

We can rewrite these in a way that highlights their parallelism:

16)
```
Q2:  . . .
     POP  .ARTGEN=.NOUNGEN & .NOUNNUM=.ARTNUM,
          BUILD(' (NP + (HEAD:NOUN +))',.ART,.NOUN);
Q3:  POP  .ARTGEN=.NOUNGEN & .ARTNUM=.NOUNNUM=.ADJNUM,
          BUILD(' (NP + (HEAD:NOUN +))',.ART,.NOUN) / /
               BUILD(' ((MOD:ADJ # +))',.ADJ);
```

In this way, the final state forms contain duplicate forms BUILD(' (NP + (HEAD: NOUN +))',.ART,.NOUN)

We can label this form just as we labeled the set of actions previously. Let us call it BLDVAL:

```
BLDVAL = RETURN(BUILD(' (NP + (HEAD:NOUN +))',.ART,.NOUN));
```

The action RETURN used here is a special one, reserved for use in subroutines. It has two purposes. First, it terminates evaluation of the subroutine. Second, its argument (which is optional and assumed to be null if omitted) indicates the value to be associated with the subroutine. Hence the value of BLDVAL will be the value of BUILD(' (NP + (HEAD:NOUN +))',.ART,.NOUN). Using BLDVAL, we may rewrite 16 as follows:

```
Q2:  . . .
     POP  .ARTGEN=.NOUNGEN & .NOUNNUM=.ARTNUM,
          BLDVAL;
```

```
Q3:  POP  .ARTGEN=.NOUNGEN=.ADJGEN & .ARTNUM=.NOUNNUM=
                .ADJNUM,
          BLDVAL / / BUILD(' ((MOD:ADJ +))',.ADJ);
```

It frequently happens that two or more forms (or sets of actions) are identical except for the values they work with. Consider the final state forms of Network 14, Final States Q5:2 and Q6:1. We have the following:

17) Q5:2, form2 -- BUILD(.DICBLD,.CATEGORY,'SG',.GEN)

Q6:1, form2 -- BUILD(.DICBLD,.CATEGORY,'PL',.GEN)

These two forms are identical except that the one has 'SG' where the second has 'PL'. We can condense these two forms into a subroutine as follows:

```
BLDENT(.NUM) = RETURN(BUILD(.DICBLD,.CATEGORY,.NUM,.GEN));
```

Note that BUILD here differs from the two forms of 17 in that the third argument is .NUM. .NUM also appears in parentheses following the subroutine's name, BLDENT. .NUM is called a **formal parameter** and functions as follows. Let us rewrite 17, using a subroutine reference instead. We have:

Q5:2, form2 -- BLDENT('SG')

Q6:1, form2 -- BLDENT('PL')

When, for example, BLDENT('SG') is evaluated, the corresponding subroutine is refered to. The formal parameter, .NUM, is set to the value of the argument in the reference, 'SG'. So when the subroutine is evaluated, .NUM is assumed to have the value SG. Thus, in this case, BUILD(.DICBLD,.CATEGORY,.NUM,.GEN) is evaluated identically to BUILD(.DICBLD,.CATEGORY,'SG',.GEN). Similarly, when BLDENT('PL') is evaluated, .NUM is set to PL and then the BUILD is evaluated.

There may be any number of formal parameters associated with a subroutine. A subroutine reference may not have more arguments than there are formal parameters in the subroutine, but it may have fewer arguments. In this case, the absent arguments are assumed to be null; that is, the corresponding formal parameters are set to null.

It is possible to change an argument value in the course of a subroutine evaluation by the following rule: Suppose Argument **n** of a subroutine reference is the name of a register. Suppose too that the corresponding formal parameter, **n**, appears on the left of an assignment action so that the formal parameter's value is changed. Then the value of the register which is Argument **n** in the call is also changed. For example, consider the following subroutine definition:

```
ADDR( .LIST,.ADDVAL) =
        .LIST := .LIST / / BUILD(' (+)',.ADDVAL);
```

Suppose register .ADJS has as its value (BIG RED). Then consider the following subroutine reference:

ADDR(.ADJS,'EXPENSIVE')

First of all, the value of .ADJS, which is (BIG RED), is set as the value of formal parameter .LIST. Then 'EXPENSIVE' is set as the value of formal parameter .ADDVAL. The subroutine's action is then evaluated. The value of .LIST is (BIG RED). The value of BUILD(' (+)',.ADDVAL) is (EXPENSIVE). Thus the value of the form is (BIG RED EXPENSIVE). This value is set into .LIST. But note that the argument in the reference corresponding to .LIST in the definition is a register .ADJS. Thus, by the above rule, since the value of .LIST is changed in the definition, so is the value of .ADJS in the actual use of the subroutine. The final result of this subroutine reference then is to set .ADJS to (BIG RED EXPENSIVE).

Throughout many of the examples given in previous sections, we have made use of GETF, which was mentioned as being a subroutine. The subroutine is shown below.

```
GETF(.FEATURE) =                                                    (1)
    .SUBENT := $,                                                   (2)
  LOOP1:  .SUBENT  := LT(.SUBENT),                                  (3)
    IF  -.SUBENT THEN RETURN,                                       (4)
    IF  -(FT(.SUBENT)='FEATURES') THEN GO LOOP1,                    (5)
    .SUBENT  := FLT(.SUBENT),                                       (6)
  LOOP2:  IF FT(.SUBENT)=.FEATURE THEN                              (7)
          RETURN(FLT(.SUBENT)),
    .SUBENT  := LT(.SUBENT),                                        (8)
    IF  .SUBENT THEN GO LOOP2                                       (9)
                ELSE RETURN;
```

The numbers to the right are for reference and are not part of the subroutine. There are three things to be noted before an example is given. The first is the minus sign (-) used in Lines (4) and (5) as a negator. Suppose in Line (4) that .SUBENT had a null value. Then -.SUBENT would have as its value the atom T. Suppose on the other hand that .SUBENT had some non-null value. Then -.SUBENT would have a null value.

The second point concerns the use of the IF action used in Lines (4), (5), (7) and (9). The IF action has two formats:

```
IF  form  THEN  action,
IF  form  THEN  action1
          ELSE  action2,
```

For both formats, suppose the form indicated has a non-null value. Then the action following THEN is evaluated. Now suppose the form has a null value. For the first format, no action is evaluated; the IF action in this case has no effect. For the second format, action2, following ELSE, is evaluated. Consider Line (4) of the subroutine. If the value of -.SUBENT is null, then nothing happens; Line (5) is evaluated next. If the value of -.SUBENT is non-null, then the subroutine terminates with the RETURN, giving a null value to the subroutine reference. Now consider Line (9). If .SUBENT has a non-null value, then the GO action (discussed below) is evaluated. If .SUBENT has a

null value, then the subroutine terminates with the RETURN, giving a null value. IF may be used in subroutines or in regular networks.

The third point is the GO action, which, like RETURN, is reserved for use in subroutines. The GO action is used in Lines (5) and (9). Consider Line (9) where GO LOOP2 appears. Looking back to Line (7) we see that the line begins with LOOP2:. The effect of GO LOOP2 is to cause subroutine evaluation to continue from the action labeled LOOP2:. Thus whenever GO LOOP2 is evaluated, evaluation continues from Line (7). LOOP2 is a label which in no way affects the evaluation of the line to which it is prefixed. When Line (6) is evaluated, for example, Line (7) evaluates without any regard for the label attached.

As an example, suppose the current word were EL. By Dictionary 13, $ would have as its value (ART FEATURES (NUM SG GEN MASC)). Now suppose that in the course of evaluation GETF('GEN') were encountered. GETF would be evaluated as follows:

(a) On beginning evaluation of GETF, the formal parameter .FEATURE would be set to GEN.

(b) Action (2) sets Register .SUBENT to the value of $, (ART FEATURES (NUM SG GEN MASC)).

(c) Action (3) has a label LOOP1, which is referred to in Action (5). .SUBENT has the value (ART FEATURES (NUM SG GEN MASC)). The last part of this is (FEATURES (NUM SG GEN MASC)). Thus .SUBENT is set to (FEATURES (NUM SG GEN MASC)).

(d) Action (4) is evaluated as follows: .SUBENT has the value (FEATURES (NUM SG GEN MASC)), a non-null value. Consequently, -.SUBENT has a null value. So Action (4) has no effect.

(e) Action (5) tests to see if the feature list has come up yet by comparing the first part of the value of .SUBENT against the atom FEATURES. Since FT(.SUBENT) has the value FEATURES, the value of FT(.SUBENT)= 'FEATURES' is the atom T, a non-null value. The value then of -(FT (.SUBENT)='FEATURES') is null and Action (5) has no effect.

(f) Action (6) extracts the feature list itself. The last part of the value of .SUBENT is ((NUM SG GEN MASC)). The first part of that is (NUM SG GEN MASC). Thus FLT(.SUBENT) has the value (NUM SG GEN MASC) which is set as the new value of .SUBENT.

(g) Action (7) checks whether the value of the first part of .SUBENT, here NUM, is the same as the feature specified, GEN. Since it is not, Action (7) has no effect.

(h) Action (8) advances the feature list by setting .SUBENT to the value of the last part of itself. The new value of .SUBENT is (SG GEN MASC).

(i) Action (9) makes sure .SUBENT is not null, which would mean that the feature specified did not appear. Since .SUBENT is not null, GO LOOP2 is evaluated and evaluation commences from Line (7).

(j) Again, Action (7) tests whether the specified feature has come up yet. Since the value of FT(.SUBENT) is SG, Action (7) has no effect.

(k) Action (8) again advances .SUBENT, setting it to (GEN MASC).

(l) Since .SUBENT still has a non-null value, evaluation continues from Line (7).

(m) At last the value of FT(.SUBENT), GEN, is the same as the value of .FEATURE. Consequently the RETURN is evaluated. The argument of the RETURN is FLT(.SUBENT). Since LT(.SUBENT) is (MASC), FLT(.SUBENT) has the value MASC. This is returned as the value of the subroutine.

For our example, then, the value of GETF('GEN') is MASC. The reader may wish to trace the subroutine for the following values of $ and references to GETF:

```
$ -- (GENM FEATURES (GEN FEM))
GETF('NUM')
(This will have a null value since there is no NUM feature.)

$ -- (ART FEATURES (NUM SG GEN MASC))
GETF('NUM')
(The value of GETF here will be SG.)

$ -- (ART FEATURE (NUM SG GEN MASC))
GETF('NUM')
(The value of GETF here will be null due to the spelling error.)
```

A subroutine may reference another subroutine, just as an arc action or form may reference a subroutine. A subroutine may also reference itself recursively. In this case, all non-static registers and all formal parameters are pushed on entry and popped on return.

One last item in connection with subroutines concerns the scope of register and formal parameter names. Consider the modified version of ADDR along with a new, frequently useful, subroutine, ADDPAREN:

```
ADDR(.LIST,.ADDVAL) =
        .LIST := .LIST / / ADDPAREN(.ADDVAL);
ADDPAREN(.ADDVAL) =
        RETURN(BUILD(' (+)',.ADDVAL));
```

ADDR performs the same function as it did before, but instead it references Subroutine ADDPAREN. Notice, however, that ADDR and ADDPAREN each have a formal parameter by the name of .ADDVAL. To avoid the problems caused by this, we adopt the convention that all formal parameters in a subroutine are assumed to be local in scope. This means that in the preceding subroutines there are actually three formal

parameters: .LIST in ADDR, .ADDVAL in ADDR and .ADDVAL in ADDPAREN. In one subroutine the name .ADDVAL refers to one formal parameter and in another subroutine it refers to another formal parameter, despite the fact that their names are composed of the same letters.

Similar problems may occur with regard to register names and so we adopt the same convention for register names: all registers used in a subroutine are assumed to be local in scope.

A natural question is: For subroutines such as 15, NOUNSET, how may registers such as .NOUN, .NOUNGEN and .NOUNNUM be referred to outside the subroutine? The answer is that the declaration GLOBAL is used. For example, we can write:

```
GLOBAL .NOUN,.NOUNGEN,.NOUNNUM;
```

This declaration reverses the assumption that the registers so named are local in scope. It makes these registers global in scope. This means that any occurrence of .NOUNGEN, for instance, anywhere in the network (including in every subroutine) refers to the same register. Thus information set into .NOUNGEN is available elsewhere in the network, just as registers that are declared STATIC in networks are accessible at all levels.

Similarly, all action labels are local in scope. Neither action labels nor formal parameters in subroutines may be declared GLOBAL.

9. AN ADDITIONAL NOTE ON SUBNETWORK INTERCOMMUNICATION

In Section 5, means of communicating information between a subnetwork and its reference were discussed, involving the use of $ as in .$REG. There is another way, which is analogous to a subroutine and its reference. Let us write a network fragment in this way and then explain it.

```
. . .   . . .
        PUSH(SPP(' (TO FROM) ')),TO Q4,,
                .PP := *;

        . . .
        PUSH(SPP('WITH')),TO Q9,,
                .PP := *;
PP:     CAT('PREP'),TO Q13,,
                .PREPW := *;
SPP(.PREP): WRD(.PREP),TO Q13,,
                        .PREPW := *;
Q13: PUSH(NP),TO Q14,,
                .NP := *;
Q14: POP  ,BUILD(' (PP (LOC:PREP +)  (# & HEAD: +))',
                        .PREPW,.NP);
```

.PREP here is a formal parameter since it appears in parentheses following the state name SPP. When the first PUSH is evaluated, the argument associated with the subnetwork reference is assigned as the value of .PREP. Thus, .PREP takes on the value of (TO FROM). The arc WRD(.PREP) is evaluated just as though it were WRD('(TO FROM)'). Similarly, with the second PUSH, .PREP takes on the value WITH.

Upward communication can also be done this way with a rule similar to that for subroutines: if a subnetwork reference argument is just a register and if the corresponding formal parameter has its value changed during subnetwork evaluation then the value of the register mentioned is also changed.

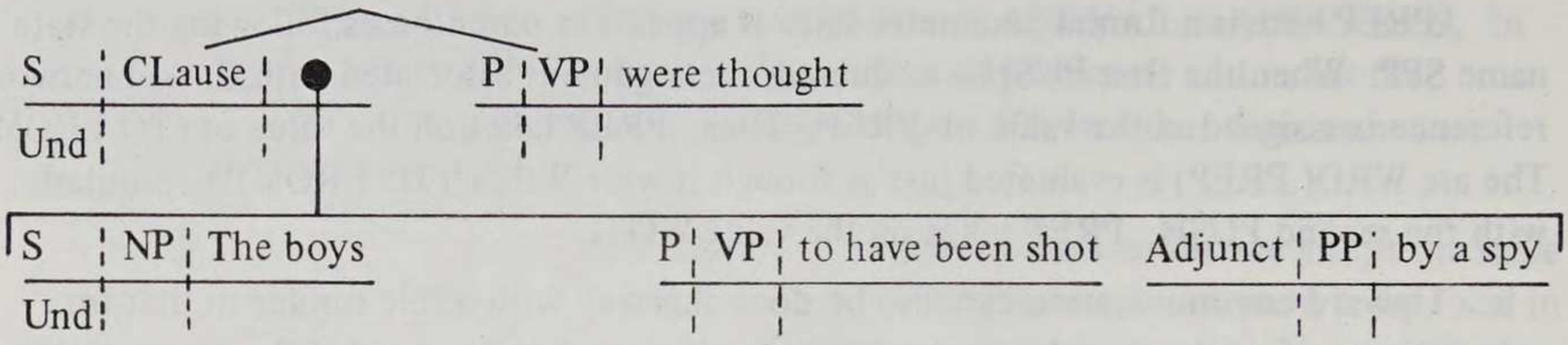
10. AN EXAMPLE FROM ENGLISH

As an illustration of many of the things we have been discussing, a network for a subset of English grammar is given here. It transduces and gives a tagmemic analysis for a significant portion of English, though by no means for all. Among the input it will transduce are such clauses as:

I see John.
I will see John.
Will I see John?
He had gone
He will have been going.
He was thought to have gone.
The man was thought to have eaten.
In the yellow car the boys were thought to have been shot by a spy
with a camera.

Certain erroneous clauses will also be accepted since, for the sake of simplicity, subject-predicate agreement has not been included. Woods (1969) gives a much more complete network for English, though his analysis is transformational rather than tagmemic, and his TNG formalism is different from the one used here.

