THE THEORY AND PRACTICE OF AUGMENTED TRANSITION NETWORK GRAMMARS

Madeleine Bates
Boston University Mathematics Department
Boston, Massachusetts 02115 / USA
and Bolt Beranek and Newman Inc.
Cambridge, Mass. 02138 / USA

1. INTRODUCTION

For the last eight years augmented transition network (ATN) grammars have been used in natural language understanding systems and question answering systems for both text and speech. They have proved to be flexible, easy to write and debug, able to handle a wide variety of syntactic constructions, and easy to interface to other components of a total system. They provide a useful way to give an account of linguistic structures which can be easily communicated to both humans and computers, and they may be (partially) presented by easily visualized diagrams. One does not need to know how to program a computer in order to write or use an ATN grammar. This fact makes it easy for linguists to learn about ATNs. Even linguists without access to a computer have found ATNs useful in the description of several languages [Grimes, 1975]. Although ATN grammars can be written for languages other than English, English will be used for examples throughout this paper; most of the techniques presented will be useful for other languages as well.

Augmented transition network grammars were developed by William Woods [Woods, 1969, 1970, 1973; Woods et al, 1972], although similar but less well developed models appeared independently in earlier work [Thorne et al, 1968; Bobrow and Fraser, 1969]. The advantages of ATNs may be summarized as 1) perspicuity, 2) generative power, 3) efficiency of representation, 4) the ability to capture linguistic regularities and generalities, and 5) efficiency of operation.

Much has been written about both the ATN formalism and various applications, but unfortunately many of these sources are not widely available. This paper attempts to bring together the primary content

of those sources and to provide the reader with examples of several styles of ATN grammars. It is intended to be a guide for those who wish to learn enough about ATNs to implement their own. I have attempted to describe not only the basic mechanism but also recent advances and applications and unsolved problems. I have also tried to indicate time/space/clarity tradeoffs since no one grammar or style of grammar is suitable for all purposes.

The following section introduces the ATN formalism and discusses several types of parsers for ATN grammars. The first part may be skipped by readers familiar with the presentation of ATNs given in [Woods, 1970] and the second may be skipped by those not interested in the details of parser implementation. There follows a discussion of some of the issues and tradeoffs involved in writing ATN grammars for a variety of purposes and illustrations of how some of the syntactic constructions of English may be handled. The paper concludes with a suggested procedure to follow in order to formulate an ATN grammar.

2. THE ATN FORMALISM

A basic transition network (BTN) grammar looks like a collection of finite state transition diagrams; each is a directed graph with labeled states and labeled arcs, a distinguished start state, and a set of distinguished final states. The label on an arc indicates the type (usually the syntactic category, i.e., part of speech) of input which will allow the transition to be made to the next state. A input sequence is said to be accepted by the network if, beginning in the start state, some path (sequence of arcs) can be followed which terminates on a final state.

The network differs from a finite state automaton in that it permits recursion by allowing the label on some arcs to be a nonterminal rather than a terminal symbol. That is, the label on some arcs may call not for a word of input but for a constituent which is found by recursively re-applying the network beginning with an indicated new start state. When such a recursive arc is encountered, the current computation is pushed onto a stack and a new process is begun to look for the desired constituent. When a final state in this lower level is reached, the stack is popped and the suspended

computation is continued. The input pointer, in the meantime, will have been moved to a later point in the sentence (just after the accepted constituent) by the lower level process. An attempt to pop an empty stack when the input pointer is at the end of the input means that the sentence has been found acceptable.

Figure 1 illustrates such a basic transition network with two levels. In it and in all subsequent diagrams in this paper the following conventions are used. States are written as circles or ovals around the name (label) of the state. Start states are indicated by double circles (the initial state for the entire grammar is usually obvious, most likely the first one on the page) and final states are shown by the presence of an arc labeled POP which does not terminate on any other state. The arc types will be described in detail later; in Figure 1 JUMP indicates that a transition may be made without processing any input, CAT means that the current word in the input string must be of the indicated syntactic category (and is "consumed" as the arc is taken), and PUSH means that a recursive call is to be made beginning in the indicated state.

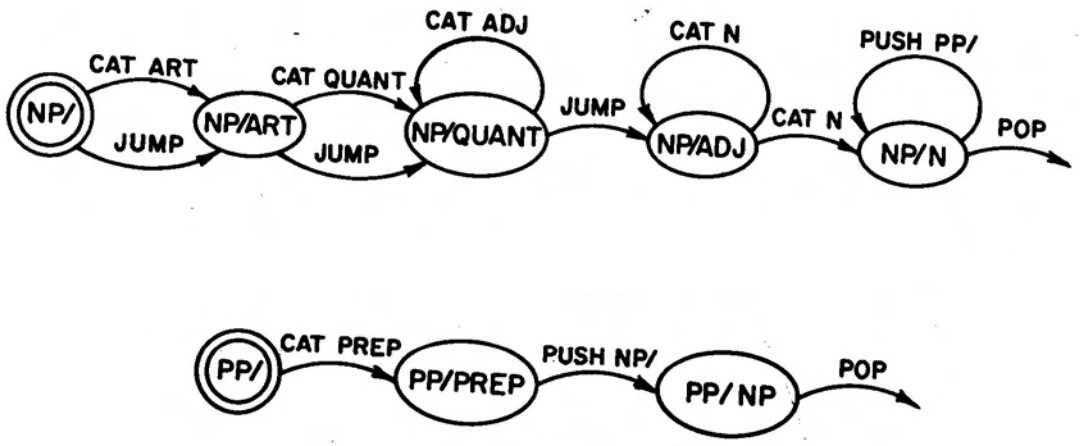
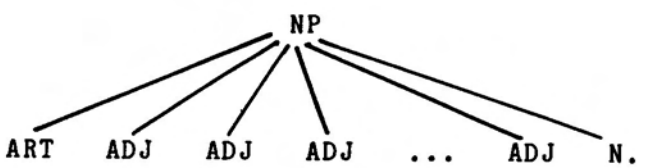


Figure 1: A Small Grammar for Noun Phrases

The convention followed in this paper for labeling states is that state names will generally composed of two parts separated by a slash.

The first part indicates the type of constituent being processed (a noun phrase, for example); the second part indicates either how far through the constituent the parse has proceeded or the sort of construction which may occur next. For example, in a hypothetical grammar: S/IMP may mean that in parsing a sentence it has been discovered to be imperative; NP/ADJ that in a noun phrase either an adjective has been found or we have gotten past the place where an adjective could occur; NP/CONJ? that in a noun phrase we are at a point where a conjunction may be expected; S/S that an entire sentence has been processed (a name like S/POP is also used in this situation to indicate that a POP, the termination of one level of the network, is about to be done). It is crucially important to remember that the state names have meaning only to the writer of the grammar; the parsing system does not use them as anything but unique identifiers for the set of arcs coming from them. Thus the states could just as well be named A, B, C, D, ... instead of S/, NP/HEAD, REL/PRO, ... without changing the operation of the parser. A grammar writer will discover, however, that the the use of mneumonically accurate state names (by the above convention or any other) will greatly clarify the grammar for human use and will simplify the writing and debugging of the grammar.

A BTN grammar as described above is weakly equivalent to a context free grammar or a pushdown store automaton. It differs from strong equivalence only in its inability to characterize unbounded branching, as in



The grammar of Figure 1 corresponds to the following context free grammar (which uses empty productions):

```
NP --> ART? QUANT? ADJS? NMODS? N PPS?
ART? -->      | the | an | a | ...
QUANT? -->      | all | some | ...
AJDS? -->      | ADJ ADJS
ADJ --> pretty | red | ...
NMODS? -->     | N NMODS
N --> dog | girl | love | ...
PPS? -->     | PP PPS?
PP --> PREP NP
PREP --> of | in | with | by | ...
```

Both grammars can produce and can be used to accept sentences such as

The new red law books.
Each beautiful picture in the recent exhibit.
Men with wives in professional careers. (ambiguous)
The tallest boy in a group of students.

One can look at a BTN as a model of a context free grammar in regular expression form. Thus a regular expression rule [Woods, 1969] such as X -> (A) B C* D can be represented by the BTN shown in Figure 2.

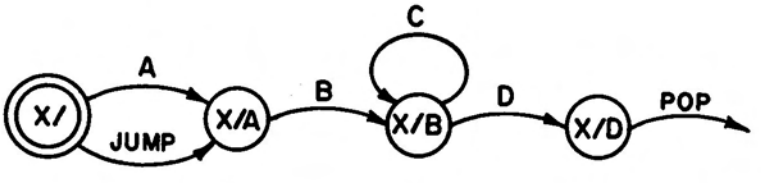


Figure 2: Another Simple Network

But it is well known that context free grammars are not adequate for English. In addition, the basic grammar we have described is capable only of accepting or rejecting input strings; it cannot produce a structure which shows something about the relationships among the words.

To make a more powerful model, each arc is provided with a test and a sequence of actions, thus producing an augmented transition network. The test associated with an arc must be satisfied (in addition to the label) for the arc to be taken, and the actions are executed as the arc is traversed. The actions construct pieces of structure (tree structures, case structures, etc.) and keep them in registers, which may be thought of in programming terms as variables local to the level of the grammar where they are set. Registers and their contents are available on subsequent arcs and can be combined, copied, changed, or added to as more of the input is processed.