In developing this example, an effort was made to convert the network provided by Woods (1970) directly into one which would report its analysis in display grammar form (Hale 1973), bearing in mind the fact that one of the goals of a tagmemic analysis is to provide a tree representing the surface structure. However, problems of skewing developed. Consider, for example, *The boys were thought to have been shot by a spy*. The main verb for the sentence is *were thought* and the apparent subject is *the boys*, with the balance a sentential complement. It is clear, however, that *The boys were thought* is not the same as, for instance, *The boys were hit*. Whereas in the latter case we can say that what was hit were the boys, we cannot say in the former case that what was thought were the boys, despite the subject-predicate agreement. What we must say is that what was thought was that the boys were shot by a spy. Hence the subject of *was thought* is not *the boys* but is instead *the boys were shot by a spy*. A rough schematic representation of this in Hale's display grammar form is shown below:

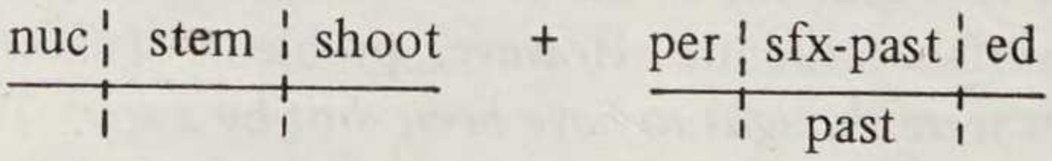


The subject of the complement sentence is discontinuous with it. It is therefore not possible to construct a surface tree without reordering the constituents.

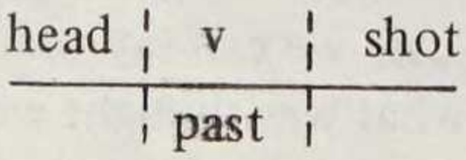
However, this sort of situation is not fatal to display grammar, which guarantees a surface tree analysis only for underived or basic forms. Derivations commonly introduce skewing from the normal pattern and consequently may distort the underlying tree. To resolve the situation for English, then, it becomes necessary to derive back to the underlying form. Consequently, *The boys were thought to have been shot by a spy* is first converted to *Someone thought the boys had been shot by a spy* and then finally to *Someone thought a spy had shot the boys.* An important point to note is that each derivation produces a pronouncable grammatical utterance, in contrast to the transformational approach where intermediate transformations may well produce structures which are unutterable. The underlying form for display grammar then is not equivalent to deep structure, but lies much closer to the surface.

It turned out that Woods's network (1970) has precisely the mechanisms needed to inversely derive a sentence back to its underlying form, though the analysis returned by Woods's network is a transformational one in which, among other things, verb tense is treated as a clitic. The network given here produces a display grammar type of analysis of the basic underived version of the input sentence. In addition, a somewhat elaborate verb phrase has been included in the network which prevents certain erroneous transductions besides admitting future tense and progressive forms.

A certain problem developed concerning the representation of irregular forms, an area in which tagmemics has had surprisingly little to say in the past. There seem to be at least two ways to represent an irregular verb such as *shot* in a tree analysis. One way is to consider *shot* to be *shoot* plus past tense, representing this as:



The other way, which the network uses, is to treat it as a complex unit:



The usefulness of TNG as a notation scheme for tagmemics was nicely demonstrated by these problems as the explicit detail which TNG requires forced them to be considered,

as well as providing a mechanism to implement the solutions.

In the analyses actually produced by the network, the six box form of a Hale display matrix has been permuted and linearized. Here:

$\begin{array}{c|c|c} 1 & 2 & 3 \\ \hline 4 & 5 & 6 \end{array}$ is permuted to $\begin{array}{c|c} 1 & 4 \\ \hline 2 & 5 \\ \hline 3 & 6 \end{array}$ and is linearized as 1,4:2,5:3,6.

The reason for this is that a subnetwork such as NP is able to provide the contents of boxes 2,5,3, and 6 whereas the contents of boxes 1 and 4 are provided by the referencing network. So an NP subnetwork might produce an analysis such as (NP,anim (det, . . .) . . .). Then the referencing network would add the box 1 and 4 information to the head of the tree, converting it to, for instance, (S,Act:NP,anim (det, . . .) . . .).

Not all the boxes always have information filling them. Blank boxes are filled in with a dollar sign, $.

Seven categories of words have been entered in the dictionary for this grammar: verbs (v), nouns (n), pronouns (prn), proper nouns (npr), articles (art), adjectives (adj) and prepositions (prep). Some words, such as *bill*, are of more than one category.

There are two kinds of verb entries. The first is what may be called a root form, and is as follows:

(definiens (V FEATURES (features) X(exclusion)
TAGS (tags)))

The FEATURES list includes those features which are invariant despite the inflection of the verb. The X list, which is optional, lists those regular inflections which are not permitted. The TAGS list gives those features peculiar to the particular form shown. For example:

(WILL (V (FEATURES (TRANS) TAGS (AUX TNS PRES UNTENSED)))
(HAVE (V FEATURES (SCOMP TRANS) X (PAST PROG PRES)
TAGS (AUX TNS PRES UNTENSED)))
(SHOOT (V FEATURES (TRANS INTRANS) X(PAST) TAGS
(TNS PRES UNTENSED)))

The FEATURES list for WILL shows that the verb is transitive. This is true despite the inflection (*I will the book, I willed the book, He wills the book*). The absence of an X list indicates that all regular inflections (past, third person singular present, progressive) may occur. The TAGS list shows that in the uninflected form, WILL may be an auxiliary, has present tense, and is in the untensed form required for certain verb phrase constructions (such as *I do will the book*). The features in the TAGS list do not in general occur with the inflected forms. For instance, WILLS is not an auxiliary (though it is present tense), nor is it untensed. Similarly, WILLING is not an auxiliary, nor is it present tense, since there is no inherent tense for a progressive form, nor is it untensed.

The FEATURES list for HAVE shows that the verb may take a sentential

complement (*I have the hot dog to eat*), and is transitive. The X list indicates that the regular inflections for past, progressive, and third person singular present do not apply. Thus we may not have instances of HAVES, HAVED, or HAVEING (the latter because we are using orthographic conventions rather than phonemic, and because orthographemic reduction rules have not been included). The TAGS list shows that the tense of HAVE is present, may be an auxiliary, and is an untensed form.

The SHOOT feature list shows that the verb may be either intransitive (*I am shooting*) or transitive (*I am shooting the bear*). The X list indicates that past tense is irregular. Thus we can have SHOOTS and SHOOTING but not SHOOTED. The TAGS list shows that SHOOT is present tense and is in untensed form.

The second kind of verb entry is what may be called an irregular form. These represent non-regular inflections. The form is as follows:

(definiens (V FEATURES (ROOT root) TAGS (tags)))

The ROOT argument refers to the root form which gives the invariant features. The TAGS list gives the features peculiar to the invariant form listed. For example:

```
(HAS  (V FEATURES  (ROOT HAVE) TAGS  (AUX TNS PRES)))
(HAD (V FEATURES  (ROOT HAVE) TAGS  (AUX PERF TNS PAST)))
(HAVING  (V FEATURES  (ROOT HAVE)  (TAGS  (PROG)))
(SHOT  (V FEATURES  (ROOT SHOOT) TAGS  (PERF TNS PAST)))
```

The ROOT argument for HAS, HAD, and HAVING is the verb HAVE. The FEATURES list for HAVE, shown earlier, shows that these three forms may take a sentence complement and are transitive. The TAGS lists for these forms give the features specific to each form. The situation is similar for SHOT. If an entry has ROOT in it, it cannot be inflected; it is treated as an invariant form. Consequently, an X list is not required.

For nouns and pronouns, the situation is similar. For example,

```
(WOMAN   (N FEATURES  (POT ANIM)  X(PL)))
(WOMEN   (N FEATURES  (ROOT WOMAN)  TAGS  (NUM PL)))
```

For nouns, pronouns, and proper nouns, the number is assumed to be SG unless otherwise included. Adjectives have no features associated with them. Articles simple specify whether they are definite or indefinite. Prepositions specify type (locative, directional, etc.). The dictionary itself is shown below:

```
(A        (ART FEATURES  (REF INDEF)))
(AM       (V FEATURES  (ROOT BE) TAGS (AUX TNS PRES)))
(ARE      (V FEATURES  (ROOT BE) TAGS (AUX TNS PRES)))
(AUSTIN   (NPR FEATURES  (POT ANIM)))
(BE       (V FEATURES  (TRANS INTRANS) X (PAST PRES)
                  TAGS  (AUX UNTENSED)))
(BEEN     (V FEATURES  (ROOT BE) TAGS (PERF)))
```

```
(BELIEVE (V FEATURES (SCOMP TRANS) X (PROG)
                TAGS (TNS PRES UNTENSED)))
(BELIEVING (V FEATURES (ROOT BELIEVE) TAGS (PROG)))
(BILL      (NPR FEATURES (POT ANIM))
           (N FEATURES (POT INANIM))
           (V FEATURES (TRANS) TAGS (UNTENSED TNS PRES)))
(BOY       (N FEATURES (POT ANIM)))
(CAMERA (N FEATURES (POT INANIM)))
(CONSIDER (V FEATURES (SCOMP TRANS INTRANS) TAGS (TNS
          PRES UNTENSED)))
(DID       (V FEATURES (ROOT DO) TAGS (AUX TNS PAST)))
(DO        (V FEATURES (TRANS) X (PAST) TAGS (AUX TNS
                PRES UNTENSED)))
(DONE      (V FEATURES (ROOT DO) TAGS (PERF)))
(GIRL      (N FEATURES (POT ANIM)))
(HAD       (V FEATURES (ROOT HAVE) TAGS (AUX PERF TNS PAST)))
(HAS       (V FEATURES (ROOT HAVE) TAGS (AUX TNS PRES)))
(HAVE      (V FEATURES (SCOMP TRANS) X (PAST PROG PRES)
                TAGS (TNS PRES UNTENSED)))
(HAVING    (V FEATURES ROOT HAVE) TAGS (PROG)))
(HE        (PRN FEATURES (POT ANIM)))
(HER       (PRN FEATURES (ROOT SHE)))
(HIM       (PRN FEATURES (ROOT HE)))
(I         (PRN FEATURES (POT ANIM)))
(IN        (PREP FEATURES (TYP LOC)))
(IS        (V FEATURES (ROOT BE TAGS (AUX TNS PRES)))
(IT        (PRN FEATURES (ROOT BE) TAGS (AUX TNS PRES)))
(JOE       (NPR FEATURES (POT ANIM)))
(JOHN      (NPR FEATURES (POT ANIM)))
(MAN       (N FEATURES (POT ANIM) X (PL)))
(ME        (PRN FEATURES (ROOT I)))
(MEN       (N FEATURES (ROOT MAN) TAGS (NUM PL)))
(ON        (PREP FEATURES (TYP LOC)))
(PLEASANT  (ADJ))
(SAW       (N FEATURES (POT INANIM))
           (V FEATURES (ROOT SEE) TAGS (TNS PAST)))
(SEE       (V FEATURES (TRANS INTRANS) X (PAST) TAGS
                (TNS PRES UNTENSED)))
(SEEN      (V FEATURES (ROOT SEE) TAGS (PERF)))
(SHE       (PRN FEATURES (POT ANIM)))
(SHOOT     (V FEATURES (TRANS INTRANS) X (PAST) TAGS
                (TNS PRES UNTENSED)))
```

```
(SHOT       (V FEATURES (ROOT SHOOT) TAGS (PERF TNS PAST)))
(SPY        (N FEATURES (POT ANIM)))
(THE        (ART FEATURES (REF DEF)))
(THEM       (PRN FEATURES (ROOT THEY)))
(THEY       (PRN FEATURES (POT ANIM/INANIM NUM PL)))
(TO         (PREP FEATURES (TYP DIR)))
(US         (PRN FEATURES (ROOT WE)))
(WAS        (V FEATURES (ROOT BE) TAGS (TNS PAST)))
(WE         (PRN FEATURES (POT ANIM NUM PL)))
(WERE       (V FEATURES (ROOT BE) TAGS (TNS PAST)))
(WILL       (V FEATURES (TRANS) TAGS (AUX TNS PRES UNTESNED)))
(WITH       (PREP FEATURES (TYP INS)))
(WOMAN      (N FEATURES (POT ANIM X (PL)))
(WOMEN      (N FEATURES (ROOT WOMAN)))
(YELLOW     (ADJ))
(YOU        (PRN FEATURES (POT ANIM NUM SG/PL)))
```

The network itself, in linear and graphic form, is shown below. First the declarations are given, then the subroutines, then the network proper, with comments on it given there.

Register .NIL is a register whose value, in this network, is always null. Similarly, register .T contains the atom 'T' throughout the network. However, register .FRSTSW is set to null after the first verb or auxiliary is found.

A comment is a list whose first element is an asterisk. An example is (* THIS IS A COMMENT). Comments may be used freely throughout a network. A comment may precede or follow a subroutine, an action, or a label.

```
GLOBAL .MODS, .PREPTYP, .NIL, .T, .FEATLIST;
STATIC .ROOT, .ROOT2 .TRANS, .INTRANS, .SCOMP, .NIL, .T, .VW, .FRSTSW;
COMPLEX N NW/ LETTER,
         V VW/ LETTER;
INITIAL .T 'T',.FRSTSW 'T';

(* ADDSMOD ADDS A CLAUSE LEVEL MODIFIER -- A PREPOSITIONAL
     PHRASE -- TO THE LIST .MODS.)
ADDSMOD =
     .MODS := .MODS / / BUILD('(# & CLMOD, + : *))', .PREPTYP);

(* ADDPAREN CONSTRUCTS A LIST WHOSE ONLY ELEMENT IS THE ARGUMENT VALUE.)
ADDPAREN(.VAL) =
     RETURN(BUILD!(+)', .VAL));
```

```
(* BLDMORPH ADDS A CHARACTER TO THE SPECIFIED MORPHEME.)
BLDMORPH(.MORPH) =
      RETURN(BUILD('(# & + *)', .MORPH)

(* FINDL INSPECTS THE INDICATED LIST FOR AN ATOM WHICH MATCHES THE LABEL
   PROVIDED.  THE VALUE OF FINDL IS THAT SUBLIST BEGINNING WITH THE
   INDICATED LABEL, OR NULL IF THE LABEL DOES NOT OCCUR.  THE LIST IS
   ASSUMED TO BE LT($) IF NOT SPECIFIED.)
FINDL(.LABEL, .LIST) =
      IF .LIST THEN .SUBENT :=  .LIST
               ELSE  .SUBENT :=  LT($), (* CHECK FOR DEFAULT)
   LOOP:  IF –.SUBENT THEN RETURN, (* CHECK FOR END OF LIST)
      IF FT(.SUBENT)=.LABEL THEN RETURN(.SUBENT), (* SEE IF FOUND)
      .SUBENT := LT(.SUBENT), (* ADVANCE THE LIST)
      GO LOOP; (* TRY AGAIN)

(* GETF RETURNS THE FEATURE VALUE SPECIFIED FROM THE LIST SPECIFIED.  IF
   UNSPECIFIED, THE LIST IS ASSUMED TO BE THE FEATURES LIST OF $.)
GETF(.FEAT,.LIST) =
      RETURN(FLT(FINDL(.FEAT,IF(.LIST,,FEATURES))));

(* CKF RETURNS THE FEATURE SPECIFIED FROM THE LIST SPECIFIED.  IF UNSPECIFIED,
   THE LIST IS ASSUMED TO BE THE FEATURES LIST OF $.  CKF DIFFERS FROM GETF IN
   THAT THERE IS NO VALUE ASSOCIATED WITH THE FEATURE.)
CKF(.FEAT,.LIST) =
      RETURN(FT(FINDL(.FEAT,IF(.LIST,,FEATURES))));

(* FEATURES RETURNS THE ENTIRE FEATURE LIST, IF PRESENT)
FEATURES =
      RETURN(FLT(FINDL('FEATURES')));

(* ADDF ADDS THE SPECIFIED VERB FEATURE, IF PRESENT, TO FEATLIST, AND RETURNS
   THE FEATURE NAME, IF PRESENT.)
ADDF(.FEAT) =
      IF CKF(.FEAT) THEN IF .FEATLIST THEN .FEATLIST := .FEATLIST / /
                                       BUILD('(& – +)',.FEAT)
                                          ELSE .FEATLIST := ADDPAREN(.FEAT),
      RETURN(.FEAT);

(* ADDLET ADDS THE LETTER IN * TO .WRD.)
ADDLET(.WRD) =
      .WRD := BLDMORPH(.WRD);
```

```
(* ROOT CAUSES $ TO BE RESET TO THE SPECIFIED ROOT, IF NECESSARY, AND RETURNS
   THE ROOT WORD, WHICH IS THE CURRENT WORD IF $ DOES NOT SPECIFY A ROOT.)
ROOT =
      .RVAL := GETF('ROOT'), (* SEE IF A ROOT IS SPECIFIED)
      IF .RVAL THEN (IF –CAT(FT($),.RVAL) THEN ABORT)
                ELSE .RVAL := *, (* RESET $ IF NECESSARY, ELSE SET ROOT TO THE
                                    CURRENT WORD)
      RETURN(.RVAL);
(* WRD IS SIMILAR TO THE WRD ARC EXCEPT IT IS USED AS A SUBROUTINE.  IT MAY BE
   DOUBLE ARGUMENTED, ALLOWING A TEST WORD TO BE CHECKED AGAINST A LIST
   OF WORDS.  IF SINGLE ARGUMENTED, THE TEST WORD IS ASSUMED TO BE THE
   CURRENT WORD –– I.E., FT(STRING).)
WRD(.WRDS,.TSTWRD) =
      IF –.TSTWRD THEN .TSTWRD := FT(STRING), (* TEST FOR DEFAULT)
      IF ATOM(.WRDS) THEN RETURN(.WRDS=.TSTWRD),
      .SCRA := .WRDS, (* SET UP A SCRATCH LIST)
   LOOP:  IF –.SCRA THEN RETURN, (* IF NOT FOUND THEN RETURN NULL)
      IF FT(.SCRA) = .TSTWRD THEN RETURN(.SCRA), (* RETURN THE SUBLIST)
      .SCRA := LT(.SCRA), (* ADVANCE THE LIST)
      GO LOOP;

(* IF TESTS THE VALUE OF .TST.  IF IT IS NON-NULL THEN THE VALUE OF IF IS .THEN;
   OTHERWISE THE VALUE IS .ELSE.)
IF(.TST,.THEN,.ELSE) =
      IF .TST THEN IF .THEN THEN RETURN(.THEN)
                            ELSE RETURN(.TST)
               ELSE RETURN(.ELSE);
```

The network itself is now shown, in both graphic and linear forms. Regarding the graphic representation in Figure 7, for some subnetworks it is convenient to show the conditions for each arc on the arc itself. Subnetworks VW/ and NW/ are written this way. For the other subnetworks, it is more convenient to show the conditions along with the actions.

Subnetwork CL/ transduces either a declarative or an interrogative clause, either of which may be transitive or intransitive. Clause level modifiers consisting of prepositional phrases may appear initially, following the object of a transitive clause, following the object of a transitive clause, or following a sentential complement.

Aside from initial prepositional phrases, a clause is considered interrogative if it begins with an auxiliary verb (*Did he go?*) and declarative if it begins with a noun phrase (*He did go.*). The CAT('V') arcs leading from state CL/VP handle the various permitted auxiliaries whose roots are DO, WILL, HAVE and BE. If a clause is passive, then arc CL/VP:4 will cause the assumed subject to be entered on the HOLD list (from which it

will be removed by the virtual arc, CL/VP:5), and the indefinite pronoun SOMEONE to be set as subject. Arcs CL/VP:5 & 6 set the noun phrase transduced to the object. Arc VP/NP:1 tests for the agent of a passive clause. Arc VP/NP:3 tests for a sentential complement (*He was believed to have gone.*).

Final state VP/VP may be taken if the main verb of the predicate is tagged INTRANS in the dictionary. Final states VP/NP and VP/SPREP may be taken if the verb is marked TRANS in the dictionary. Final state VP/SPREP will be taken only if the clause is passive with an agent or if it takes a sentential complement.

Subnetwork VP/ transduces the verb phrase portion of a sentential complement and, aside from arc VP/:1, follows the CL/ subnetwork from state CL/VP.

Subnetwork NP/ transduces the following noun phrase types:

NP = ± Det:art ± Mod:adj* + Head:noun ± NPmod:PP
NP = + Head:npr
NP = + Head:prn

Subnetwork PP/ transduces the following prepositional phrase type:

PP = + (typ):prep + Head:NP

Here, (typ) represents the preposition type. For instance, TO is directional, IN is locative, etc.

Subnetwork NW/ transduces a nojn word as represented by the following formulae:

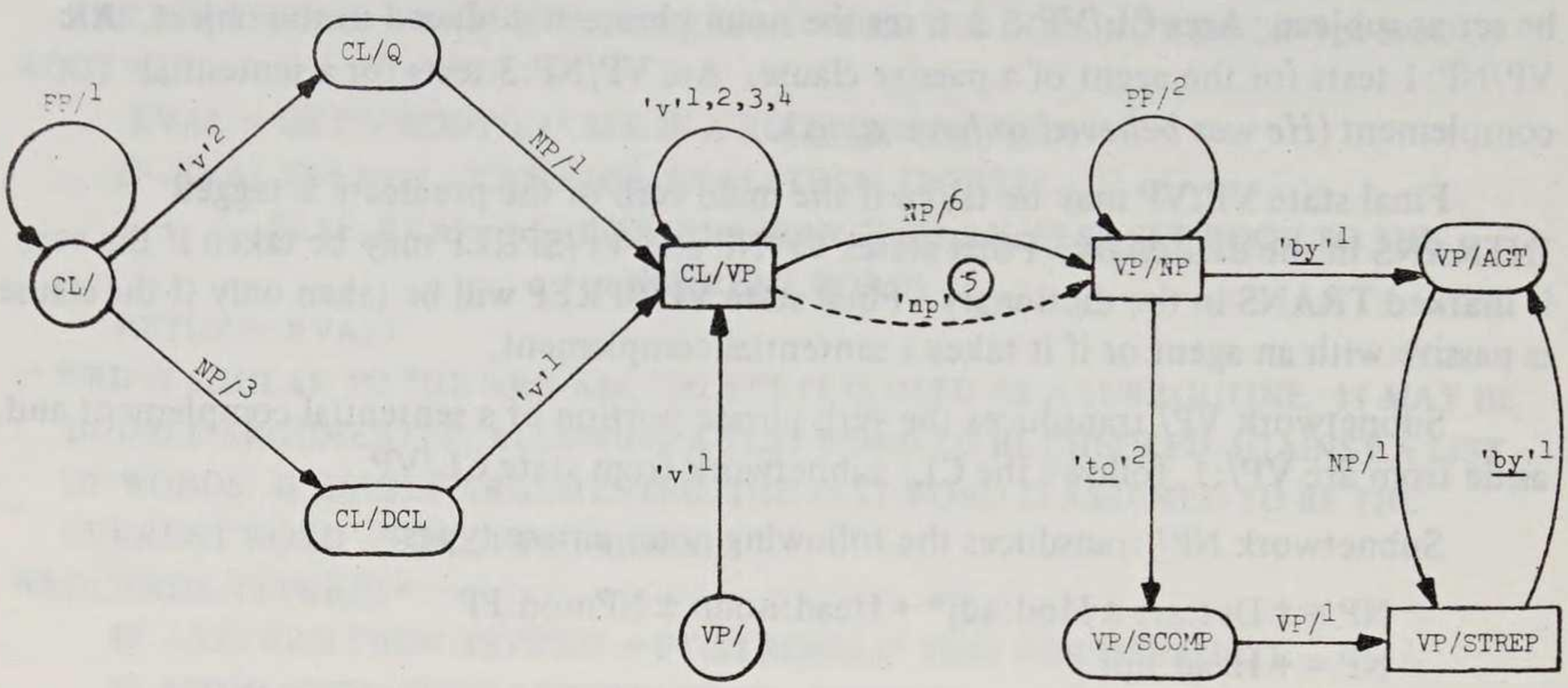
noun = + Nuc:stem ± Per:sfxl
sfxl: **s** 'plural'
es 'plural'

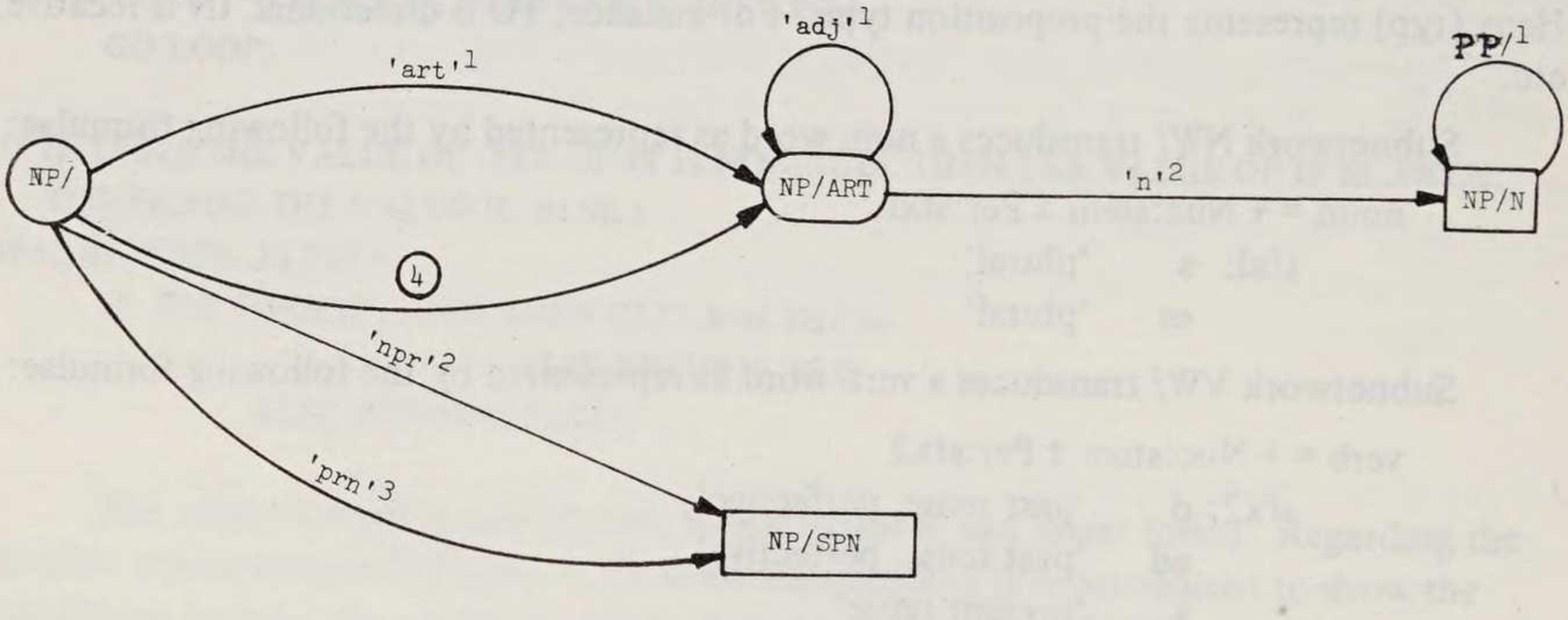
Subnetwork VW/ transduces a verb word as represented by the following formulae:

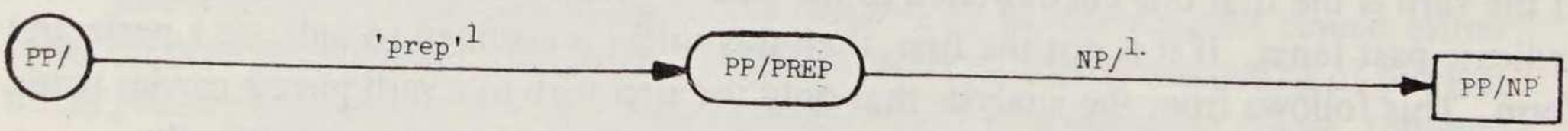
verb = + Nuc:stem ± Per:sfx2
sfx2: **d** 'past tense, perfective'
ed 'past tense, perfective'
s 'present tense'
es 'present tense'
ing 'progressive'

If the verb is the first one encountered in the clause, then a suffix of **d** or **ed** is assumed to indicate past tense. If it is not the first, then this suffix is assumed to indicate a perfective form. This follows from the analysis that only the first verb in a verb phrase carries tense, whereas the other verbs indicate aspect such as perfective, progressive, passive, etc.

Following the network itself, some sample transductions and analyses are shown.


CL/Q
PP/1
'v'2
NP/1
'v',1,2,3,4
PP/2
NP/6
CL/
CL/VP
VP/NP
'by'1
VP/AGT
NP/3
'v'1
'np'
5
CL/DCL
'v'1
'to'2
NP/1
'by'1
VP/
VP/SCOMP
VP/1
VP/STREP