The arrangement of states and arcs reflects the surface structure of acceptable input sentences, and the actions permit rearrangements and embeddings to create a structure which may be quite different from the surface structure. This very general mechanism provides a transformational capability which can produce deep structures of the same sort as those of a transformational grammar, and it makes the ATN formalism equivalent in power to a Turing machine.

We now describe in detail the format and operation of the arcs of an ATN grammar. An arc is represented as a parenthesized list of elements which may themselves be words or lists. There are seven types of arcs as shown by the schemas in Table 1. (Capitalized words are actual elements, lower case words in brackets are descriptions of elements which will be defined below, and * is the Kleene star operator which indicates zero or more occurrences of the previous element.)

The first element of each arc indicates its type. The interpretation of the second element depends on the type of the arc and will be explained below. The third element is an arbitrary test which must be satisfied in order for the arc to be taken. Actions, which may occur in any number on all arcs except POP arcs, generally manipulate information that is stored in registers. The register contents are either constants (often used as flags to be tested on later arcs) or pieces of structure. The structures are built using previous register contents and/or the current item of input and/or the features of the current item which are found in the dictionary. The last element of every arc type except JUMP an POP indicates which state of the grammar is to be considered next.

(CAT <category> <test> <action>* (TO nextstate>))

(WRD <word> <test> <action>* (TO nextstate>))

(MEM <list> <test> <action>* (TO nextstate>))

(PUSH <state> <test> <pre-action>* <action>* (TO <nextstate>))

(VIR <constit-type> <test> <action>* (TO <nextstate>))

(JUMP <nextstate> <test> <action>*)

(POP <form> <test>)

Table 1: The Form of Arcs of an ATN Grammar

A CAT arc may be taken if the current input word is of the (syntactic) category specified by the second element of the arc. A WRD arc specifies the exact word of input which is required, rather than a category, and a MEM arc is exactly like a WRD arc except that the input word must be one of the list of words which is the second element of the arc. (Some implementations eliminate MEM arcs but allow the second element of a WRD arc to be a list of words.) All three of these arcs "consume" input when they are taken, that is, they cause the input pointer to be advanced to the next word.

A JUMP arc specifies the state to which a transition is to be made without "consuming" anything from the input string. A VIR arc checks to see whether a constituent of the named type has been placed on the HOLD list by a HOLD action of some previous arc (see below).

A PUSH arc initiates a new, perhaps recursive, call to the network which begins in the indicated state and which looks for a constituent. A POP arc, which has no destination state, marks the state that it leaves as a terminal state for some level of the network; its second element indicates the form (usually some sort of syntactic structure) which is to be returned as the result of the

analysis of the portion of input parsed by the current level of the network. The POP causes control to return to the PUSH arc which caused the process at it's level to be invoked. See Figure 3 for an example of the order in which the arcs of a path involving a PUSH and POP are traversed.

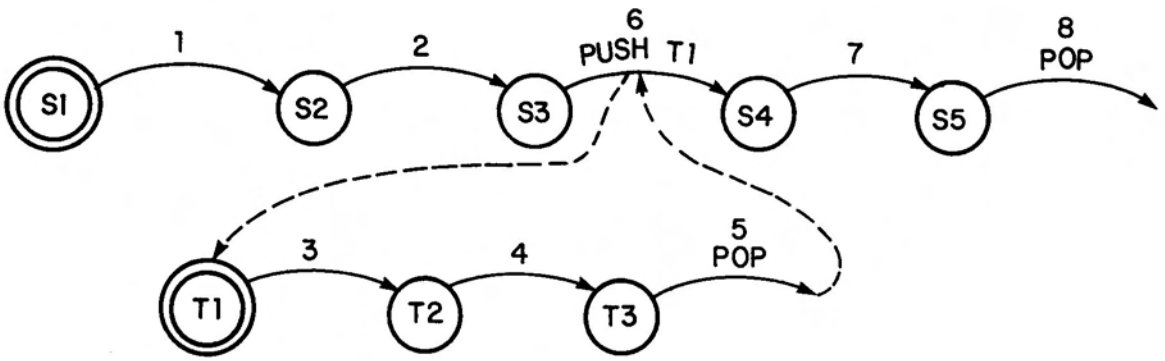
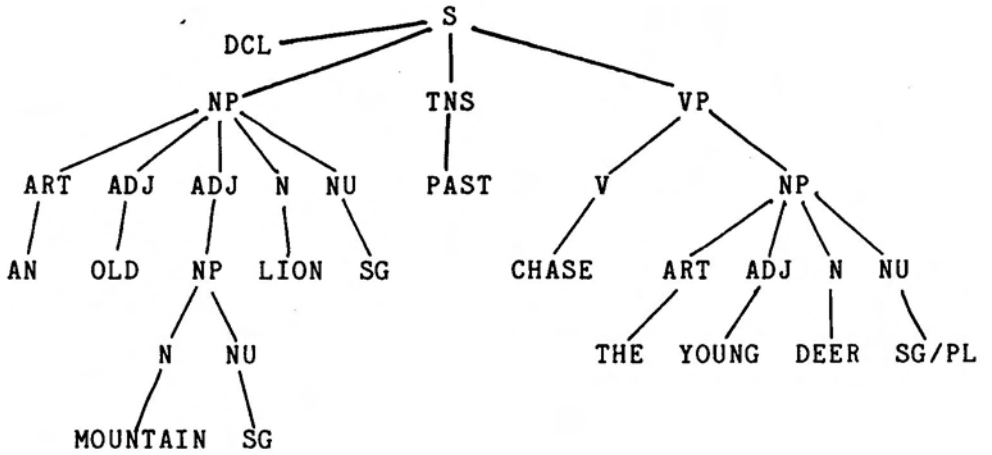


Figure 3: The Operation of PUSH and POP Arcs

When the parser is operating, a number of registers are active. Whenever a PUSH occurs, this register list, along with other information, is saved on a stack while the parser recursively operates on the new (lower) level beginning with an empty register list. When a POP arc is taken, the stack is popped, wiping out the current (lower level) register list and restoring the register list which was current before the last PUSH. The constituent which was POPed (the value of the second element of the POP arc) then becomes the current input item for the rest of the PUSH arc.

In most of the examples given in this paper, the type of structure produced is a parse tree like the deep structure trees of transformational grammar theory. To represent a tree structure in linear form, it is written as a list of elements surrounded by parentheses; the first element is the root of the tree and the subsequent elements are the sons of the root which may be either terminal leaves or subtrees. Figure 4 shows both a standard tree and a linear representation which has been formatted for further clarity.




```
(S DCL
   (NP (ART AN)
       (ADJ OLD)
       (ADJ (NP (N MOUNTAIN)
                (NU SG)))
       (N LION)
       (NU SG))
    (TNS PAST)
    (VP (V CHASE)
        (NP (ART THE)
            (ADJ YOUNG)
            (N DEER)
            (NU SG/PL))))
```

Figure 4: Two Representations of a Parse Tree

Other structures can be produced just as easily, for example, case representations:

```
((ACT: HIDE)
 (ACTOR: JOHN)
 (OBJECT: MONEY)
 (TIME: PAST)
 (LOCATION: FLOWER-POT))
```

tagmemic representations [modified from Grimes, 1975 p.169]:

```
(NP ,ANIM (DET:ART,DEF:THE)
          (MOD:ADJ:PLEASANT)
          (HEAD:NW,ANIM_PL
                    (NUC:STEM,ANIM:BOY)
                    (PER:SFX,PL:S)))
```

semantic representations:

```
(FOR: ALL X1 / (FINDQ: TRIP
                    (DESTINATION CHICAGO)
                    (TRAVELER BILL))
  ; (EXPENSIVE X1) : (OUTPUT: (TIMEOF: X1)))
```

dependency structures, stratificational analyses, and many others as well.

Figure 5 shows the details of the arcs which are needed to make the grammar of Figure 1 an ATN. It is a slightly edited listing of a computer file containing the grammar. Because ATN grammars were developed using the LISP language [Weisman, 1967; Teitelman, 1974] they were written in the LISP form shown below. Each state is represented as a list whose first element is the state name and whose other elements are the arcs emanating from that state. Comments are in upper and lower case and begin with a *. (Readers not familiar with LISP should read Appendix I at this point.) It should be emphasized that ATN parsers and grammars may be written in a variety of programming languages, and that if one other than LISP is chosen it may be advisable to modify the notation.

The actions SETR, SENDR, and LIFTR are the basic actions of an ATN grammar, but implementors may extend this set. Here are descriptions of a number of actions which have been found convenient.

(SETR <reg> <form>) This causes the indicated register to be set to the value of the form.