'adj'1
'art'1
PP/1
NP/
NP/ART
'n'2
NP/N
4
'npr'2
'prn'3
NP/SPN


PP/
'prep'1
PP/PREP
NP/1.
PP/NP

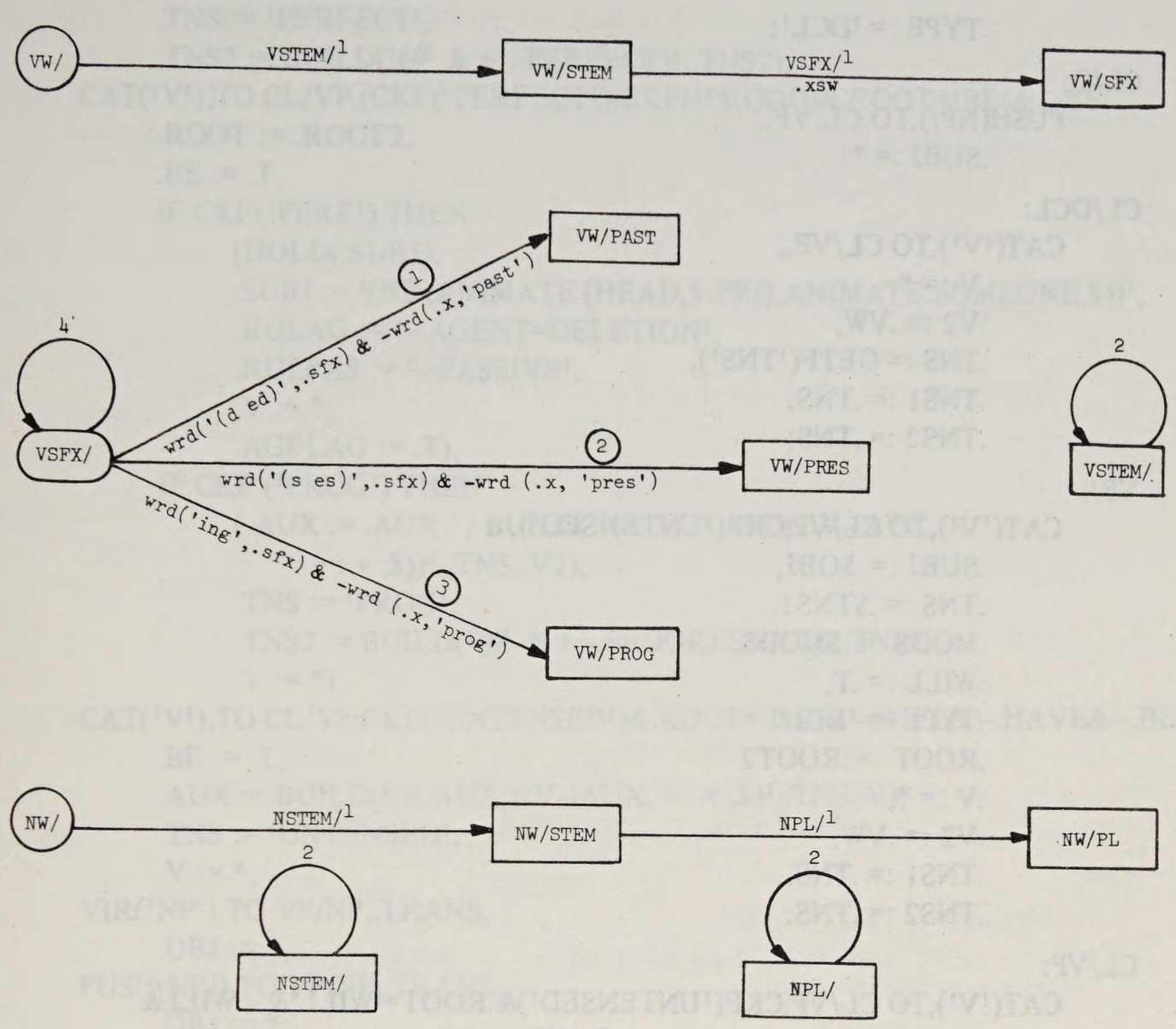


Figure 7

```
CL/:
        PUSH(PP/),TO CL/,CAT('PREP'),
                ADDSMOD;
        CAT('V'),TO CL/Q,CKF('AUX'),
                .V := *,
                .V2 := .VW,
                .TNS := GETF('TNS'),
                .TNS1 := .TNS,
                .TNS2 := .TNS,
                .TYPE := 'Q';
        PUSH(NP/),TO CL/DCL,,
                .SUBJ := *,
```

```
            .TYPE := 'DCL';
CL/Q:
      PUSH(NP/),TO CL/VP,,
            .SUBJ := *;
CL/DCL:
      CAT('V'),TO CL/VP,,
            .V := *,
            .V2 := .VW,
            .TNS := GETF('TNS'),
            .TNS1 := .TNS,
            .TNS2 := .TNS;
 VP/:
      CAT('V'),TO CL/VP,CKF('UNTENSED'),
            .SUBJ := .$OBJ,
            .TNS := .$TNS1,
            .MODS := .$MODS,
            .WILL := .T,
            .TYPE := 'DCL',
            .ROOT := .ROOT2
            .V := *,
            .V2 := .VW,
            .TNS1 := .TNS,
            .TNS2 := .TNS;
 CL/VP:
      CAT('V'),TO CL/VP,CKF('UNTENSED')&.ROOT='WILL'&-.WILL&
                                          -.HAVE&-.BE,
            .ROOT := .ROOT2,
            .WILL := .T,
            .AUX := '((AUX,$:V-AUX,FUT:WILL,$))',
            .V := *,
            .V2 := .VW,
            .TNS1 := 'PRES',
            .TNS2 := 'FUT',
            .TNS := 'UNTENSED';
CAT('V'),TO CL/VP,CKF('PERFECT')&.ROOT='HAVE'&-.HAVE&-.BE,
      .ROOT := .ROOT2,
      .HAVE := .T,
      .AUX := .AUX / / BUILD'((& AUX,PF:V-AUX, + : + ,$))',
            .TNS,.V2),
      .V := *,
      .V2 := .VW,
```

```
        .TNS := 'PERFECT',
        .TNS2 := BUILD('(# & + –PERFECT)',.TNS2);
CAT('V'),TO CL/VP,(CKF('PERFECT')/CKF('PROG'))&.ROOT='BE'&–.BE,
        .ROOT := .ROOT2,
        .BE := .T,
        IF CKF('PERF') THEN
              (HOLD(.SUBJ),
              .SUBJ := '(NP ,ANIMATE (HEAD,$:PRO,ANIMATE:SOMEONE,$))',
              .RULAG := '–AGENT=DELETION',
              .RULPAS := '–PASSIVE',
              .V := *,
              .AGFLAG := .T),
        IF CKF ('PROG') THEN
              (.AUX := .AUX / / BUILD('((& AUX,PROG:V–AUX,
                    + : + ,$))',.TNS,.V2),
              .TNS := 'PROG',
              .TNS2 := BUILD('(# & + –PROGRESSIVE)',.TNS2),
              .V := *);
CAT('V'),TO CL/VP,CKF('UNTENSED')&.ROOT='DO'&–.WILL&–.HAVE&–.BE,
        .BE := .T,
        .AUX := BUILD('(& AUX,$:V–AUX, + : + ,$)',.TNS,.V),
        .TNS := 'UNTENSED',
        .V := *;
VIR('NP'),TO VP/NP,.TRANS,
        .OBJ := *;
PUSH(NP/),TO VP/NP,.TRANS,
        .OBJ := *;
POP   .INTRANS&.TNS,
        BUILD('(# + CL + + ,I– + ((# & S,STA: +))
              ((# & P,P:VP, + + ((# & HD,$: +)))) +)',
              IF(–STACK,'&'),.RULAG,.RULPAS,.TYPE,.SUBJ,
              .TNS2,.AUX,.V,.MODS);

VP/NP:
      WRD(!BY!),TO VP/AGT,.AGFLAG,
            .AGFLAG := .NIL;
      WRD('TO'),TO VP/SCOMP,.SCOMP;
      PUSH(PP/),TO VP/NP,CAT(!PREP!),
            ADDSMOD;
      POP   .TNS,
            BUILD('(# + CL + + ,T– + ((# & S,STA: +))
                  ((# & P,P:VP, + + ((# & HD,$: +))))
                  ((# & O,UND: +)) +)',
```

```
                    IF(–STACK,'&'),.RULAG,.RULPAS,.TYPE,.SUBJ,
                    .TNS2,.AUX,.V,.OBJ,.MODS);
VP/SCOMP:
    PUSH(VP/),TO VP/SPREP,,
        .MODS:= .NIL,
        .OBJ := *;
VP/AGT:
    PUSH(NP/),TO VP/SPREP,,
        .SUBJ := *,
        .RULAG := .NIL;
VP/SPREP:
    WRD('BY'),TO VP/AGT,.AGFLAG,
        .AGFLAG := .NIL;
    PUSH(PP/),TO VP/SPREP,CAT('PREP'),
        ADDSMOD;
    POP  .TNS,
        BUILD('(# + CL + + ,T– + ((# & S,STA: +))
                ((# & P,P:VP, + + ((# & HD,$: +))))
                ((# & O,UND: +)) +)',
                IF(–STACK,'&'),.RULAG,.RULPAS,.TYPE,.SUBJ,
                .TNS2,.AUX,.V,.OBJ,.MODS);
NP/:
    CAT('ART'),TO NP/ART,,
        .ART := BUILD('((& DET,$:ART, + : * ,$)),GETF('REF'));
    CAT('NPR'),TO NP/SPN,,
        .N := *,
        POT := GETF('POT'),
        .$POT := .POT,
        .TYP := 'NPR';
    CAT('PRN'),TO NP/SPN,,
        .N := *,
        .RT := ROOT,
        POT := GETF('POT'),
        .$POT := .POT,
        .TYP := 'PRN';
    ,JUMP NP/ART;
NP/SPN:
    POP  ,BUILD('(NP , + (& HEAD,$: + , + : + ,$))',
                .POT,.TYP,.POT,.N);
```

```
NP/ART:
        CAT('ADJ'),TO NP/ART,,
                .ADJ := .ADJ / / BUILD('((& MOD,$:ADJ,$: * ,$))');
        CAT('N'),TO NP/N,,
                .N := *,
                POT := GETF('POT')
                .$POT := .POT;
NP/N:
        PUSH(PP/),TO NP/N,CAT('PREP'),
                .PPMODS := .PPMODS / / BUILD('(# & NPMOD, + : *))',.PREPTYP);
        POP  ,BUILD('(# NP , + + + ((# & HEAD,$: +)) +)',
                        .POT,.ART,.ADJ,.N,.PPMODS);
PP/:
        CAT('PREP'),TO PP/PREP,,
                .PREP := *,
                .PREPTYP := GETF('TYP'),
                .$PREPTYP := .PREPTYP;
PP/PREP:
        PUSH(NP/),TO PP/NP,,
                .NP := *;
PP/NP:
        POP  ,BUILD('(PP , + (& + , + :PREP,$: + ,$)
                                (# & HEAD,$: +))',
                        .POT,.PREPTYP,.PREPTYP,.NP);
VW/:
        PUSH(VSTEM/),TO VW/STEM,,
        .VW := *,
        .VSW := .FRSTSW,
        .FRSTSW := .NIL,
        .TAGS := FLT(FINDL('TAGS')),
        .ROOT2 := ROOT1
        IF –.ROOT THEN .ROOT=.ROOT2,
        .XSW := (.ROOT=.VW),
        .TRANS := ADDF('TRANS'),
        .INTRANS := ADDF('INTRANS'),
        .SCOMP := ADDF('SCOMP'),
        IF –.FEATLIST THEN .FEATLIST := '($)',
        .X := FLT(FINDL('X')),
        .FEAT := FEATURES;
```

```
VW/STEM:
      PUSH(VSFX/),TO VW/SFX,,
            .TAGS := FLT(FINDL('TAGS')),
            .SFX := *,
            .TNS := GETF('TNS',.TAGS),
            .ATLIST := FT(BUILD('(+ + +)',CKF('PERF',.TAGS),
                                          CKF('PROG',.TAGS),CKF('UNTENSED',.TAGS)));
      POP   ,IF(.VSW,BUILD('(V , + – + : + ,$)',GETF('TNS',.TAGS),
                                    .FEATLIST,.VW),
                        BUILD('(V , + + + – + : + ,$)',CKF('PERF',.TAGS),
                                    CKF('PROG',.TAGS),CKF('UNTENSED',.TAGS),
                                    .FEATLIST,.VW)),
                  BUILD('(V FEATURES +)',.FEAT / / .TAGS);
VW/SFX:
      POP   , IF(.VSW,BUILD('(V , + – +  (& NUC,$:VSTEM, + : + ,$)
                                          (& PER,$:SFX, + : + ,$))',
                                    GETF('TNS',.TAGS),.FEATLIST,.FEATLIST,.VW,
                                    GETF('TNS',.TAGS),.SFX),
                              BUILD('(V, + – + (& NUC,$:VSTEM, + : + ,$)
                                          (& PER,$:SFX, + : + ,$))',
                                    .ATLIST,.FEATLIST,.FEATLIST,.VW,.ATLIST,.SFX),
            BUILD('(V FEATURES +)',.FEAT / / .TAGS);
VSFX/:
      ,JUMP VW/PAST,WRD('(D ED)',.SFX)&–WRD(.$X,'PAST');
      ,JUMP VW/PRES,WRD('(S ES)',.SFX) &–WRD(.$X,'PRES');
      ,JUMP VW/PROG,WRD('ING',.SFX)&–WRD(.$X,'PROG');
      ,JUMP VSFX/,.
            ADDLET(.SFX);
VW/PAST:
      POP   ,.SFX,
            '(VSFX TAGS (PERF TNS PAST))';
VW/PRES:
      POP   ,.SFX,
            '(VSFX TAGS (TNS PRES))';
VW/PROG:
      POP   .,SFX,
            '(VSFX TAGS (PROG))';
VSTEM/:
      POP CAT('V',.VW),
            .VW,
            $;
```

```
      ,JUMP VSTEM/,,
            ADDLET(.VW);
NW/:
      PUSH(NSTEM/),TO NW/STEM,,
            .NW := *,
            .NUM := GETF('NUM',FLT(FINDL('TAGS'))),
            .ROOT := ROOT,
            IF –.NUM THEN .NUM := GETF('NUM'),
            IF –.NUM THEN .NUM := 'SG',
            .POT := GETF('POT'),
            .$POT := .POT;
NW/STEM:
      PUSH(NPL/),TO NW/PL,,
            .SFX := *,
            .NUM := 'PL';
      POP  ,BUILD('(NW , + – + : + ,$)',.POT,.NUM,.NW),
            BUILD('(N FEATURES (NUM + POT +))',.NUM,.POT);
NW/PL:
      POP  ,BUILD('(NW , + – +   (& NUC,$:STEM, + : + ,$)
                                 (& PER,S:SFX, + : + ,$))',
                  .POT,.NUJ,.POT,.NW,.NUM,.SFX),
            BUILD('(N FEATURES (NUM + POT +)',.NUM,.POT);
NSTEM/:
      POP  CAT('N',.NW),
            .NW,
            $;
      ,JUMP NSTEM/,,
            ADDLET(.NW);
NPL/:
      POP WRD('(S ES)',.NPL),
            .NPL,
            .NIL;
      ,JUMP NPL/,,
            ADDLET(.SFX);
```

First, we trace through two examples completely. The two examples are *The pleasant boys will have been considered*. and *Am I considering?* After these, we will trace through a major example, *In the yellow car the boys were believed to have been shot by a spy with a camera.* For this latter example, we trace through the clause level transduction, allowing the reader to trace through the phrase and word level subnetworks.

These examples are designed to show how sentential complements are handled, how passive constructions are changed to active ones, and how word level transduction is accomplished.

Example 1: (THE PLEASANT BOYS WILL HAVE BEEN CONSIDERED)

```
(1)  CL/:3
     .SUBJ     (NP , ANIM   (DET,$:ART,DEF:THE,$)
                            (MOD,$:ADJ,$:PLEASANT,$)
                            (HEAD,$:NW,ANIM–PL
                                  (NUC,$:STEM,ANIM:BOY,$)
                                  (PER,$:SFX,PL:S,$)))
     .TYPE     DCL
     STRING    (BOYS WILL HAVE BEEN CONSIDERED)
(2)  NP/:1
     .ART      ((DET,$:ART,DEF:THE,$))
     STRING    (THE PLEASANT BOYS WILL HAVE BEEN CONSIDERED)
(3)  NP/ART:1
     .ADJ      ((MOD,$:ADJ,$:PLEASANT,$))
     STRING    (PLEASANT BOYS WILL HAVE BEEN CONSIDERED)
(4)  NP/ART:2
     $         (N FEATURES (NUM PL POT ANIM))
     .N        (NW , ANIM – PL   (NUC,$:STEM,ANIM:BOY,$)
                                 (PER,$:SFX,PL:S,$))
     .POT      ANIM
     .$POT     ANIM
     STRING    (BOYS WILL HAVE BEEN CONSIDERED)
(5)  NW/:1
     $         (N FEATURES (POT ANIM))
     .NW       BOY
     .NUM      null
     .ROOT     BOY
     .NUM      SG
     .POT      ANIM
     .$POT     ANIM
     STRING    (Y S)
(6)  NSTEM/:2
     .NW
     STRING    (B O Y S)
(7)  NSTEM/:2
     .NW       BO
     STRING    (O Y S)
(8)  NSTEM/:2
     .NW       BOY
```

```
        STRING    (Y S)
    (9) NSTEM/:1, final state
        form1     BOY
        form2     (N FEATURES (POT ANIM))
(10) NW/STEM:1
    .SFX          S
    .NUM          PL
    STRING        (S)
    (11) NPL/:2
        .SFX      S
        STRING    (S)
        (12) NPL/:1, final state
            form1     S
            form2     null
        (13) NW/PL:1, final state
            form1     (NW , ANIM – PL   (NUC,$:STEM,ANIM:BOY,$)
                                        (PER,$:SFX,PL:S,$))
            form2     (N FEATURES (NUM PL POT ANIM))
    (14) NP/N:2, final state
        form      (NP , ANIM    (DET,$:ART,DEF:THE,$)
                                (MOD,$:ADJ,$:PLEASANT,$)
                                (HEAD,$:NW,ANIM–PL
                                    (NUC,$:STEM,ANIM:BOY,$)
                                    (PER,$:SFX,PL:S,$)))
(15) CL/DCL:1
    $         (V FEATURES (TRANS AUX TNS PRES UNTENSED))
    .V        (V , PRES – TRANS : WILL ,$)
    .V2       WILL
    .TNS      PRES
    .TNS1     PRES
    .TNS2     PRES
    STRING    (WILL HAVE BEEN CONSIDERED)
(16) VW/:1
        $         (V FEATURES (TRANS) TAGS (AUX TNS PRES UNTENSED))
        .VW       WILL
        .VSW      T
        .FRSTSW   null
        .TAGS     (AUX TNS PRES UNTENSED)
        .ROOT2    WILL
        .ROOT     WILL
        .XSW      T
        .TRANS    TRANS
        .INTRANS  null
```

```
.SCOMP      null
.FEATLIST   (TRANS)
.X          null
.FEAT       (TRANS)
STRING      (L)
(17) VSTEM/:2
     .VW        W
     .STRING    (W I L L )
(18) V STEM/:2
     .VW        WI
     STRING     (I L L)
(19) VSTEM/:2
     .VW        WIL
     STRING     (L L)
(20) VSTEM/:2
     .VW        WILL
     STRING     (L)
(21) VSTEM/:1, final state
     form1      WILL
     form2      (V FEATURES (TRANS) TAGS (AUX TNS PRES
                    UNTENSED))
(22) VW/STEM:2,  FINAL STATE
     form1      (V , PRES – TRANS : WILL ,$)
     form2      (V FEATURES (TRANS AUX TNS PRES UNTENSED))
(23) CL/VP:1
     $          (V FEATURES (SCOMP TRANS TNS PRES UNTENSED))
     .ROOT      HAVE
     .WILL      T
     .AUX       ((AUX,$:V–AUX,FUT:WILL,$))
     .V         (V , UNTENSED – TRANS –SCOMP : HAVE ,$)
     .V2        HAVE
     .TNS1      PRES
     .TNS2      FUT
     .TNS       UNTENSED
     STRING     (HAVE BEEN CONSIDERED)
(24) VW/:1
     $          (V FEATURES (SCOMP TRANS) X(PAST PROG PRES)
                    TAGS (TNS PRES UNTENSED))
     .VW        HAVE
     .VSW       null
     .FRSTSW    null
     .TAGS      (TNS PRES UNTENSED)
```

```
.ROOT2      HAVE
.XSW        T
.TRANS      TRANS
.INTRANS    null
.SCOMP      SCOMP
.FEATLIST   (TRANS –SCOMP)
.X          (PAST PROG PRES)
.FEAT       (SCOMP TRANS)
STRING      (E)
(25) VSTEM/:2
     .VW          H
     STRING       (H A V E)
(26) VSTEM/:2
     .VW          HA
     STRING       (A V E)
(27) VSTEM/:2
     .VW          HAV
     STRING       (V E)
(28) VSTEM/:2
     .VW          HAVE
     STRING       (E)
(29) VSTEM/:1, final state
     form1        HAVE
     form2        (V FEATURES (SCOMP TRANS) X(PAST PROG
                       PRES) TAGS (TNS PRES UNTENSED)))
(30) VW/STEM:2, final state
     form1        (V , UNTENSED – TRANS –SCOMP : HAVE ,$)
     form2        (V FEATURES (SCOMP TRANS TNS PRES UNTENSED))
(31) CL/VP:2
     $            (V FEATURES (TRANS INTRANS PERF))
     .ROOT        BE
     .AUX         ((AUX,$:V–AUX,FUT:WILL,$) (AUX,PF:V–AUX,UNTENSED:
                       HAVE,$))
     .V           (V , PERF – TRANS –INTRANS : BEEN ,$)
     .V2          BEEN
     .TNS         PERFECT
     .TNS2        FUT–PERFECT
     STRING       (BEEN CONSIDERED)
     (32) VW/:1
          $            (V FEATURES (ROOT BE) TAGS (TNS PRES))
          .VW          BEEN
          .VSW         null
```

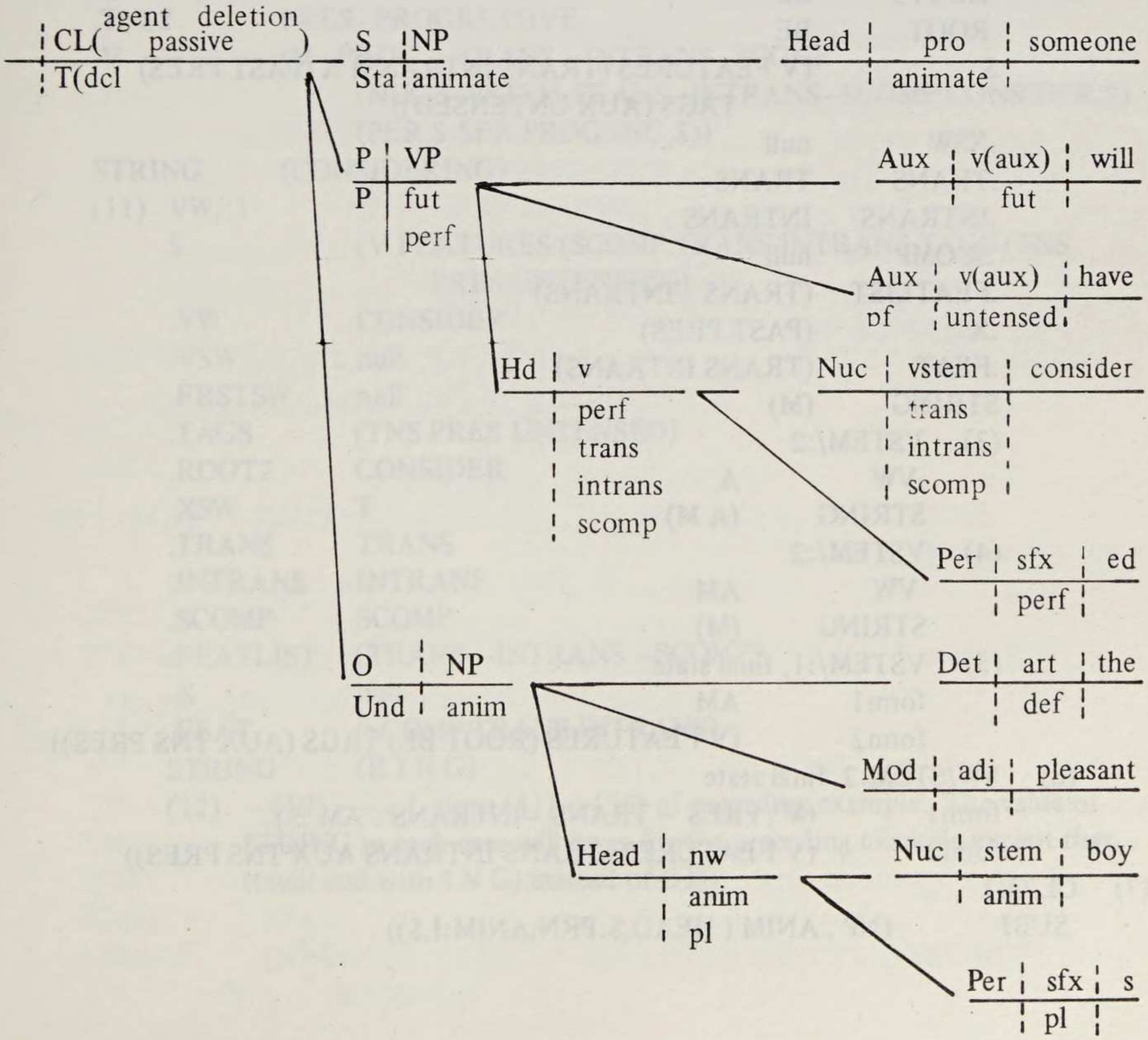
```
.FRSTSW     null
.TAGS       (PERF)
.ROOT2      BE
$           (V FEATURES (TRANS INTRANS) X (PAST PRES)
                  TAGS (AUX UNTENSED))
.XSW        null
.TRANS      TRANS
.INTRANS    INTRANS
.SCOMP      null
.FEATLIST   (TRANS –INTRANS)
.X          (PAST PRES)
.FEAT       (TRANS INTRANS)
STRING      (N)
(33) VSTEM/:2
     .VW          B
     STRING       (B E E N)
(34) VSTEM/:2
     .VW          BE
     STRING       (E E N)
(35) VSTEM/:2
     .VW          BEE
     STRING       (E N)
(36) VSTEM/;2
     .VW          BEEN
     STRING       (N)
(37) VSTEM/:1, final state
     form1        BEEN
     form2        (V FEATURES (ROOT BE) TAGS (TNS PRES))
(38) VW/STEM:2, final state
     form1        (V , PERF – TRANS –INTRANS : BEEN)
     form2        (V FEATURES (TRANS INTRANS PERF))
(39) CL/VP:3
     $            (V FEATURES (TRANS INTRANS PERF))
     .ROOT        CONSIDER
     .BE          T
     holdlist     (NP , ANIM   (DET,$:ART,DEF:THE,$)
                               (MOD,$:ADJ,$:PLEASANT,$)
                               (HEAD,$:NW,ANIM–PL
                                     (NUC,$:STEM,ANIM:BOY,$)
                                     (PER,$:SFX,PL:S,$)))
     .SUBJ        (NP ,ANIMATE (HEAD,$:PRO,ANIMATE:SOMEONE,$))
     .RULAG       –AGENT=DELETION
```

```
.RULPAS     –PASSIVE
.V          (V , PERF – TRANS –INTRANS –SCOMP
                (NUC,$:VSTEM,TRANS–INTRANS–SCOMP:CONSIDER.S)
                (PER,$:SFX,PERF:ED,$))
.AGFLAG     T
STRING      (CONSIDERED)
(40) VW/:1
     $          (V FEATURES (SCOMP TRANS INTRANS TAGS(TNS
                     PRES UNTENSED))
     .VW        CONSIDER
     .VSW       null
     .FRSTSW    null
     .TAGS      (TNS PRES UNTENSED)
     .ROOT2     CONSIDER
     .XSW       T
     .TRANS     TRANS
     .INTRANS   INTRANS
     .SCOMP     SCOMP
     .FEATLIST  (TRANS –INTRANS –SCOMP)
     .X         null
     .FEAT      (SCOMP TRANS INTRANS)
     STRING     (R E D)
(41) VSTEM/:2
     .VW        C
     STRING     (C O N S I D E R E D)
(42) VSTEM/:2
     .VW        CO
     STRING     (O N S I D E R E D)
(43) VSTEM/:2
     .VW        CON
     STRING     (N S I D E R E D)
(44) VSTEM/:2
     .VW        CONS
     STRING     (S I D E R E D)
(45) VSTEM/:2
     .VW        CONSI
     STRING     (I D E R E D)
(46) VSTEM/:2
     .VW        CONSID
     STRING     (D E R E D)
(47) VSTEM/:2
     .VW        CONSIDE
```

```
            STRING      (E R E D)
      (48)  VSTEM/:2
            .VW         CONSIDER
            STRING      (R E D)
      (49)  VSTEM/:1, final state
            form1       CONSIDER
            form2       (V FEATURES (SCOMP TRANS INTRANS)
                                    TAGS (TNS PRES UNTENSED))
(50)  VW/STEM:1
      $                 (VSFX TAGS (PERF TNS PAST))
      .TAGS             (PERF TNS PAST)
      .SFX              ED
      .TNS              PAST
      .ATLIST           PERF
      STRING            (D)
      (51)  VSFX/:4
            .SFX        E
            STRING      (E D)
      (52)  VSFX/:4
            .SFX        ED
            STRING      (D)
      (53)  VSFX/:1
            STRING      null
      (54)  VW/PAST:1, final state
            form1       ED
            form2       (VSFX TAGS (PERF TNS PAST))
(55)  VW/SFX:1, final state
      form1             (V , PERF – TRANS –INTRANS –SCOMP
                          (NUC,$:VSTEM,TRANS–INTRANS–SCOMP:
                            CONSIDER,$)
                          (PER,$:SFX,PERF:ED,$))
      form2             (V FEATURES (SCOMP TRANS INTRANS PERF TNS PAST))
(56)  CL/VP:5
      .OBJ              (NP , ANIM  (DET,$:ART,DEF:THE,$)
                                    (MOD,$:ADJ,$:PLEASANT,$)
                                    (HEAD,$:NW,ANIM–PL
                                      (NUC,$:STEM,ANIM:BOY,$)
                                      (PER,$:SFX, PL:S,$)))
      holdlist          null
      STRING            (CONSIDERED)
(57)  VP/NP:4, final state
      form              (CL–AGENT=DELETION–PASSIVE,T–DCL
```

```
(S,STA:NP,ANIMATE (HEAD,$:PRO,ANIMATE:SOMEONE,$))
(P,P:VP,FUT–PERFECT
      (AUX,$:V–AUX,FUT:WILL,$)
      (AUX,PF:V–AUX,UNTENSED:HAVE,$)
      (HD,$:V,PERF–TRANS–INTRANS–SCOMP
            (NUC,$:VSTEM,TRANS–INTRANS–SCOMP:
                  CONSIDER,$)
            (PER,$:SFX,PERF:ED,$)))
(O,UND:NP,ANIM
      (DET,$:ART,DEF:THE,$)
      (MOD,$:ADJ,$:PLEASANT,$)
      (HEAD,$:NW,ANIM–PL
            (NUC,$:STEM,ANIM:BOY,$)
            (PER,$:SFX,PL:S,$))))
```

This last state is the final one and corresponds to the following tagmeme tree:



Example 2: (AM I CONSIDERING)

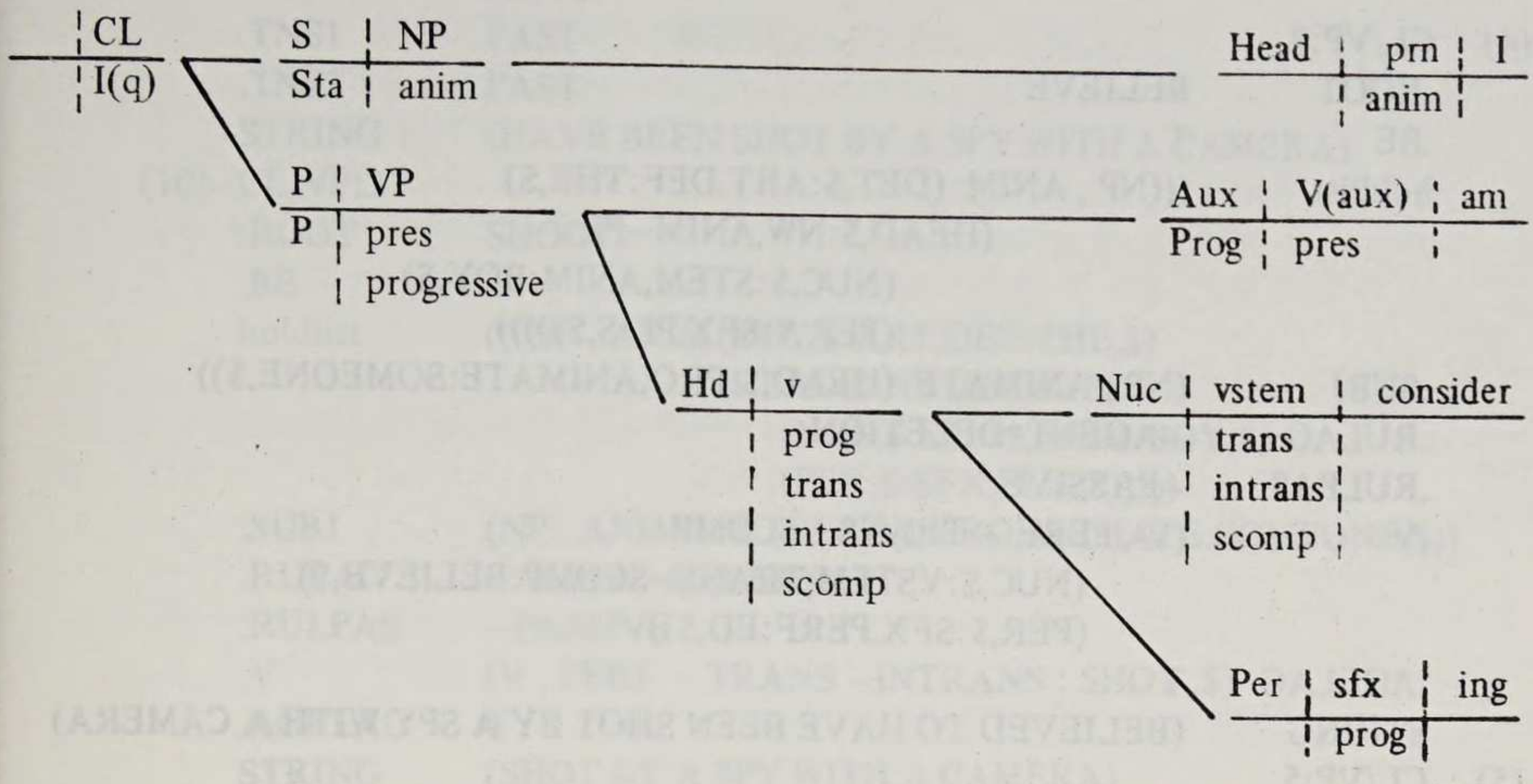
```
(1)  CL/:2
     $            (V FEATURES (TRANS INTRANS AUX TNS PRES))
     .V           (V , PRES – TRANS –INTRANS : AM , $)
     .V2          AM
     .TNS         PRES
     .TNS1        PRES
     .TNS2        PRES
     .TYPE        Q
     STRING       (AM I CONSIDERING)
     (2)  VW/:1
     $            (V FEATURES (ROOT BE) TAGS (AUX TNS PRES))
     .VW          AM
     .VSW         T
     .FRSTSW      null
     .TAGS        (AUX TNS PRES)
     .ROOT2       BE
     .ROOT        BE
     $            (V FEATURES (TRANS INTRANS) X (PAST PRES)
                        TAGS (AUX UNTENSED))
     .XSW         null
     .TRANS       TRANS
     .INTRANS     INTRANS
     .SCOMP       null
     .FEATLIST    (TRANS –INTRANS)
     .X           (PAST PRES)
     .FEAT        (TRANS INTRANS)
     STRING       (M)
          (3)  VSTEM/:2
               .VW          A
               STRING       (A M)
          (4)  VSTEM/:2
               .VW          AM
               STRING       (M)
          (5)  VSTEM/:1, final state
               form1        AM
               form2        (V FEATURES (ROOT BE) TAGS (AUX TNS PRES))
     (6)  VW/STEM:2, final state
          form1        (V , PRES – TRANS –INTRANS : AM ,$)
          form2        (V FEATURES (TRANS INTRANS AUX TNS PRES))
(7)  CL/Q:1
     .SUBJ        (NP , ANIM ( HEAD,$:PRN,ANIM:I,$))
```

```
STRING      (I CONSIDERING)
(8)  NP/:2
     $              (PRN FEATURES (POT ANIM))
     'N             I
     .RT            I
     .POT           ANIM
     .$POT          ANIM
     .TYP           PRN
     STRING         (I CONSIDERING)
(9)  NP/SPN:1, final state
     form           (NP , ANIM (HEAD,$:PRN,ANIM:I,$)
(10) CL/VP:3
     $              (V FEATURES (SCOMP TRANS INTRANS PROG))
     .ROOT          CONSIDER
     .BE            T
     .AUX           ((AUX,PROG:V–AUX,PRES:AM,$))
     .TNS           PROG
     .TNS2          PRES–PROGRESSIVE
     .V             (V , PROG – TRANS –INTRANS–SCOMP
                         (NUC,$:VSTEM,TRANS–INTRANS–SCOMP:CONSIDER,$)
                         (PER,$:SFX,PROG:ING,$))
     STRING         (CONSIDERING)
(11) VW/:1
     $              (V FEATURES (SCOMP TRANS INTRANS TAGS (TNS
                         PRES UNTENSED))
     .VW            CONSIDER
     .VSW           null
     .FRSTSW        null
     .TAGS          (TNS PRES UNTENSED)
     .ROOT2         CONSIDER
     .XSW           T
     .TRANS         TRANS
     .INTRANS       INTRANS
     .SCOMP         SCOMP
     .FEATLIST      (TRANS –INTRANS –SCOMP)
     .X             null
     .FEAT          (SCOMP TRANS INTRANS)
     STRING         (R I N G)
```