(SETRQ <reg> <value>) This is like SETR except that the last element is not evaluated. It is equivalent to (SETR <reg> (QUOTE <value>)). Thus (SETR HEAD (NPCLAUSE)) sets a register named HEAD to the value returned by the function NPCLAUSE; (SETRQ HEAD

```
(NP/
   (CAT ART T (* T is a predicate which is always true)
        (SETR ART (BUILDQ ((ART *))))
        (TO NP/ART))
   (JUMP NP/ART T))

(NP/ADJ
   (CAT N T
        (SETR N *)
        (SETR NU (GETF NUMBER))
        (TO NP/N))
   (CAT N T
        (ADDL ADJS (BUILDQ (ADJ (NP (N *) (NU #)))
                           (GETF NUMBER)))
        (TO NP/ADJ)))

(NP/ART
   (CAT QUANT T
        (SETR QUANT (BUILDQ ((QUANT *))))
        (TO NP/QUANT))
   (JUMP NP/QUANT T))

(NP/QUANT
   (CAT ADJ T
        (ADDR ADJS (BUILDQ (@ (ADJ) # (*))
                           (GETF DEGREE)))
        (* This will add the form (ADJ SUPERLATIVE root) for words
              like BIGGEST and the form (ADJ word) if uninflected.)
        (TO NP/QUANT))
   (JUMP NP/ADJ T))

(NP/N
   (PUSH PP/ (PPSTART)
        (* the test checks that the next word is a preposition)
        (ADDL NMODS *)
        (TO NP/N))
   (POP (BUILDQ (@ (NP) + + + ((N +)) ((NU +)) +)
                ART QUANT ADJS N NU NMODS)
        (DETAGREE)
        (* the predicate DETAGREE tests for agreement between the ART
           and N registers to screen out "a books", "an table")))

(PP/
   (CAT PREP T
        (SETR PREP *)
        (TO PP/PREP)))

(PP/PREP
   (PUSH NP/ (NPSTART)
        (* predicate fails if the next word cannot begin a NP)
        (SETR NP *)
        (TO PP/NP)))

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

Figure 5: Details of the Noun Phrase Grammar

(NPCLAUSE)) sets the register to a list which has one element, the word NPCLAUSE.

(ADDL <reg> <form>) This action takes the previous contents (which is expected to be a list) of the named register, adds the value of the form to the left end of the list, and sets the register to this new list. If the register has not been previously set, ADDL sets it to a list which contains the value of the form. It is equivalent to (SETR <reg> (CONS (EVAL <form>) (GETR <reg>))).

(ADDR <reg> <form>) This action is exactly like ADDL except that it adds elements to the right of the previous list. It is equivalent to (SETR <reg> (APPEND (GETR <reg>) (LIST (EVAL <form>)))). ADDL and ADDR are useful for accumulating things like adjectives and conjuncts.

(SENDR <reg> <form>) This is a pre-action which is only used on PUSH arcs. It causes the register to be set to the value of form at the lower level of recursion about to be initialized by the PUSH. This, in effect, allows the lower level network to be like a subroutine to which parameters can be passed via SENDRs.

(SENDRQ <reg> <value>) This is to SENDR as SETRQ is to SETR.

(LIFTR <reg> <form>) This is the inverse of SENDR in that it sets the register to the value of the form at the level just above the current level.

(HOLD <constit-type> <form>) This places the indicated form on the HOLD list as a constituent of constit-type. The HOLD list is a global list which is accessible at all levels. This action together with VIR arcs constitute a mechanism for dealing with the phenomenon called left extraposition in transformational grammar theory. Examples of this will be given below.

(VERIFY <form>) Sometimes it is useful to have a second test on an arc. VERIFY evaluates its form as if it were a predicate. If the form fails, the arc is aborted just as if the condition on the arc had failed. (If this action is implemented, one may do away with the test component on arcs.)

A number of different <form>s may be used within an action. The basic ones are the variables * and LEX and the functions GETR, GETF, and BUILDQ, but others may be included. (In a LISP implementation it is useful to allow any LISP form.) These forms are evaluated as follows:

LEX is always set to the current word of input as it appears before any morphological analysis has been performed on it.

* refers to the current item of input. On a JUMP, POP, WRD, or MEM arc it has the same value as LEX. On a CAT arc it is the root form of the word; thus if LEX = "stopped", * = "stop". On a PUSH arc, * is the current input word for the test and pre-actions, but on subsequent actions on the arc it is the value returned from the lower level computation which was initiated by the PUSH arc.

(GETR <reg>) returns the current contents of the indicated register. If the register has never been set, it returns the value NIL but does not cause an error.

(GETF <feature> <word>) checks the dictionary entry of the word and returns the value of the indicated feature. If the word is omitted, the current word of input is assumed. Thus (GETF NUMBER) may return SG or PL. It may also be used as a predicate for features without values; e.g., (GETF PASSIVE) will return true or false depending on whether or not the current word has the feature PASSIVE. It may be used without its second argument only on a CAT arc, where features are dependent on the syntactic category involved.

(BUILDQ <template> <form>*) is a constructor that takes a template (an arbitrary fragment of structure containing constants and special marks) and a series of forms. It returns a piece of structure that results from substituting the values of the forms for the special marks in the template. The special marks are as follows:

+ expects the corresponding form to be a register name and causes the contents of the register to be substituted for the +.
# causes the corresponding form to be evaluated as a LISP form and substitutes the value for the #.

* causes the value of * to be substituted. It does not need a corresponding form.

@ appears in the form (@ x x ... x) and causes the lists which follow it to be appended together. It does not take a corresponding form.

As an example, if the DET register contained the list (DET THE) and the current word of input on a CAT arc were "books" then the form (BUILDQ (NP + (N *) (NU #)) DET (GETF NUMBER)) would return the structure (NP (DET THE) (N BOOK) (NU PL)). If in addition to the registers just mentioned the ADJS register were set to ((ADJ OLD)(ADJ DUSTY)(ADJ RED)) then the form (BUILDQ (@ (NP +) + ((N *) (NU #))) DET ADJS (GETF NUMBER)) would produce the structure (NP (DET THE) (ADJ OLD) (ADJ DUSTY) (ADJ RED) (N BOOK) (NU PL)).

(ABORT) is a form which causes the arc to fail just as if the test on the arc had failed. It is useful in conditional expressions used as actions.

```
(COND (<pred1> <e11> <e12> ... <e1n>)
      (<pred2> <e21> <e22> ...<e2m>)
       ...
      (<predi> <ei1> <ei2> ... <eij>))
```

This is the LISP way of writing a nested conditional statement. The <e>s may be either actions or forms. See Appendix I for details.

(QUOTE <value>) This is a LISP function which keeps value from being evaluated. See SETRQ above for an example.

It is probably useful to mention a few of the predicates which may be used as tests on the arcs (or in VERIFY or COND forms). The most common predicates are NULLR and the Boolean functions AND, OR, NOT, and EQUAL, but the user will be even more likely to develop his own tests than to invent new actions and forms. Some useful ones follow.

(NULLR <reg>) is true if the register has never been set or if it has been set to NIL.

(CHECKF <feature> <value>) succeeds if the value of feature for the current word is the indicated value; e.g., (CHECKF ROLE OBJ) is equivalent to (EQUAL (GETF ROLE) (QUOTE OBJ)).

(CATCHECK <word> <cat>) tests whether word is of the lexical category cat. It is useful to test words in registers, as in (CATCHECK (GETR V) (QUOTE MODAL)).

(ENDOFSENTENCE) succeeds only if there are no more words in the input string.

(x-AGREE <form> <form>) represents a family of predicates to check some sort of agreement between the forms, e.g., DET-N-AGREE, SUBJ-V-AGREE, etc.

(x-START) represents a set of "look-ahead predicates" which are useful on JUMP and PUSH arcs. For example, IMP-START could be used on a JUMP arc from the initial state of the grammar to test whether there were an untensed verb at the beginning of the sentence; similarly QSTART could be used to see if the input began with a question word.

NIL is a LISP consant which means false. this does not appear by itself as the test on an arc (since the arc would never be taken), but it is the value returned by a predicate function which fails.

T is a LISP constant which is used to represent true; however, since any non-NIL value in LISP is interpreted as true in a Boolean context, any function may be used as a predicate. For example, (GETR SUBJ) can be used not only in a form which uses the contents of the SUBJ register but also as a predicate which tests whether or not the SUBJ register has been set.

To test his understanding, the reader may wish to modify the grammar of Figures 1 and 5 to include pronouns, proper nouns, and/or possessives.

2a. A Detailed Example

In this section we present a more complex grammar and describe the processing of an actual sentence. Figures 6 and 7 show a grammar which, together with the noun phrase and prepositional phrase levels of Figures 1 and 5, can handle a large number of fairly complex English sentences. (This grammar is an elaboration of that given in [Woods, 1970].) Some of the sentences it can parse are:

The girl on the red bus was wanted in several countries by the police.

Will a boy scout help an old woman to cross the street?

The mayor would not have wanted to be elected to the position of dog-catcher.

The money was believed to have been hidden by a thief.

Was the fire engine trying to get to the fire?

A forest fire had been burning in western Colorado for several weeks.

The diagram of the grammar is fairly straightforward, but the details probably look very confusing to those readers new to ATNs. For this reason we will take the time to examine this grammar in detail. It is hoped that the reader will not just skim this section but will take pencil in hand and put as much effort into understanding the following explanation as was put into writing it.

Let us consider the parsing of the sentence "The mayor would not have wanted to be elected to the position of dog-catcher." We will give an overview of the purpose of all the arcs in the grammar, with special attention to the ones used in the parse of this sentence.

Beginning in state S/ we look for a noun phrase to serve, perhaps temporarily, as the subject of the sentence. If one is found, we assume the sentence is declarative. (This is reasonable since our current noun phrase grammar does not accept question determiners like "what" or "which".) If we cannot find a noun phrase at the beginning of the sentence, we assume that we are in a question. We will not give the details of the parse through the NP/ level (it would be a good exercise for the reader) but we assert that the PUSH arc succeeds and that the structure (NP (DET THE)(N MAYOR)(NU SG)) is returned from the lower level and is put in the SUBJ register. The TYPE register is

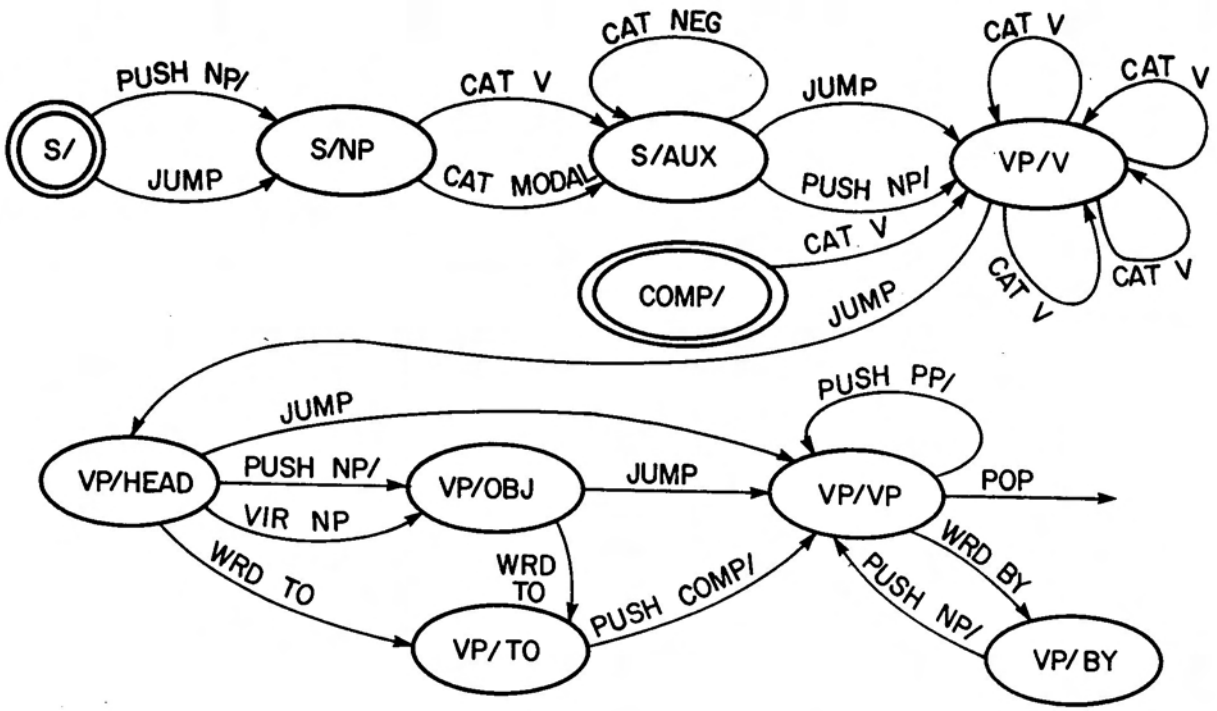


Figure 6: A Grammar for Sentences

set to the constant DCL to indicate that the sentence is declarative, and control moves to S/NP.

In state S/NP the CAT V arc cannot be taken with the word "would" in the input, but the CAT MODAL arc succeeds. The MODAL register is set to the form ((MODAL WILL)), using the root form of "would." (The need for the extra parentheses will be clear later.) The tense is recorded in the TNS register in the form (TNS PAST).

In state S/AUX we pick up the negative. The test on this arc ensures that the arc has not been taken before, thus making double negatives unacceptable. The NEG register is being used both as a flag for the test and as a piece of structure to be later incorporated in the parse structure. The test for "do" in the verb register is necessary to undo the effect of do-support in sentences like "Didn't I meet you in Pango-Pango?" If we were not permitted to have such a conditional action it would be necessary to have two CAT NEG arcs, one with the test for "do" and an unconditional action removing it from the V register and one with a condition that "do" not be in the V register.