(12) – (19) –– c.f. steps (41) – (58) of preceding example. The value of STRING in each case will be as for the preceding example except that it will end with I N G) instead of E D).

```
        (20) VSTEM/:1, final state
             form1      CONSIDER
             form2      (V FEATURES (SCOMP TRANS INTRANS) TAGS
                            (TNS PRES UNTENSED))
    (21) VW/STEM:1
         $              (VSFX TAGS (PROG))
         .TAGS          (PROG)
         .SFX           ING
         .TNS           null
         .ATLIST        PROG
         STRING         (G)
    (22) VSFX/:4
         .SFX           I
         STRING         (I N G)
    (23) VSFX/:4
         .SFX           IN
         STRING         (N G)
    (24) VSFX/:4
         .SFX           ING
         STRING         (G)
    (25) VSFX/:3
         STRING         null
    (26) VW/PROG:1, final state
         form1          ING
         form2          (VSFX TAGS (PROG))
(27) VW/SFX:1, final state
     form1              (V , PROG – TRANS –INTRANS –SCOMP
                        (NUC,$:VSTEM,TRANS–INTRANS–SCOMP:
                            CONSIDER,$)
                        (PER,$:SFX,PROG:ING,$))
           form2        (V FEATURES (TRANS INTRANS SCOMP PROG))
(28) CL/VP:7, final state
     form               (CL,I–Q   (S,STA:NP,ANIM (HEAD,$:PRN,ANIM:I,$)
                                  (P,P:VP,PRES–PROGRESSIVE
                                    (AUX,PROG:V–AUX,PRES:AM,$)
                                    (HD,$:V,PROG–TRANS–INTRANS–SCOMP
                                      (NUC,$:VSTEM,TRANS–INTRANS–SCOMP:
                                          CONSIDER,$)
                                      (PER,$:SFX,PROG:ING,$))))
```

This last form is the final one and corresponds to the following tree:



Example 3: (IN THE YELLOW CAR THE BOYS WERE BELIEVED TO HAVE BEEN SHOT BY A SPY WITH A CAMERA)

(1) CL/:1
.MODS ((CLMOD,LOC:PP,INANIM
(LOC,LOC:PREP,$:IN,$)
(HEAD,$:NP,INANIM
(DET,$:ART,DEF:THE,$)
(MOD,$:ADJ,$:YELLOW,$)
(HEAD,$:NW,INANIM–SG:CAR,$))))
STRING (CAR THE BOYS WERE BELIEVED TO HAVE BEEN SHOT BY A SPY WITH A CAMERA)

(2) CL/:3
.SUBJ (NP , ANIM (DET,$:ART,DEF:THE,$)
(HEAD,$:NW,ANIM–PL
(NUC,$:STEM,ANIM:BOY,$)
(PER,$:SFX,PL:S,$)))
.TYPE DCL
STRING (BOYS WERE BELIEVED TO HAVE BEEN SHOT BY A SPY WITH A CAMERA)

(3) CL/DCL:1
.V (V , PAST – TRANS –INTRANS : WERE ,$)
.V2 WERE
.TNS PAST
.TNS1 PAST
.TNS2 PAST
STRING (WERE BELIEVED TO HAVE BEEN SHOT BY A SPY WITH A CAMERA)

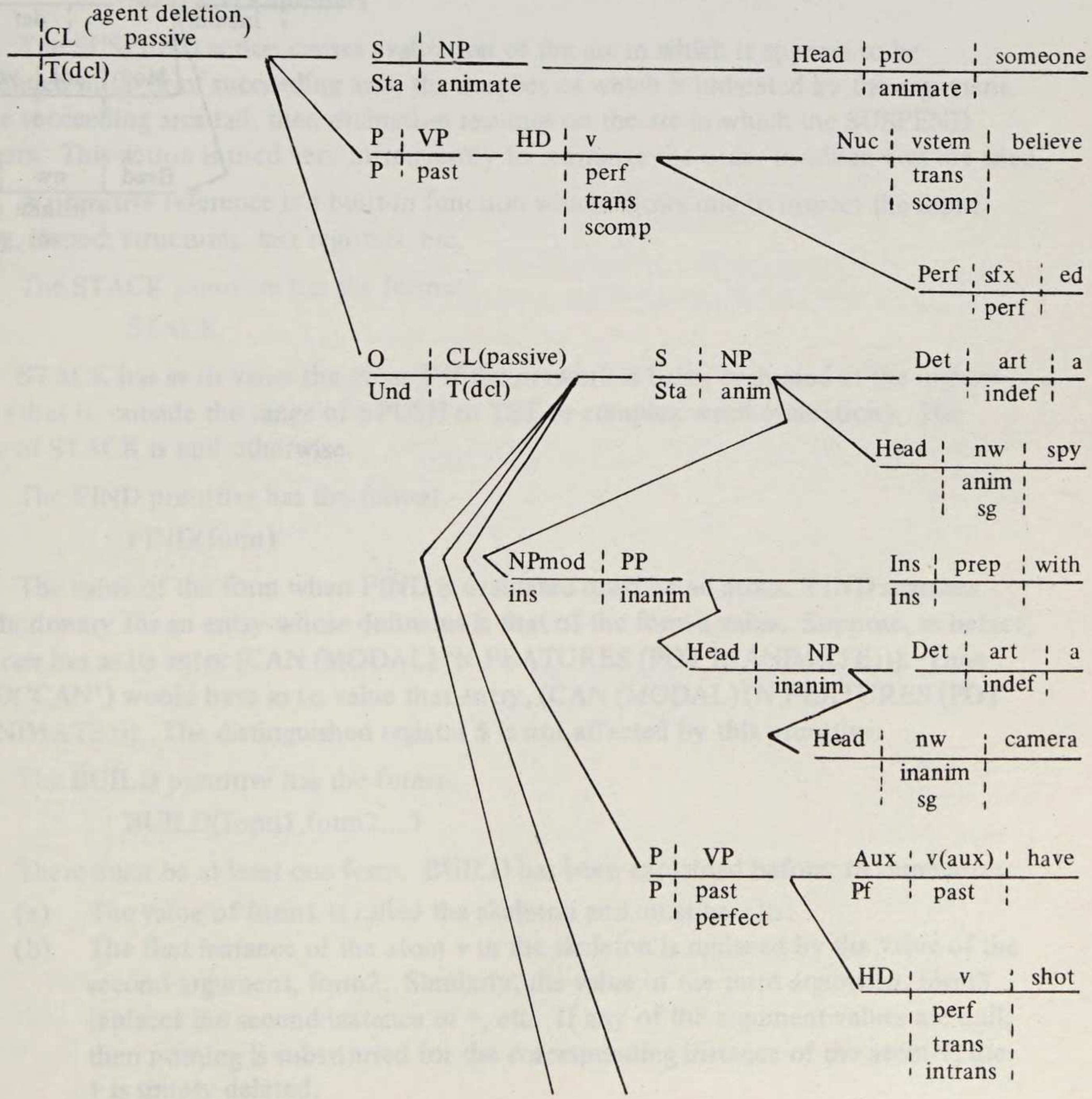
```
(4)  CL/VP:3
     .ROOT        BELIEVE
     .BE          T
     holdlist     (((NP , ANIM  (DET,$:ART,DEF:THE,$)
                                (HEAD,$:NW,ANIM–PL
                                    (NUC,$:STEM,ANIM:BOY,$)
                                    (PER,$:SFX,PL:S,$)))))
     .SUBJ        (NP , ANIMATE  (HEAD,$:PRO,ANIMATE:SOMEONE,$))
     .RULAG       –AGENT=DELETION
     .RULPAS      –PASSIVE
     .V           (V , PERF – TRANS –SCOMP
                      (NUC,$:VSTEM,TRANS–SCOMP:BELIEVE,$)
                      (PER,$:SFX,PERF:ED,$))
     .AGFLAG      T
     STRING       (BELIEVED TO HAVE BEEN SHOT BY A SPY WITH A CAMERA)
(5)  CL/VP:5
     .OBJ         (NP , ANIM    (DET,$:ART,DEF:THE,$)
                                (HEAD,$:NW,ANIM–PL
                                    (NUC,$:STEM,ANIM:BOY,$)
                                    (PER,$:SFX,PL:S,$)))
     holdlist     null
     STRING       (BELIEVED TO HAVE BEEN SHOT BY A SPY WITH A CAMERA)
(6)  VP/NP:2
     STRING       (TO HAVE BEEN SHOT BY A SPY WITH A CAMERA)
(7)  .MODS        null
     .OBJ         (see form for (14) below)
     (8)  VP/:1
          .SUBJ        (NP , ANIM    (DET,$:ART,DEF:THE,$)
                                     (HEAD,$:NW,ANIM–PL
                                         (NUC,$:STEM,ANIM:BOY,$)
                                         (PER,$:SFX,PL:S,$)))
          .TNS         PAST
          .MODS        ((CLMOD,LOC:PP,INANIM
                            (LOC,LOC:PREP,$:IN,$)
                            (HEAD,$:NP,INANIM
                                (DET,$:ART,DEF:THE,$)
                                (MOD,$:ADJ,$:YELLOW,$)
                                (HEAD,$:NW,INANIM–SG:CAR,$))))
          .WILL        T
          .TYPE        DCL
          .ROOT        HAVE
          .V           (V , UNTENSED – TRANS –SCOMP : HAVE ,$)
```

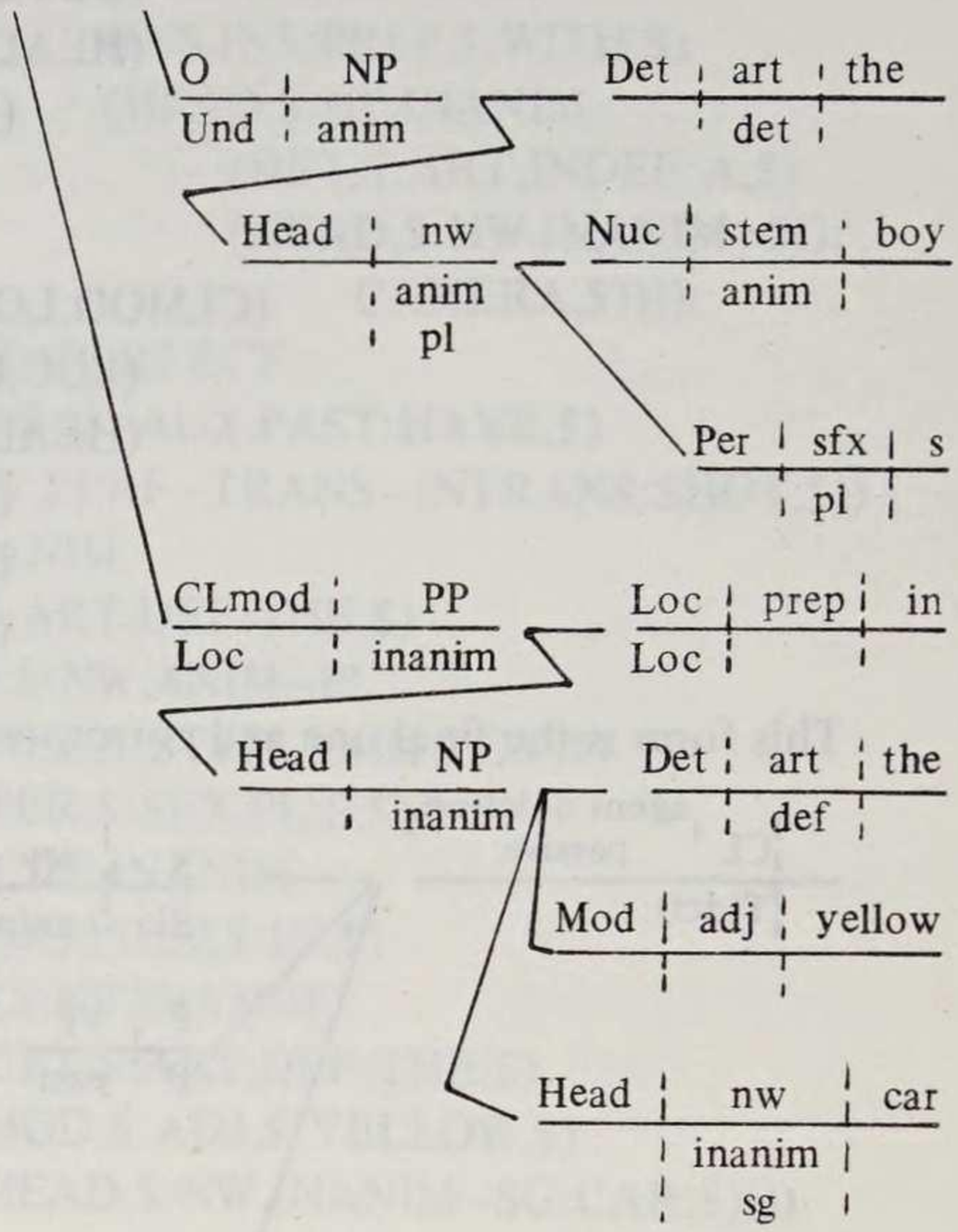
```
      .V2          HAVE
      .TNS1        PAST
      .TNS2        PAST
      STRING       (HAVE BEEN SHOT BY A SPY WITH A CAMERA)
(10)  CL/VP:3
      .ROOT        SHOOT
      .BE          T
      holdlist     (((NP , ANIM (DET,$:ART,DEF:THE,$)
                                (HEAD,$:NW,ANIM –PL
                                    (NUC,$:STEM,ANIM:BOY,$)
                                    (PER,$:SFX,PL:S,$)))))
      .SUBJ        (NP , ANIMATE (HEAD,$:PRO,ANIMATE:SOMEONE,$))
      .RULAG       –AGENT=DELETION
      .RULPAS      –PASSIVE
      .V           (V , PERF – TRANS –INTRANS : SHOT ,$)
      .AGFLAG      T
      STRING       (SHOT BY A SPY WITH A CAMERA)
(11)  CL/VP:5
      .OBJ         (NP , ANIM    (DET,$:ART,DEF:THE,$)
                                 (HEAD,$:NW,ANIM–PL
                                     (NUC,$:STEM,ANIM:BOY,$)
                                     (PER,$:SFX,PL:S,$)))
      holdlist     null
      STRING       (SHOT BY A SPY WITH A CAMERA)
(12)  VP/NP:1
      .AGFLAG      null
      STRING       (BY A SPY WITH A CAMERA)
(13)  VP/AGT:1
      .SUBJ        (NP , ANIM
                         (DET,$:ART,INDEF:A,$)
                         (HEAD,$:NW,ANIM–SG:SPY,$)
                         (NPMOD,INS:PP,INANIM
                             (INS,INS:PREP,$:WITH)
                             (HEAD,$:NP,INANIM
                                 (DET,$:ART,INDEF:A,$)
                                 (HEAD,S:NW,INANIM–SG:CAMERA))))
      .RULAG       null
      STRING       (CAMERA)
(14)  VP/SPREP:2, final state
      form         (CL –PASSIVE ,T– DCL
                         (S,STA:NP,ANIM
                             (DET,$:ART,INDEF:A,$)
                             (HEAD,$:NW,ANIM–SG:SPY,$)
```

```
                                        (NPMOD,INS:PP,INANIM
                                            (INS,INS:PREP,$:WITH,$)
                                            (HEAD,$:NP,INANIM
                                                (DET,$:ART,INDEF:A,$)
                                                (HEAD,$:NW,INANIM-SG:
                                                    CAMERA,$))))
                                    (P,P:VP,PAST-PERFECT
                                        (AUX,PF:V-AUX,PAST:HAVE,$)
                                        (HD,$:V,PERF-TRANS-INTRANS:SHOT,$))
                                    (O,UND:NP,ANIM
                                        (DET,$:ART,DEF:THE,$)
                                        (HEAD,$:NW,ANIM-PL
                                            (NUC,$:STEM,ANIM:BOY,$)
                                            (PER,$:SFX,PL:S,$)))
                                    (CLMOD,LOC:PP,INANIM
                                        (LOC,LOC:PREP,$:IN,$)
                                        (HEAD,$:NP,INANIM
                                            (DET,$:ART,DEF:THE,$)
                                            (MOD,$:ADJ,$:YELLOW,$)
                                            (HEAD,$:NW,INANIM-SG:CAR,$))))
(15) VP/SPREP:2, final state
     form           (CL-AGENT=DELETION-PASSIVE,T-DCL
                        (S,STA:NP,ANIMATE (HEAD,$:PRO,ANIMATE:
                            SOMEONE,$))
                        (P,P:VP,PAST
                            (HD,$:V,PERF-TRANS-SCOMP
                                (NUC,$:VSTEM,TRANS-SCOMP:BELIEVE,$)
                                (PER,$:SFX,PERF:ED,$))
                        (O,UND:CL-PASSIVE,T-DCL
                            (S,STA:NP,ANIM
                                (DET,$:ART,INDEF:A,$)
                                (HEAD,$:NW,ANIM-SG:SPY,$)
                                (NPMOD,INS:PP,INANIM
                                    (INS,INS:PREP,$:WITH,$)
                                    (HEAD,$:NP,INANIM
                                        (DET,$:ART,INDEF:A,$)
                                        (HEAD,$:NW,INANIM-SG:CAMERA,
                                            $))))
                            (P,P:VP,PAST-PERFECT
                                (AUX,PF:V-AUX,PAST:HAVE,$)
                                (HD,$:V,PERF-TRANS-INTRANS:SHOT,$))
                            (O,UND:NP,ANIM
```

```
                                (DET,$:ART,DET:THE,$)
                                (HEAD,$:NW,ANIM–PL
                                    (NUC,$:NW,ANIM–PL
                                        (NUC,$:STEM,ANIM:BOY,$)
                                        (PER,$:SFX,PL:S,$)))
                            (CLMOD,LOC:PP,INANIM
                                (LOC,LOC:PREP,$:IN,$)
                                (HEAD,$:NP,INANIM
                                    (DET,$:ART,DEF:THE,$)
                                    (MOD,$:ADJ,$:YELLOW,$)
                                    (HEAD,$:NW,INANIM–SG:CAR,$)))))
```

This form is the final one and corresponds to the tree shown below:




O NP
Und anim
Det art the
det
Head nw
anim
pl
Nuc stem boy
anim
Per sfx s
pl
CLmod PP
Loc inanim
Loc prep in
Loc
Head NP
inanim
Det art the
def
Mod adj yellow
Head nw car
inanim
sg

APPENDIX

This appendix gives the actions and forms of TNG which are not discussed in the body of the paper.

The **ABORT** action has the format

ABORT

This action causes the arc in which it appears to fail just as though the initial conditions for travelling the arc had not been satisfied. This action is usually embedded in an IF action, and permits additional tests to be conducted on registers, etc., to determine if the arc is really to be travelled or not.

The **SUSPEND** action has the format

SUSPEND(number)

The SUSPEND action causes evaluation of the arc in which it appears to be suspended in favor of succeeding arcs, the number of which is indicated by the argument. If the succeeding arcs fail, then evaluation resumes on the arc in which the SUSPEND appears. This action is used very infrequently to rearrange the order in which arcs are tried.

A primitive reference is a built-in function which allows one to inspect the input string, inspect structures, test registers, etc.

The **STACK** primitive has the format

STACK

STACK has as its value the atom T if the network is being evaluated at the highest level (that is, outside the range of a PUSH or TST or complex word evaluation). The value of STACK is null otherwise.

The **FIND** primitive has the format

FIND(form)

The value of the form when FIND is evaluated must be an atom. FIND searches the dictionary for an entry whose definiens is that of the form's value. Suppose, as before, that *can* has as its entry (CAN (MODAL) (N FEATURES (POT INANIMATE))). Then FIND('CAN') would have as its value that entry, (CAN (MODAL) (N FEATURES (POT INANIMATE))). The distinguished register $ is not affected by this primitive.

The **BUILD** primitive has the format

BUILD(form1,form2,...)

There must be at least one form. BUILD has been explained before; to summarize:

(a) The value of form1 is called the skeleton and must be a list.

(b) The first instance of the atom + in the skeleton is replaced by the value of the second argument, form2. Similarly, the value of the third argument, form3, replaces the second instance of +, etc. If any of the argument values are null, then nothing is substituted for the corresponding instance of the atom +; the + is simply deleted.

(c) Each instance of the atom * is replaced by the current value of the Distinguished Register *.

(d) If a sublist in the skeleton begins with the atom # (Woods uses @), then all subsequent sublists (after substitution for + and *) are combined into a single list. That is, the outer parentheses are removed from each list, which are then combined into a single list. Atoms are presumed to have parentheses around them. Suppose .MODS contained (BIG BAD RED POCKMARKED). Then BUILD('(#MOD:ADJ +)',.MODS) would have as its value (MOD:ADJ BIG BAD RED POCKMARKED). Suppose .MOD1 contained (BIG BAD) and .MOD2 contained (RED POCKMARKED). Then BUILD('(#MOD:ADJ + +)',.MOD1,.MOD2) would have as its value (MOD:ADJ BIG BAD RED POCKMARKED). Or suppose .CONJ contained AND and .NPS contained ((NP (DET:ART THE) (HEAD:NOUN BOY)) (NP (DET:ART A) (HEAD:NOUN GIRL))). Then BUILD('(#(NP +) +)',.CONJ,.NPS) would have as its value (NP AND (NP (DET:ART THE) (HEAD:NOUN BOY)) (NP (DET:ART A) (HEAD:NOUN GIRL))).

(e) If a sublist in a skeleton begins with &, then all subsequent atoms (after (b), (c) and (d) above have been performed) are squeezed into a single atom. This affects all atoms either up to the end of the sublist or until an embedded sublist is encountered. The value of BUILD ('(& A B +)','C') is the list (ABC). The value of BUILD('(# & HEAD: +)','(NP (HEAD:NPR JOHN))') is (HEAD:NP (HEAD:NPR JOHN)).

The **ATOM** primitive has the format

ATOM(form)

Suppose the value of the argument is an atom or is null. Then the value of **ATOM** is the atom T. Suppose instead that the value of the argument is a list. Then the value of **ATOM** is null. Suppose .REG has the value ABC. Then ATOM(.REG) has as its value the atom T. Suppose instead that .REG has the value (A B C). Then ATOM(.REG) has a null value.

The **PUSH** primitive has the format

PUSH(subnetwork reference)

The PUSH primitive operates in exactly the same way as does the PUSH arc, except that it is used as a primitive. The value of the primitive is the atom T if the subnetwork evaluation is successful; the value is null otherwise. The distinguished register * is set exactly as for the PUSH arc. This primitive is used very infrequently, usually in conjunction with TST and SUSPEND. See Woods' grammar of English as listed by Grimes (ms, 47).

The **CAT** primitive has the formats

CAT(form) or CAT(form1,form2)

In the case of one argument, the primitive operates just as does the CAT arc, except that it is used as a primitive. suppose *can* had a dictionary entry as shown previously, (CAN (MODAL) (N FEATURES (POT INANIMATE)). Then if the current word were *can*. CAT('MODAL') would have as its value the atom T. CAT('V') would have a null value. In the case of two arguments, the value of form2 must be an atom, specifying a dictionary entry, which is used in place of the current word. CAT('MODAL', 'CAN') has as its value the atom T. CAT('V','CAN') has a null value. In either case, the distinguished register $ is set to the subentry selected by CAT, if any; if the value of the primitive is null, $ retains its original value. Thus CAT('N','CAN') causes $ to be set to (N FEATURES (POT INANIMATE)). CAT('V','CAN') does not affect $ in any way. As with the CAT arc, a reference to the CAT primitive may cause an automatic push to a complex word subnetwork as outlined in Section 7.

Forms may be combined to make up more complex forms by means of **operators**. The use of the catenation operator in the body of the paper is an example. .MODS / / BUILD('(*)') consists of two forms, .MODS and BUILD('(*)'), joined by the catenation operator / /. This is more formally described below.

Up to now in the discussion of forms we have been concerned only with primary forms. Primary forms consist of registers, distinguished registers, primitives and subroutine references. Now we are in a position to say what a full form is.

A **full form** (usually referred to as just a **form** when there is no ambiguity) consists of either a factor form (see below) or a string of factor forms separated by the **equivalence** operator (=). The value of a full form is that of the factor form in the first case. In the second case it is as follows: If the value of each of the factor forms is identical, then the value of the full form is the atom T. If, however, the value of any of the factor forms differs from that of any other of the factor forms, then the value of the full form is null. Examples of full forms are given at the end of this subsection. As a simple example, consider .NOUN1 = .NOUN2 = .NOUN3. Suppose .NOUN1, .NOUN2 and .NOUN3 each contain the atom MAN. Then the value of the full form would be T. If, however, one of the registers (say .NOUN2) contains the list (MAN), then the value of the full form would be null. The same would be true if this register contained instead the atom MEN.

A factor form (usually referred to as just a **factor**) consists of either a term form (see below) or a string of term forms separated by the **logical 'or'** operator (/). The value of the factor in the first case is that of the term form. In the second case, the value is defined as follows: If the value of any of the term forms is not null, then the value of the factor is the atom T. If, however, all of the term forms are null, then the value of the factor is null. Suppose .R1 is null and .R2 contains (I (JK) L). Then .R1 / .R2 has the value T. If .R2 were also null, then .R1 / .R2 would have a null value. More examples are given later. This operator is not to be confused with the catenation operator.

A term form (usually referred to as just a **term**) consists of either a negation form (see below) or a string of negation forms separated by the **logical 'and'** operator (&). The value of the term in the first case is that of the negation form. In the second case,

the value is defined as follows: If the value of any of the negation forms is null, then the value of the term is null. If the value of each of the negation forms is not null, then the value of the term is the atom T. Suppose, as above, .R1 is null and .R2 contains (I (J K) L). Then .R1 & .R2 has a null value. If .R1 has instead the value MNOP, then .R1 & .R2 has as its value the atom T. More examples are given later.

A **negation form** consists of either a secondary form or a secondary form with a **negation** operator (–) preceding it. In the first case, the value of the negation form is that of the secondary form. In the second case, the value is defined as follows: If the value of the secondary form is null, then the value of the negation form is the atom T. If the value of the secondary form is not null, then the value of the negation form is null. For instance, suppose .REG contained the list (NOT NULL). Then the value of –.REG would be null. If .REG had a null value, then the value of –.REG would be the atom T. This form could also be written as a subroutine NOT but is included here for convenience.

A **secondary form** is either a primary form (as explained above) or is a string of primary forms separated by the **catenation** operator (/ /). In the first case, the value of the form is that of the primary form. In the second case, the value is defined as follows: The value of each primary form in the string must be a list. The value of the secondary form is that of the primary forms, joined into a single list. For instance, suppose .R1 contained (A B C), .R2 contained (((D E (F G) (H) I J K) L) M (N O)) and .R3 contained ((P Q R)). Then the value of .R1 / / .R2 / / .R3 would be (A B C ((D E (F G) (H) I J K) L) M (N O) (P Q R))).

Evaluation of a form is from left to right with the following qualifications:

(1) When a particular primary form is to be evaluated, the most deeply parenthesized embedding is evaluated first. For instance, in FLT(SPLIT(FT(STRING))), the STRING is embedded in FT which is embedded in SPLIT which is embedded in FLT. Thus STRING is evaluated first, then FT, then SPLIT, then FLT.

(2) The definitions of the operators give the following rules of precedence: evaluation of primary forms takes precedence over evaluation of secondary forms, which takes precedence over evaluation of negation forms, which takes precedence over evaluation of factors which takes precedence over evaluation of full forms. To summarize:

/ /	highest precedence operator
–	
&	
/	
=	lowest precedence operator

For example, consider the following form:

.A / .B & –.C / / .D = .E

This is evaluated as follows: Since Operator / / has the highest precedence, .C / / .D is first evaluated. Since Operator – has the next highest precedence, the result of the

preceding evaluation is negated, so –.C/ / .D is evaluated. Operator & has the next highest precedence, so the value of .B is "anded" with the result of the preceding evaluation, resulting in evaluation of .B & –.C/ / .D. Operator / has the next highest order of precedence, so the value of .A is "ored" with the result of the last evaluation, resulting in evaluation of .A / .B & –.C/ / .D. Finally, Operator = has the lowest order of precedence, so the result of the last evaluation is tested for equality against the value of .E, finishing evaluation of the form.

(3) As indicated earlier, a form enclosed in parentheses is a primary form. This is useful to control order of evaluation when the normal precedence is not desired. Consider the following two forms:

.A / .B & –.C

(.A / .B) & –.C

These two forms are evaluated differently. In the first case, we have the following partials evaluated to make up the full form:

–.C

.B & –.C

.A / .B & –.C

In the second case, use of parentheses has changed the order of evaluation. (.A / .B) is a primary form, so the order of evaluation is as follows:

(.A / .B)

–.C

(.A / .B) & –.C

This section has provided all the tools necessary to write forms and actions except for subroutines, which are covered in Section 8.

APPENDIX

NOTES ON LISP

Joseph E. Grimes

LISP is an artificial language that is suitable for the precise expression of functions that apply to complicated data structures. Many of its uses lie within linguistics and artificial intelligence.

The primitive elements of LISP are lists and atoms. An **atom** is represented by a string of characters that contains no spaces and that begins with a letter of the alphabet, or by a number with or without a sign. **Lists** are represented by parenthesization. A list consists of zero or more atoms or lists enclosed in a pair of parentheses. The list representation is useful for data structures of unlimited complexity. LISP programs are themselves lists, and may be operated on like any other lists.