```
(S/
   (PUSH NP/ T
       (SETR SUBJ *)
       (SETRQ TYPE DCL)
       (TO S/NP))
   (JUMP S/NP T
       (SETRQ TYPE Q)))

(S/NP
   (CAT V (GETF TNS)
       (SETR V *)
       (SETR TNS (BUILDQ (TNS #)
                         (GETF TENSE)))
       (SETR PNCODE (GETF PNCODE))
       (TO S/AUX))
   (CAT MODAL T
       (SETR MODAL (BUILDQ ((MODAL *))))
       (SETR TNS (BUILDQ (TNS #)
                         (GETF TENSE)))
       (TO S/AUX)))

(S/AUX
   (CAT NEG (NULLR NEG)
       (SETRQ NEG (NEG))
       (COND ((EQUAL (GETR V) (QUOTE DO))
              (SETRQ V NIL)))
       (TO S/AUX))
   (JUMP VP/V (AND (GETR SUBJ)
                   (AGREE (GETR SUBJ) (GETR PNCODE))))
   (PUSH NP/ (NULLR SUBJ)
       (COND ((NOT (AGREE * (GETR PNCODE)))
              (ABORT)))
       (SETR SUBJ *)
       (TO VP/V)))

(VP/V
   (CAT V (AND (GETF PASTPART)
               (EQUAL (GETR V) (QUOTE BE)))
       (HOLD (QUOTE NP) (GETR SUBJ))
       (SETRQ SUBJ (NP (PRO SOMEONE)))
       (SETR AGFLAG T)
       (SETR V *)
       (TO VP/V))
   (CAT V (AND (GETF PASTPART)
               (EQUAL (GETR V) (QUOTE HAVE)))
       (ADDR TNS (QUOTE PERFECT))
       (SETR V *)
       (TO VP/V))
   (CAT V (AND (GETF UNTENSED)
               (GETR MODAL)
               (NULLR V))
       (SETR V *)
       (TO VP/V))
   (CAT V (AND (GETF PRESPART)
               (EQUAL (GETR V) (QUOTE BE)
       (ADDR TNS (QUOTE PROGRESSIVE))
       (SETR V *)
       (TO VP/V))
   (JUMP VP/HEAD T
       (COND ((OR (GETR MODAL) (GETR NEG))
              (SETR AUX (BUILDQ ((@ (AUX) + +))
                                MODAL NEG))))))
```

```
(VP/HEAD
   (JUMP VP/VP (GETF INTRANS (GETR V)))
   (PUSH NP/ (GETF TRANS (GETR V))
       (SETR OBJ *)
       (TO VP/OBJ))
   (VIR NP (GETF TRANS (GETR V))
       (SETR OBJ *)
       (TO VP/OBJ))
   (WRD TO (AND (GETF SCOMP (GETR V))
                (NULLR AGFLAG))
       (SETR SPECIALSUBJ (GETR SUBJ))
       (TO VP/TO)))

(VP/OBJ
   (JUMP VP/VP T)
   (WRD TO (GETF SCOMP (GETR V))
       (SETR SPECIALSUBJ (GETR OBJ))
       (TO VP/TO)))

(VP/VP
   (PUSH PP/ T
       (ADDR VMODS *)
       (TO VP/VP))
   (WRD BY (GETR AGFLAG)
       (SETR AGFLAG NIL)
       (TO VP/BY))
   (POP (COND ((GETR OBJ)
               (BUILDQ (S + + + (@ + (VP (V +) +) +))
                       TYPE SUBJ TNS AUX V OBJ VMODS))
              (T (BUILDQ (S + + + (@ + (VP (V +)) +))
                         TYPE SUBJ TNS AUX V VMODS)))
      T))

(VP/BY
   (PUSH NP/ T
       (SETR SUBJ *)
       (TO VP/VP)))

(VP/TO
   (PUSH VP/ T
       (SENDR SUBJ (GETR SPECIALSUBJ))
       (SENDR TNS (TENSEOF (GETR TNS)))
       (SENDRQ TYPE COMP)
       (SETR OBJ *)
       (TO VP/VP)))

(COMP/
   (CAT V (GETF UNTENSED)
       (SETR V *)
       (TO VP/V)))
```

Figure 7: Details of the Grammar for Sentences

Going back to our analysis, we find ourselves again in state S/AUX with the input pointer positioned at the word "wanted". Since we have something in the SUBJ register which agrees with the verb, we can follow the JUMP arc to state VP/V. If we had had no subject, the PUSH NP/ arc would try to find one, as in "Will the clock strike thirteen?"

Whatever the sentence, by the time state VP/V is reached the first verb (or modal) has been processed. In this state all the rest of the complex verb auxiliary structure is processed. All four CAT V arcs are self loops, implying that they may be taken multiple times in any order unless the conditions on the arcs make this impossible. The first arc may be taken only if the previous verb were some form of "be" and if the current verb may be passivized ("The promise was broken," "Would he have been killed?"); it places what we thought was the subject on a special HOLD list as a NP so it can be picked up later, probably as the object. We invent a dummy subject "someone" to put in the SUBJ register. This may not be correct, so we set a flag, AGFLAG, to indicate that if an agent is found in a by-phrase later in the sentence ("Juliet was loved by Romeo.") it should replace the dummy. This ability to put information in registers and later remove it on the basis of subsequent context is one of the best features of ATN grammars. Although this looks like we are making and later reversing a decision about the role of a portion of input, it can also be thought of as postponing a decision until the differential point is reached.

The second CAT V arc in state VP/V requires that the previous verb be a form of "have" and that the current one be a past participle, as in "may have been killed" or "has surprised." It remembers the effect of the "have" by appending an indicator, PERFECT, to the TNS register and replaces the "have" in the V register by the current verb.

The third CAT V arc may be taken only if the current word is untensed and is preceded by only a modal, as in "Can I go to Washington?" and "He will emulate his superiors." The final CAT V arc handles present participles which are preceded by a form of "be," as in "He will be going to school" and "Was the clown crying?" It adds a PROGRESSIVE indicator to the TNS register and remembers the current verb. (For a sentence like "Has the gambler been losing?" the TNS register would now be (TNS PRESENT PERFECT PROGRESSIVE).)

Note that it would be possible to collapse all four CAT V arcs into one arc which had a T test and a complicated conditional action. That would be slightly more efficient for time and space but would be less clear. This method of processing verb sequences may seem complex, but it is simpler than a set of context free rules to express the same constraints.

For our sample sentence, the third and second arcs would be taken, resulting in the following register settings:

```
SUBJ:  (NP (DET THE) (N MAYOR) (NU SG))
TYPE:  DCL
MODAL: ((MODAL WILL))
NEG:   (NEG)
TNS:   (TNS PAST PERFECT)
V:     WANT
```

Finally only the JUMP arc to state VP/HEAD may be taken. This arc creates an AUX register if either the MODAL or NEG registers are set, so for our sample sentence we get ((AUX (MODAL WILL) NEG)) in the AUX register. (This is a good example of the use of @ in a BUILDQ.)

In state VP/HEAD we know that the head verb of the sentence is in the V register. If the verb is intransitive we jump directly to VP/OBJ. We can look for an object if it is transitive. There are two ways to get an object: by pushing for a noun phrase directly ("I lost my mittens.") or by picking up on the VIR arc the noun phrase which was placed on the HOLD list in VP/V when we discovered that the sentence was passive ("The seed was sown."). None of those three arcs can be taken in our example, but we will postpone discussion of the last arc while describing the rest of the network.

When we get to VP/OBJ, either with or without a direct object, we either jump directly to VP/VP or find the word "to" and go to VP/TO. In state VP/VP we can, if the AGFLAG was set to indicate that the sentence is passive, look for the real agent of the action. If we find it, it is put in the SUBJ register (wiping out the dummy "someone") and the AGFLAG is reset to prevent finding another agent. We can also pick up any number of prepositional phrases modifying the verb. The agent may be interspersed with the prepositional phrases. We now postpone discussion of the POP arc and return to the place we left off above.

Returning to our sample sentence in state VP/HEAD, we find the word "to" in the input string. Since the verb "want" in the V register is marked in the dictionary with the feature SCOMP, meaning it can take a sentential complement and since we are not in a passive sentence, we can take the WRD TO arc to VP/TO. A new register, SPECIALSUBJ, holds the subject; the word "to" is not placed in any register since it does not need to appear in our final tree. A complement of the type we are looking for looks like a declarative sentence beginning with an untensed verb which has no subject in the surface structure. However the deep structure subject of the embedded clause may be the subject of the upper level, e.g. "The sword swallower liked to eat greasy food". After the first verb, we may have anything in the rest of the complement which could occur in the verb phrase of a sentence, hence it is convenient to merge the end of the complement network with the end of the sentence network as shown

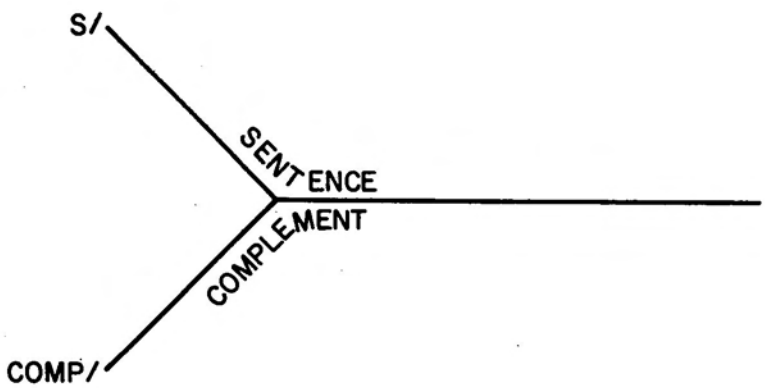


Figure 8: Merging Networks

abstractly in Figure 8. In order for this double use of a portion of the network to work correctly, we must be sure that the registers which are used in the tail of the paths are set on both initial portions. To accomplish this, the PUSH COMP/ arc from VP/TO has three pre-actions which initialize the SUBJ, TNS, and TYPE registers at the lower level with values from the current level. On the path of our example, the SUBJ register is initialized with the contents of SPECIALSUBJ, which is the old subject. If we had reached this state via the WRD TO arc from VP/OBJ, we would be sending down the object to become the lower level subject ("Alice wanted the parrot to speak," "The prodigy was expected to win the prize." Notice the movement of

"the prodigy" from the SUBJ to OBJ registers and then to SUBJ at a lower level!). TENSEOF is a function which will extract the tense, leaving behind the mood and aspect, if any.

Thus we find ourselves in a lower level computation in state COMP/ with three registers set; the registers which existed before the PUSH are hidden on the stack and are completely inaccessible. Taking the CAT V arc from COMP/ (since "be" is untensed) results in a transition to VP/V with the following registers set:

```
SUBJ: (NP (ART THE) (N MAYOR) (NU SG))
TYPE: COMP
TNS:  (TNS PAST)
V:    BE
```

Now the first CAT V arc may be taken. We put the mayor on the HOLD list and replace the SUBJ register by the dummy (NP (PRO SOMEONE)). Then the AGFLAG register is set, and we take the JUMP arc to VP/HEAD without setting the AUX register. Here the VIR arc removes the mayor from the HOLD list and puts him in the OBJ register. NOTE: we did not really need to use the HOLD list here. We could have put the mayor into another register, say EXTRANP, in state VP/V and had a (JUMP VP/VP (GETR EXTRANP)(SETR OBJ (GETR EXTRANP))) arc in place of the VIR arc. The real power of the HOLD list is in holding constituents which must be picked up at an even lower level, e.g. "What painter did he want to have his portrait painted by?"

In VP/VP we push for the prepositional phrase "to the position of dog-catcher" (details left to the reader) and place the resulting structure into the VMODS register. Thus the registers are:

```
SUBJ:   (NP (PRO SOMEONE))
TYPE:   COMP
TNS:    (TNS PAST)
AGFLAG: T
V:      ELECT
OBJ:    (NP (ART THE) (N MAYOR) (NU SG))
VMODS:  ((PP (PREP TO) (NP ...)))
```

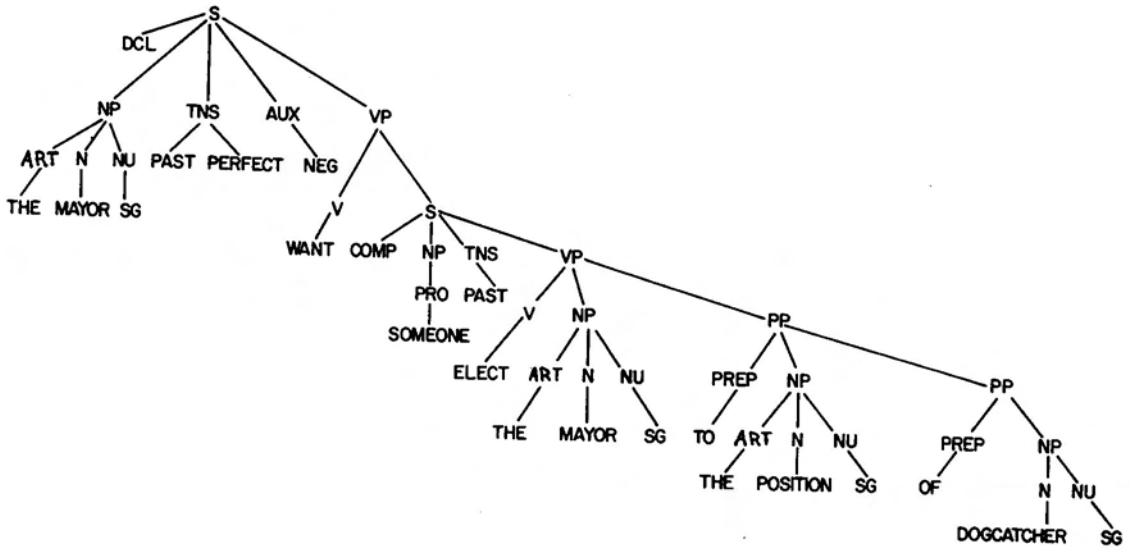
Finally the POP arc in state VP/VP is taken. The BUILDQ creates the structure

```
(S COMP
   (NP (PRO SOMEONE))
   (TNS PAST)
   (VP (V ELECT)
       (NP (ART THE)
           (N MAYOR)
           (NU SG))
         (PP (PREP TO)
             (NP (ART THE)
                 (N POSITION)
                 (NU SG)
                 (PP (PREP OF)
                     (NP (N DOG-CATCHER)
                         (NU SG)))))))).
```

This structure is returned to the configuration we were in at the time of the PUSH from state VP/TO, and the remaining actions on that arc place the structure in the OBJ register. Since there are no more words in the input string, the PUSH PP/ and WRD BY arcs cannot be taken, so the POP arc creates the following structure to return as the value of the successful parse:

```
(S DCL
   (NP (ART THE)
       (N MAYOR)
       (NU SG))
   (TNS PAST PERFECT)
   (AUX (MODAL WILL)
        NEG)
   (VP (V WANT)
       (S COMP
          (NP (PRO SOMEONE))
          (TNS PAST)
          (VP (V ELECT)
              (NP (ART THE)
                  (N MAYOR)
                  (NU SG))
              (PP (PREP TO)
                   (NP ...)))))))
```

which in tree form is:



In this discussion we have been viewing the parsing process as a top down, non-deterministic process where, at any state, any arc whose label and test are satisfied may be taken. Of course, for purposes of illustration, the correct choice was made at every state. The next section will discuss other parsing algorithms which may be used with ATN grammars.

A few comments are in order about the inadequacies of the sample grammar just presented. For passive sentences, there is no check that the NP picked up in a by-phrase can really be the deep structure subject. This is not a check which can be made strictly on syntactic grounds. (Consider, for example, "Caesar was killed by Brutus" and

"They will be married by the time the clock strikes ten.") If a semantic check were made on the head of the NP to see whether it could be the subject of the verb, the incorrect parsing could be ruled out. Similarly, the dummy subject should be "something," not "someone" in cases like "The mountains were worn down (by the wind)." Again, a call to a function outside the parser could determine which form to use.

The complement structure in this grammar is also limited to the simplest cases. Some verbs should cause their subject rather than their object to be sent down:

John told his friend to leave. (obj)
John promised his friend to leave. (subj)

This feature, if noted in the dictionary, may be tested on the WRD TO arc in VP/OBJ to determine which NP to send down. This grammar will also accept various badly formed sentences, for example, "The house has have painted." Additional tests could be added to screen out these ill-formed strings, but only at the cost of complicating the grammar more than is desired for this introduction.

To show how large, and yet understandable, ATN grammars can grow, Figure 9 shows a diagram of the grammar which was used for the LUNAR question answering system [Woods et al, 1972]. This grammar can handle some forms of anaphoric reference, some ellipsis, several types of complements, reduced and unreduced relative clauses, parenthetical expressions, comparatives, passives, and a number of other syntactic constructions.

ATN grammars large enough to be really useful are too large to be included in a paper such as this. Readers who are interested in seeing complete listings of large ATN grammars should consult other references. The final report of the LUNAR question answering system [Woods et al, 1972] contains a listing of the grammar shown in Figure 9 together with descriptions of many of the functions used in tests and actions on the arcs. Leal [Leal, 1975] gives a grammar based on tagmemic theory that is written in a semi-LISP notation which is particularly easy for non-LISP-users to follow. Bates gives a listing of the large syntactic grammar used in the BBN speech understanding system in [Bates, 1975] and diagrams of a portion of the semantic grammar for the same system in [Woods et al., 1976]. Anyone who knows

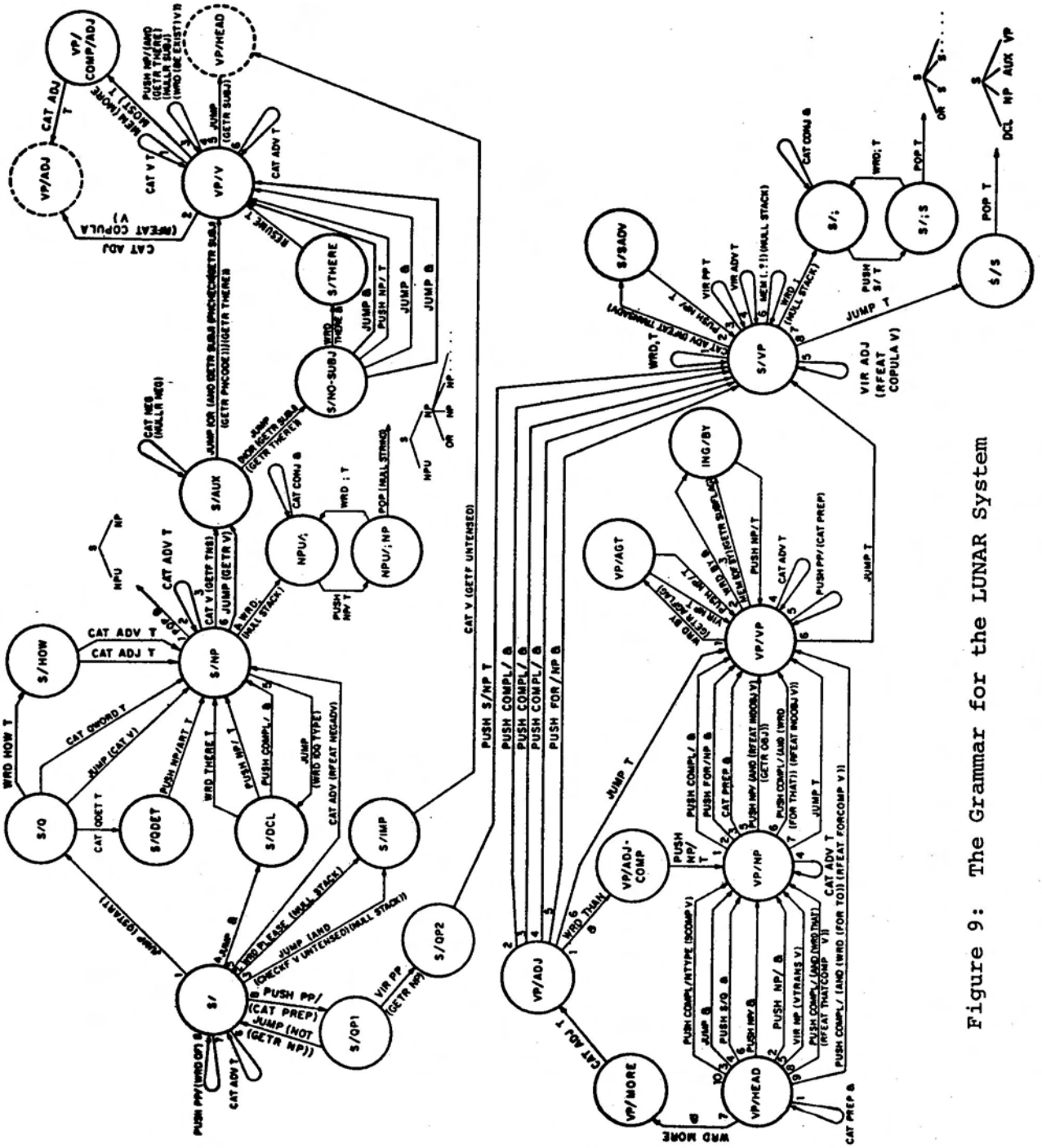


Figure 9: The Grammar for the LUNAR System

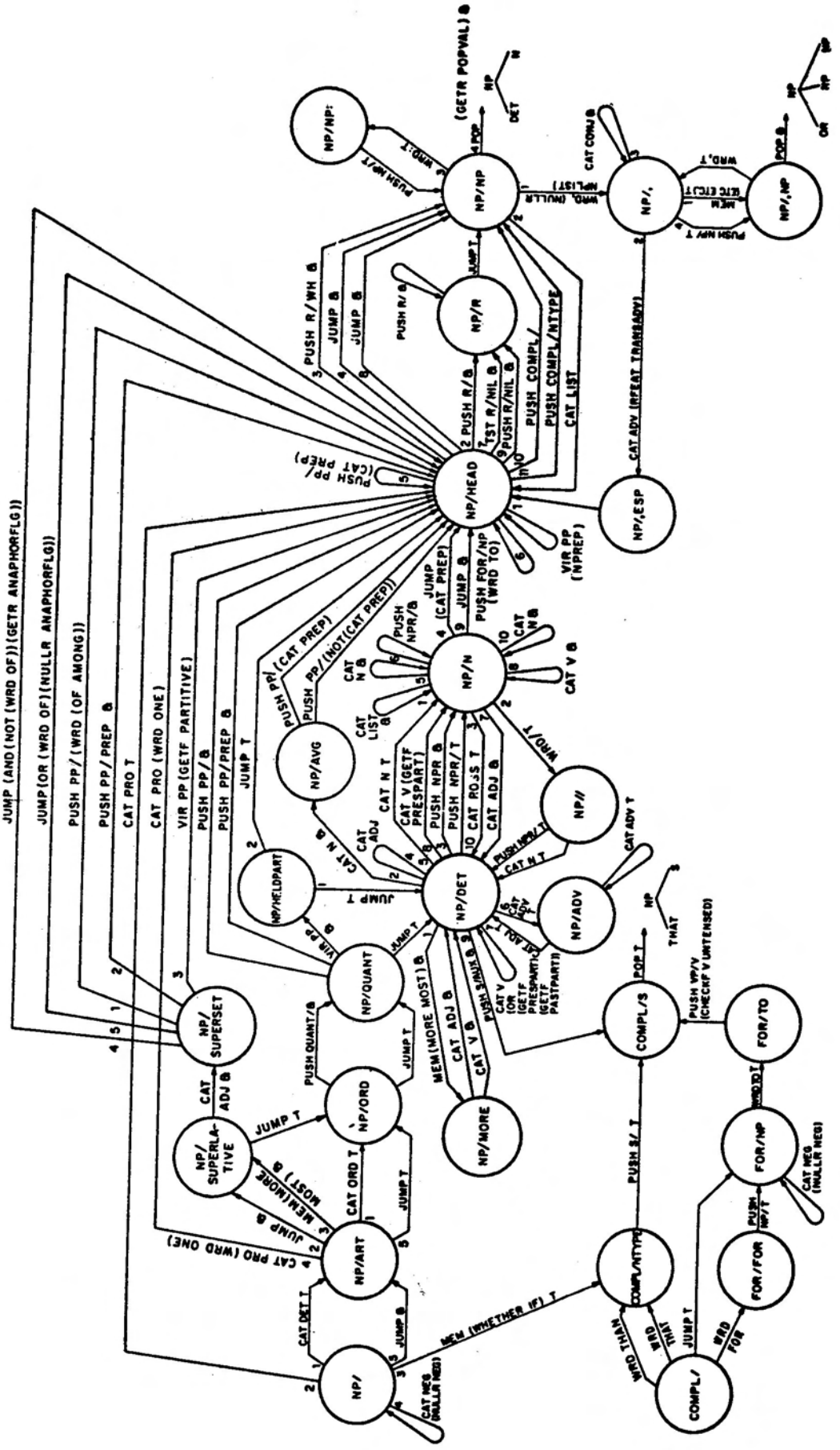


Figure 9 (con't): The Grammar for the LUNAR System

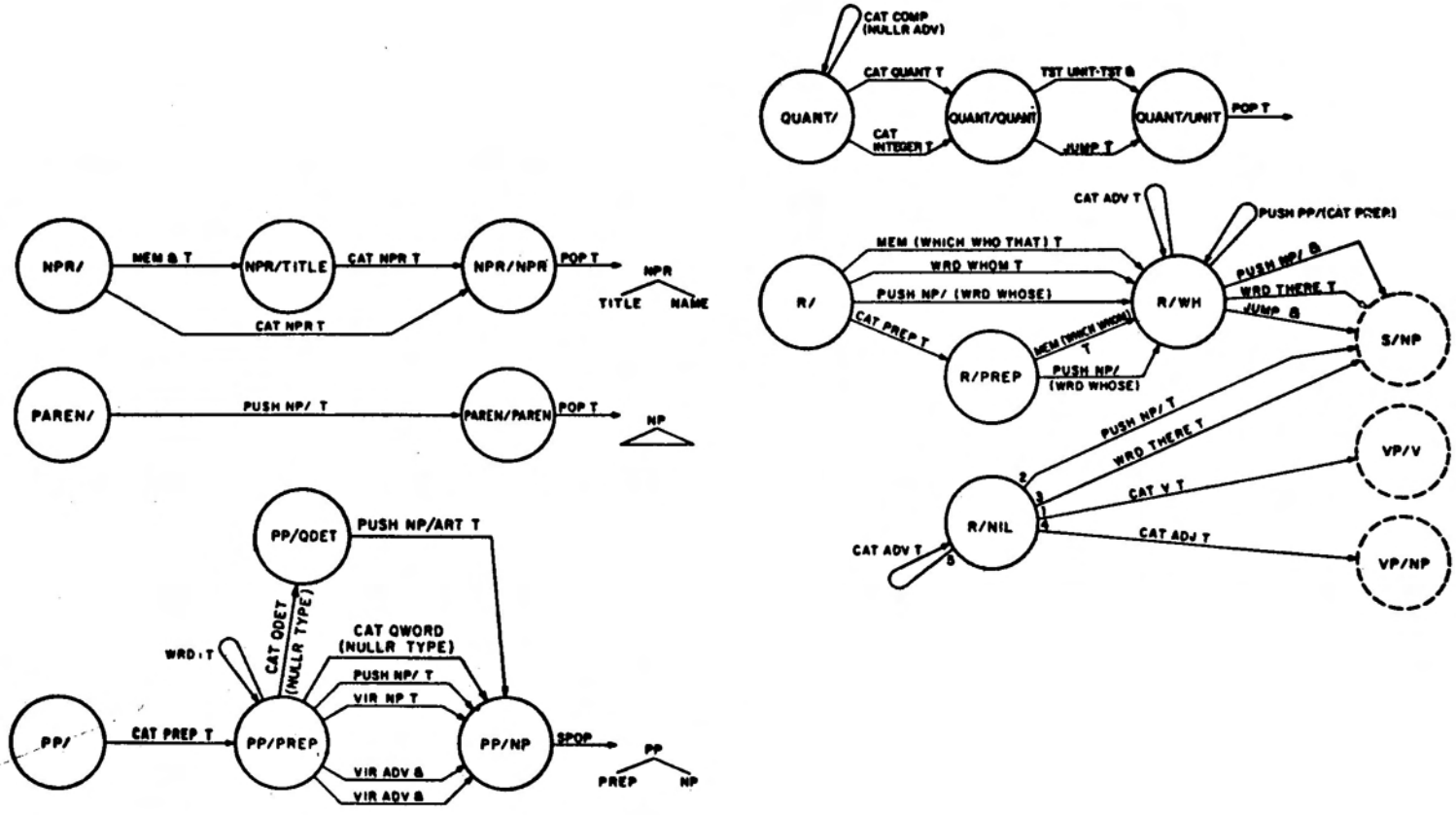


Figure 9 (con't): The Grammar for the LUNAR System

Lamnsok (a Benue-Congo language spoken in part of the United Republic of Cameroun) will be interested in the verb cluster grammar given in [Grebe, 1975]. Grimes [Grimes, 1975] gives an ATN diagram followed by two sets of arcs and actions: one produces tagmemic structures and the other regular parse trees.

2b. Formal Properties of ATNs