There are three basic LISP **functions** for manipulating lists. (CAR L) has as its value the first element of the list L. (CDR L)-- the two function names go back to the earliest implementations of LISP and refer to machine specific features of those implementations, but are widely accepted--has as its value the rest of the list L except for the first element. (CONS E L) connects the element E to the beginning of the list L to form a new list. The three functions are related in that the value of (CONS (CAR L) (CDR L)) is L.

A LISP expression may have as its value an atom, a list, or one of the truth values T for true or NIL for false. Expressions that yield truth values are called **predicates**. The basic predicates in LISP are (EQ atom atom), which is true if the two atoms named are the same atom, (NULL expression), which is true if the expression named is false, (NUMBERP expression), true if the expression is a number, (GREATERP expression1 expression2), true if the first expression is numerically greater than the second, (ATOM expression), true if the value of the expression is a list.

Predicates are used in the LISP **conditional** expression. It has the form (COND ((A) (X)) ((B) (Y)) ((C) (Z))). A, B, and C are predicate expressions, and X, Y, Z are expressions that can take any value. COND first evaluates A. If A is true, COND takes as its value the value of X. If A is not true, but B is, then the value of COND is Y, and so on through any number of predicate-expression pairs. The last predicate in a COND list must be true in ordinary LISP statements; only in a PROG expression may all predicates in a COND list be false.

Using the elements named so far we can construct a LISP expression which when applied to two LISP elements gives a value of T if they have the same structure containing the same atoms in the same positions, and NIL otherwise. The expression, which is a predicate, is named EQUAL. LAMBDA (adapted from Alonzo Church's lambda notation

for functions) states that the expressions to which EQUAL applies are arbitrarily named X and Y in the function definition:

```
(EQUAL   (LAMBDA (X Y) (COND
      ((ATOM X) (EQ X Y))
      ((ATOM Y) NIL)
      ((EQUAL (CAR X) (CAR Y))
            ((EQUAL (CDR X) (CDR Y)))
      (T NIL) )))
```

This is a typical LISP function in that it is **recursively** defined; EQUAL uses EQUAL as part of its own definition. A recursively defined function must, however, have a stopping place. This is accomplished by having an ATOM predicate as part of the same condition as the recursive part that contains EQUAL. If ATOM is true, this stops the recursion. (Other recursive functions may stop NULL rather than ATOM.)

Once a function is defined, it can be **called** by another function. EQUAL is used by another function named ASSOC which takes two arguments: any LISP element at all as the first argument, and a list of two-element lists as the second argument. If the first argument is EQUAL—in the sense just defined—to the first element of any of the two-element lists, the value of ASSOC is the complete two-element list:

```
(ASSOC (LAMBDA (X Y) (COND
      ((EQUAL (CAR (CAR Y)) X) (CAR Y))
      (T (ASSOC X (CDR Y))) )))
```

A LISP program consists of one or more **expressions** contained within the frame DEFINE((...)). Each of these expressions is itself a list whose first element is an atom that is assigned as the **name** of the expressions, followed by the definition of the expression. The definition is a list whose first argument is the word LAMBDA, the second, a list of **argument** names used in the definition, and third, a list that constitutes the definition itself. In the last example the name of the expression is ASSOC, the arguments used in its definition are X and Y, and the definition is (COND ((EQUAL (CAR (CAR Y)) X) (CAR Y)) (T (ASSOC X (CDR Y))))).

Specific values are assigned in several ways. The simplest is QUOTE, whose value is whatever follows quote. CSET associates a quoted value with a name in the form (CSET (QUOTE TITLE) (QUOTE VICE-PRESIDENT)). Once this assignment has been made, use of the name TITLE will provide the value VICEPRESIDENT assigned to it by CSET. There is a shorthand form (CSETQ TITLE (QUOTE VICEPRESIDENT)) that saves quoting the first argument directly.

Although recursive definitions are typical of LISP, there is a **program** feature that allows LISP expressions to be executed in sequence like the statements in an ordinary computer program. Any recursive expression can be duplicated by a program expression and vice versa. A program expression is a list whose first element is PROG, whose second element is a list of program **variables** to which values are to be assigned within the expression, and whose succeeding elements are a mixture of expressions and atoms. The

atoms are equivalent to statement labels for the expressions that follow them. Integer division can be expressed as a program as follows:

```
(QUOTIENT  (LAMBDA  (N D) (PROG  (U V)
      (SETQ U N)
      (SETQ V D)
   K  (COND ((GREATERP D U)
                (RETURN V)))
      (SETQ U    (PLUS U  (MINUS D)))
      (SETQ V    (PLUS 1 V))
      (GO K)  )))
```

Here SETQ assigns the value of the second argument to the first taken as a name within the program segment. RETURN gives the value that is to be assigned to the whole expression upon completion; it finishes the program and exits from it. PLUS is a **numeric** function whose value is the sum of however many arguments it has, (MINUS n) has as value the negative of **n**. (There is another numeric function TIMES whose value is the product of all its arguments.) GO takes the program back to the label given as its argument. Note that the last predicate of COND does not have to be true in a program expression.

Here are two expressions that accomplish exactly the same thing: counting the number of elements in a list:

```
(LENGTH  (LAMBDA  (M)  (COND
      ((NULL M)  ∅)
      (T  (PLUS 1  (LENGTH  (CDR M))))  )))
(LENGTH  (LAMBDA  (L)  PROG  (U V)
      (SETQ V ∅)
      (SETQ U L)
   K  (COND  ((NULL U)  (RETURN V)))
      (SETQ U  (CDR U))
      (SETQ V  (PLUS 1 V))
      (GO K)  )))
```

Any atom may have a **property list** associated with it. Various kinds of information associated with the atom are put on a property list. If it is the name of a function, for example, the definition of the function is found on the property list under the property EXPR. Constant values set with CSET are entered on the property list of the first argument with the property APVAL. Special flags like TRACE also appear on property lists. CDR of an atom is its property list. (CAR of an atom is meaningless.) For some purposes it is more convenient to keep a function definition on a list known as the **A-list** of atoms paired with their corresponding values than it is to put it on the property list of the atom that names the function. The form (LABEL name function) keeps the function definition off the property list of the name and puts it on the A-list.

The following functions are useful in the management of lists. LENGTH and ASSOC have already been defined.

```
DEFINE((`
  (APPEND (LAMBDA (X Y) (COND
        (( NULL X) Y)
        (T (CONS (CAR X)   (APPEND (CDR X) Y))) )))
  (MEMBER (LAMBDA (A X) (COND
        ((NULL X) NIL)
        ((EQ A (CAR X)) T)
        (T (MEMBER A (CDR X))) )))
 (LAST (LAMBDA (L) (COND
        ((NULL L) NIL)
        ((NULL (CDR L)) (CAR L))
        (T (LAST (CDR L))) )))
 (UNION (LAMBDA (X Y) (COND
        ((NULL X) Y)
        ((MEMBER (CAR X) Y) (UNION (CDR X) Y))
        (T (CONS  (CAR X (UNION (CDR X) Y)))  )))
(DIFFLIST (LAMBDA (A X) (COND
        ((NULL X) NIL)
        ((EQUAL A (CAR X)) (DIFFLIST A (CDR X)))
        (T (CONS  (CAR X) (DIFFLIST A (CDR X))))  )))
(SUBST (LAMBDA (X Y Z) (COND
        ((EQUAL YZ) X)
        ((ATOM Z) Z)
        (T (CONS (SUBST X Y (CAR Z)) (SUBST X Y (CDR Z))))  )))
(MINIMUN (LAMBDA (X) (COND
        ((NULL X) NIL)
        ((NULL (CDR X)) (CAR X))
        (T (SMALLER (CAR X) (MINIMUM (CDR X))))  )))
 (SEQUENCE (LAMBDA (L) (PROG (U V W)
        (SETQ U L)
        (SETQ V (MINIMUM L))
        (SETQ W NIL)
      K (COND ((NULL U) (RETURN W)))
        (SETQ V (MINIMUM U))
        (SETQ U (DIFFLIST V U))
        (SETQ W (APPEND W (LIST V))) (GO K) )))
(MAPLIST (LAMBDA (L FN) (COND
        ((NULL L) NIL)
        (T (CONS (FN L) (MAPLIST (CDR L) FN))) )))
```

where FN is the name of a function an L a list

```
(REVERSE (LAMBDA (X) (REVERSE2 X NIL)))
(REVERSE2 (LAMBDA (X Y) (COND
      ((NULL X) Y)
      (T (REVERSE2 (CDR X) (CONS (CAR X) Y))) )))
(REVERSE (LAMBDA (X) (PROG (Y)
   A (COND ((NULL X) (RETURN Y)))
      ((SETQ Y (CONS (CAR X) Y))
      ((SETQ X (CDR X)) (GO A) )))  ))
```

The two major sources for LISP are McCarthy et al. and Berkeley, 1964.

BIBLIOGRAPHY

Berkeley, Edmund C. and Daniel G. Bobrow, eds. 1964. *The programming language LISP: its operation and applications.* Cambridge: Information International, Inc.

Chomsky, Noam. *Syntactic structures* (Janua linguarum, no. 4) 's-Gravenhage: Mouton and Co.

Elson, Benjamin, and Velma Pickett. 1968. *An introduction to morphology and syntax.* Santa Ana: Summer Institute of Linguistics.

Gratrix, Carol. ms. *Godié narrative.* To appear in Grimes, Papers on Discourse.

Grimes, Joseph E., ed. Forthcoming. *Papers on discourse.*

------. *Transition network grammars: a guide.*

Hale, Austin, ed. 1973. *Clause, sentence, and discourse patterns in selected languages of Nepal, part I.* Norman, Okla.: Summer Institute of Linguistics.

Hale, Austin. 1973. *Toward the systematization of display grammar.* In Hale 1973.

Hewer, Philip. ms. *A lexical approach to clause series in Kasem.* To appear in Papers on Discourse.

Hunt, Geoffrey. ms. *Paragraphing, identification, and discourse types in Hanga.* To appear in Papers on Discourse.

McCarthy, John, et al. 1962. *LISP 1.5 programmer's manual.* Cambridge: The MIT Press.

McKinney, Norris P. ms. *Participant identification in Kaje narrative.* To appear in Grimes, Papers on Discourse.

Perrin, Mona. ms. *Who's who in Mambila folk stories.* To appear in Papers on Discourse.

Schaefer, Robert. ms. *Tone in Gurenne.*

Schöttlendreyer, Burkhard. 1972. *Clause patterns in Sherpa.* Nepal: Summer Institute of Linguistics. mimeo.

Walker, Donald E. 1973. *Automated language processing* (Stanford Research Institute Artificial Inteligence Center Technical Note 77), to appear in Carlos A. Cuadra, ed. Annual Review of Information Science and Technology, Washington, D.C.: American Society for Information Science.

Winograd, Terry. 1971. *Procedures as a representation for data in a computer program for understanding natural language.* Artificial Intelligence Laboratory Report AI TR-17. Cambridge: MIT. Reissued 1972 as Understanding natural language. New York: Academic Press.

Woods, W. A. 1969. *Augmented transition networks for natural language analysis.* Report CS-1, Computer Laboratory, Harvard University, Cambridge, Massachussets.

Woods, William A. 1970. *Transition network grammars for natural language analysis.* Communications of the ACM 13:10.591-606.

------. 1972. *An experimental parsing system for transition network grammars* (BBN Report No. 2362). Cambridge: Bolt, Beranek and Newman Inc.

------, and Ronald Kaplan. 1971. *The lunar sciences natural language information system* (BBN Report No. 2265). Cambridge: Bolt, Beranek and Newman, Inc.

------, R. M. Kaplan, and B. Nash-Webber. 1972. *The lunar sciences natural language information system: final report* (BBN Report No. 2378). Cambridge: Bolt, Beranek and Newman, Inc.

27 Central Bontoc: Sentence, Paragraph, and Discourse by L. Reid (1970) $3.25

28 Identification of Participants in Discourse: A Study of Aspects of Form and Meaning in Nomatsiguenga by M. Wise (1971) 4.50

29 Tupi Studies I by D. Bendor-Samuel et al (1971) 3.00

30 L'enounce Toura (Cota d'Iviore) by T. Bearth (1971) 7.00

31 Instrumental Articulatory Phonetics: An Introduction to Techniques and Results by K. Keller (1971) 3.50

32 According to Our Ancestors by M. Shaw (1971) 6.25

33 Two Studies on the Lacandones of Mexico by P. Baer and W. Merrifield (1971) 4.00

34 Toward a Generative Grammar of Blackfoot by D. Frantz (1971) 5.00

35 Languages of the Guianas by J. Grimes (1972) 3.00

36 Tagmeme Sequences in the English Noun Phrase by P. Fries (1970) 5.00

37 Hierarchical Structures in Guajajara by D. Bendor-Samuel (1972) 4.50

38 Dialect Intelligibility Testing by E. Casad (1973) 5.00

39 Preliminary Grammar of Auca by M. Peeke (1973) 4.50

40 Clause, Sentence, and Discourse Patterns in Selected Languages of Nepal by A. Hale (1973)

Part I	$9.75	Part III	$8.75	
Part II	5.00	Part IV	6.50	Set of Four 28.00

41 Patterns in Clause, Sentence, and Discourse in Selected Languages of India and Nepal by R. Trail (1973)

Part I	$7.00	Part III	$7.50	
Part II	9.00	Part IV	4.25	Set of Four 26.00

42 A Generative Syntax of Penoles Mixtec by J. Daly (1973) 3.25

43 Daga Grammar by E. Murane (1974) 14.00

Copies of the above publications, and a catalog of other items by members of S.I.L., may be ordered from:

Summer Institute of Linguistics
Huntinton Beach, CA. 92648

SUMMER INSTITUTE OF LINGUISTICS PUBLICATIONS
IN LINGUISTICS AND RELATED FIELDS

1	Comanche Texts by E. Canonge (1958)	$2.25
	Microfiche	.85
2	Pocomchi Texts by M. Mayers (1958) Microfiche only	.85
3	Mixteco Texts by A. Dyk (1959) Microfiche only	1.00
4	A Synopsis of English Syntax by E. Nida (1964)	4.00
5	Mayan Studies I by W. Townsend et al Microfiche only	.85
6	Sayula Popoluca Texts by L. Clark (1961)	2.00
7	Studies in Ecuadorian Indian Languages I by C. Peeke et al (1962)	4.00
8	Totontepec Mixe Phonotagmemics by J. Crawford (1963)	3.00
9	Studies in Peruvian Indian Languages I by M. Larson et al (1963)	3.00
10	Verb Studies in Five New Guinea Languages by A. Pence et al (1964)	3.00
11	Some Aspects of the Lexical Structure of a Mazatec Historical Text by G. Cowan	2.25
12	Chatino Syntax by K. Pride (1965)	2.50
13	Chol Texts on the Supernatural by A. Whittaker and V. Warkentin (1965)	2.25
14	Phonemic Systems of Colombian Languages by V. Waterhouse et al (1967) Microfiche only	.85
15	Bolivian Indian Tribes by H. & M. Key (1967)	2.00
16	Bolivian Indian Grammars I by E. Matteson et al (1967)	5.00
	Bolivian Indian Grammars II by E. Matteson et al	5.00
17	Totonac: From Clause to Discourse by R. Bishop et al (1968)	3.00
18	Tzotzil Grammar by M. Cowan (1969)	2.25
19	Aztec Studies I by D. Robinson et al (1969)	2.00
20	Phonology of Capanahua and its Grammatical Basis by E. Loos (1969)	3.50
21	Philippine Languages: Discourse, Paragraph, and Sentence Structure by R. Longacre (1968)	6.50
22	Aztec Studies II -Sierra Nahuat Word Structure by D. Robinson (1970)	3.50
23	Tagmemic and Matrix Linguistics Applied to Selected African Languages by K. Pike (1970)	3.00
24	A Grammar of Lamani by R. Trail (1970)	5.50
25	A Linguistic Sketch of Jicaltepec Mixtec by C. Bradley (1970)	3.00
26	Major Grammatical Patterns of Western Bukidnon Manobo by R. Elkins (1970)	2.00

Date Due

P 98 .S9 1975
Summer Institute of Lingu
Network grammars / edited by J
c.1
010101 000
0 1163 0062122 8
TRENT UNIVERSITY

P98 .S9 1975
Summer Institute of Linguisitcs.
Network grammars.

RECON

DATE | ISSUED TO

265739