It can be shown that the ATN formalism described above is equivalent in power to a Turing Machine. However, so are many unwieldy and inefficient formal systems. It has also been shown [Woods, 1969] that it is possible to eliminate direct left and right recursion and to optimize an ATN network using finite state optimization techniques.

Earley's algorithm for context free recognition [Earley, 1970] follows all alternatives in parallel, using a particularly clever method of merging parse paths. It can parse strings in an amount of time bounded by Kn↑3 where n is the length of the input string and K is a constant of proportionality which is dependent on the grammar (but not on the input); for linear grammars the recognition time is bounded by n↑2, and for LR(k) grammars the bound is n.

There are two interesting things to note about Earley's algorithm. First, it works for context free grammars in any form (not in just a normalized form) and it automatically achieves the smaller bounds for the linear or LR(k) grammars (without having to be told that the grammar is so restricted). Second, an ATN grammar can be recognized by a modification of Earley's algorithm which maintains the time bounds. That is, removing recursion from an ATN causes the upper time bound to be reduced from n↑3 to n↑2 or even to n. Simple finite state optimization may be used to reduce the constant of proportionality in the n↑3 case.

2c. Modifications to the Grammar Formalism

For particular applications, changes may easily be made in the grammar formalism (necessitating corresponding changes in the parser).

The following changes have been incorporated in various ATN grammars.

1. Weights on arcs. A number can be placed on every arc, just after the test, to indicate either how likely the arc is to be taken from that state or how much information is likely to be gained from taking the arc. The parser can use this information to score alternative paths in order to pursue the best ones first. A variation of this would be to have an action on the arc which calculates (using all context available) a score for the parser to incorporate in its paths.

2. Actions on POP arcs. Such actions could be used to lift register contents to the next level or to ready registers for building the final structure.
If actions are not permitted here, they may have to be duplicated on each arc entering the final state.

3. Factorization of tests. For the BBN speech parser [Bates, 1975b, 1976] each arc test was split into two tests; one was context free (testing only features of the current word) and the other was context-sensitive (involving register contents).

4. Grouping of arcs. The Burton grammar-compiler (see below) allows a subset of the arcs from any state to be grouped in a list which means that whenever one arc from that set is taken the others will necessarily fail, so they do not have to be examined even if the parse should back up over the arc which was originally taken. For example, a set of mutually exclusive WRD arcs should be so grouped. This is purely an efficiency measure.

5. New actions. Because of the EVAL feature of LISP, any LISP form can be used as an action on an arc. If this feature is not available, a fixed set of actions must be known to the interpreter, but the designer of the parser/grammar combination still has the freedom to specify actions particular to his domain before implementation. If the parser is to be interfaced with another program, say a semantic interpretation routine, it is convenient to be able to call that program by an action on an arc. In that way the results of the interpretation (or whatever) may be used to guide the further parsing.

3. PARSERS FOR ATN GRAMMARS

Because an ATN grammar is separate from its parser, a number of different parsing algorithms can be used, just as there exist top down, bottom up, table driven, and other types of parsers for context-free grammars. In fact, almost any classical context free parsing algorithms could be adapted to ATNs by providing a mechanism to carry along register contents and to perform the tests and actions on arcs of the grammar. Parsers can be tailored for speed, debugging aids, or special requirements of the application for which they will be used.

Although all the parsers described below have been implemented in LISP, there is no reason why one could not be written in a high level language such as PL/1, ALGOL, PASCAL, or BCPL, or even in an assembler language. The language should be recursive or at least able to simulate recursion, and the ability to evaluate a portion of the grammar as if it were a part of a program is a very desirable, though not absolutely necessary, feature.

3a. ATN Interpreters

The first parsers which were written for ATNs (and the easiest to reproduce) act like interpreters for the grammar-language, using the input string and dictionary as data. The simplest ATN parser to build is a top down, depth first interpreter of the grammar similar to a parser for a context free grammar. In this model the arcs from each state are tried in the order in which they are written. This allows the grammar writer to deliberately order the arcs so that the most likely ones come first and thus gain control over the order in which alternative paths are tried. Such a parser may be written using less than 5000 LISP cells.

The original ATN parser [Woods, 1970. 1973] is more versatile than the parser just described. It is based on a list of alternatives, each of which contains all the information necessary to restart the parser at the point at which the alternative was created. This implies that alternatives can be done in any order, not just depth first. An alternative consists of the current input string, the

current state of the grammar, the arcs remaining to be tried at that state, the register list, and the stack. If the HOLD list is used, it must be remembered as part of an alternative too, and it is often useful to remember the path of arcs which have been taken at the current level. Alternatives may be created at several different times during the parsing process. The most common one remembers the arcs remaining to be tried after one arc from the state is followed. Resuming this alternative "backs up" to that state and looks for another arc to try. Another type of alternative may be created when there is ambiguity about the part of speech of a word. Another is created (by a function called LEXIC) when there is ambiguity about the next word of input. (This happens in cases where there are compound words like "common market" which possibly should be collapsed.

One may think of the parsing process as one of moving from one configuration to another, where a configuration is a snapshot of the current state of the parse (i.e., the current input state, stack and register list). When processing one word of input the parser may follow just one path or may try to follow several in parallel. The process of creating a path is exactly the process of creating one or more configurations from a given configuration. To make this a little more concrete, let us look at the main functions of the parser for the LUNAR system.

The function PARSER is called with the input string as its argument. It sets up an initial configuration (initial state, empty register list and stack). It calls the function LEXIC to perform lexical analysis of the string to determine the next word. This may involve expanding contractions, compressing compound words, making substitutions according to dictionary information, etc. Then PARSER calls STEP to compute a new set of configurations from each currently active configuration.

STEP, which may be given either a configuration or an alternative (usually as a result of backup), uses its argument to restore the state of the parse and attempts to follow an arc, thus creating a new configuration. If no more arcs may be taken from the current state, STEP returns NIL. The functions PARSER, LEXIC, and STEP are shown in flowchart form in Figure 10. (These flowcharts are from [Woods et al, 1972]).

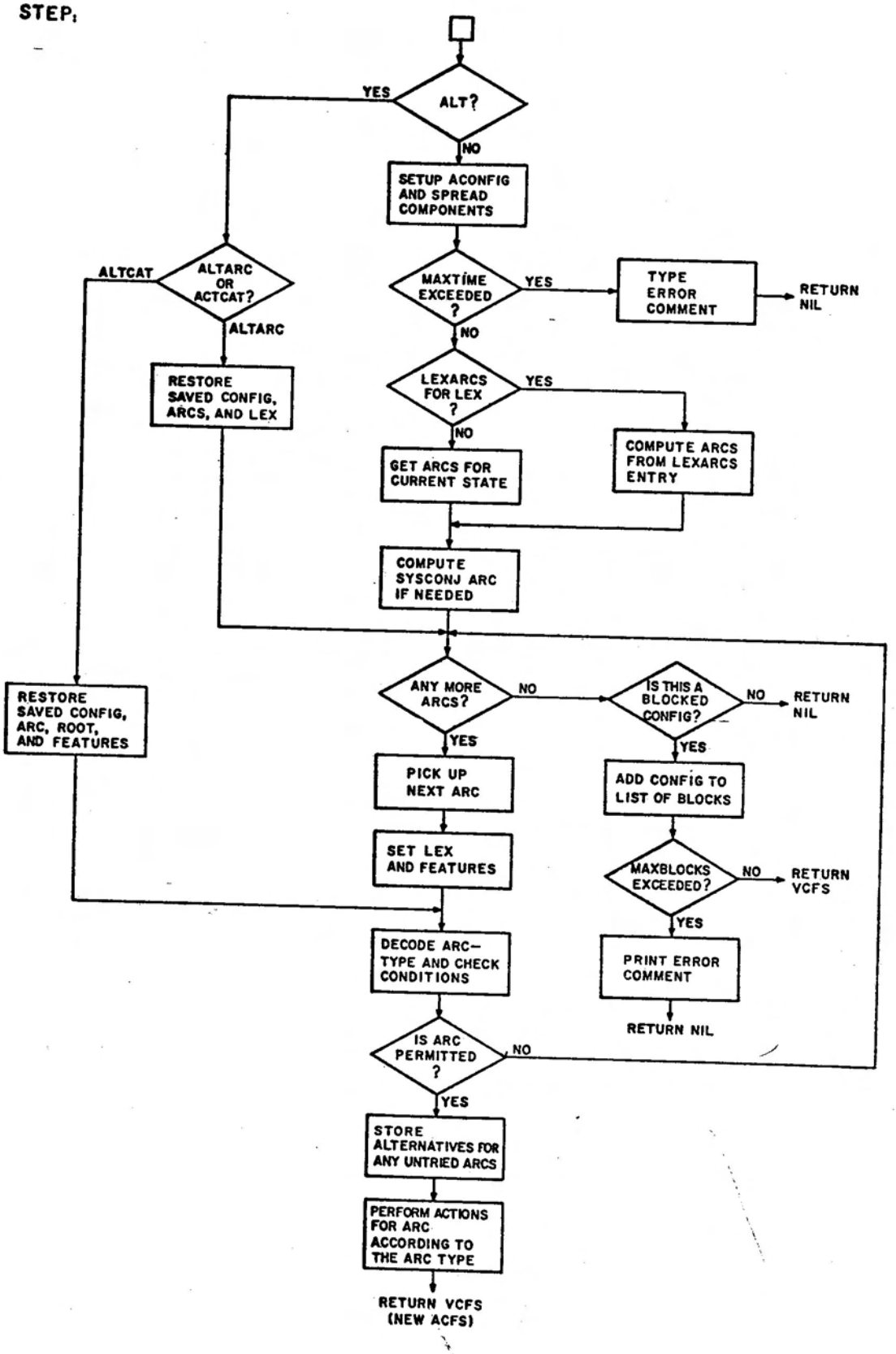


Figure 10c: The Function STEP

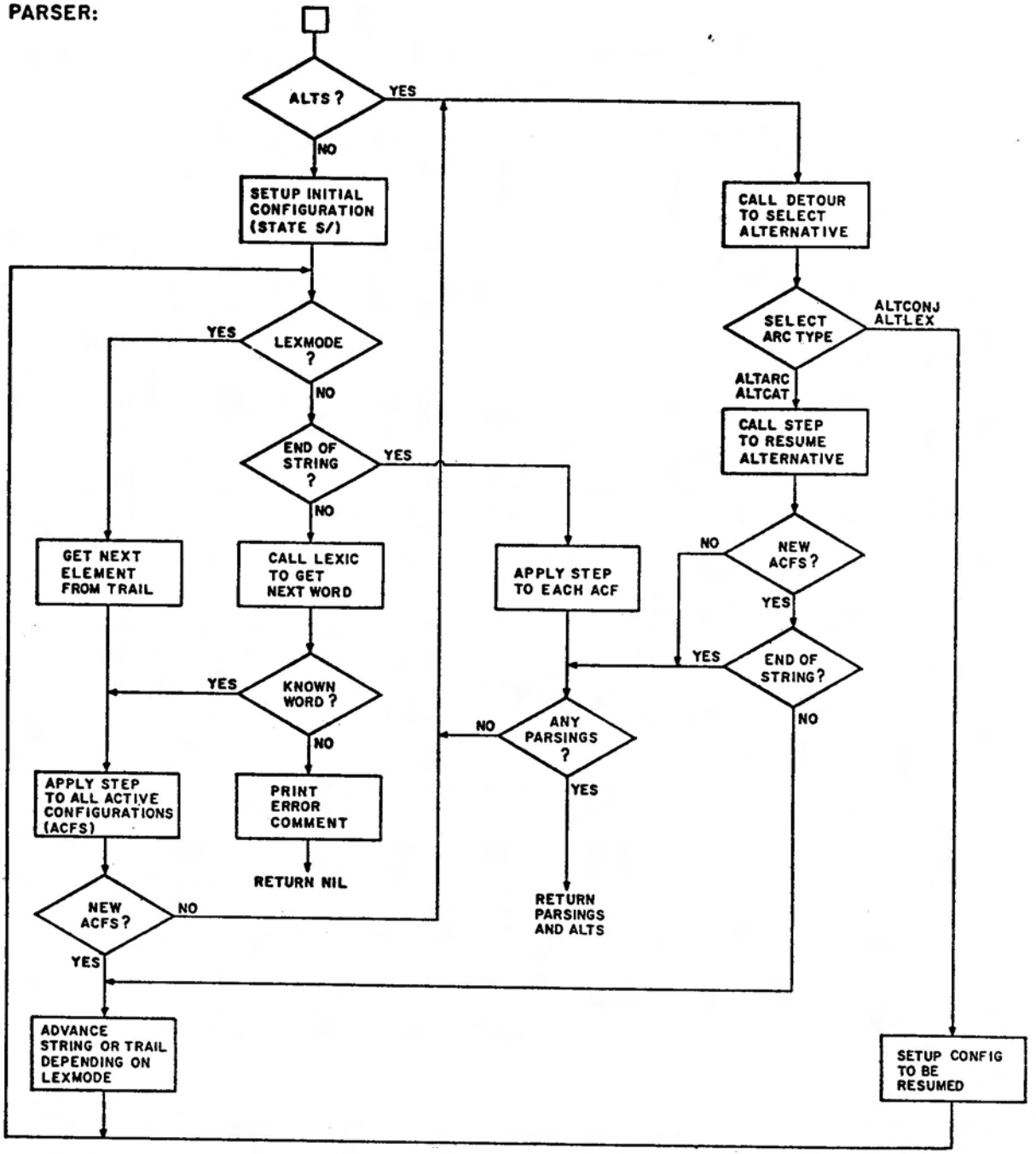


Figure 10a: The Function PARSER

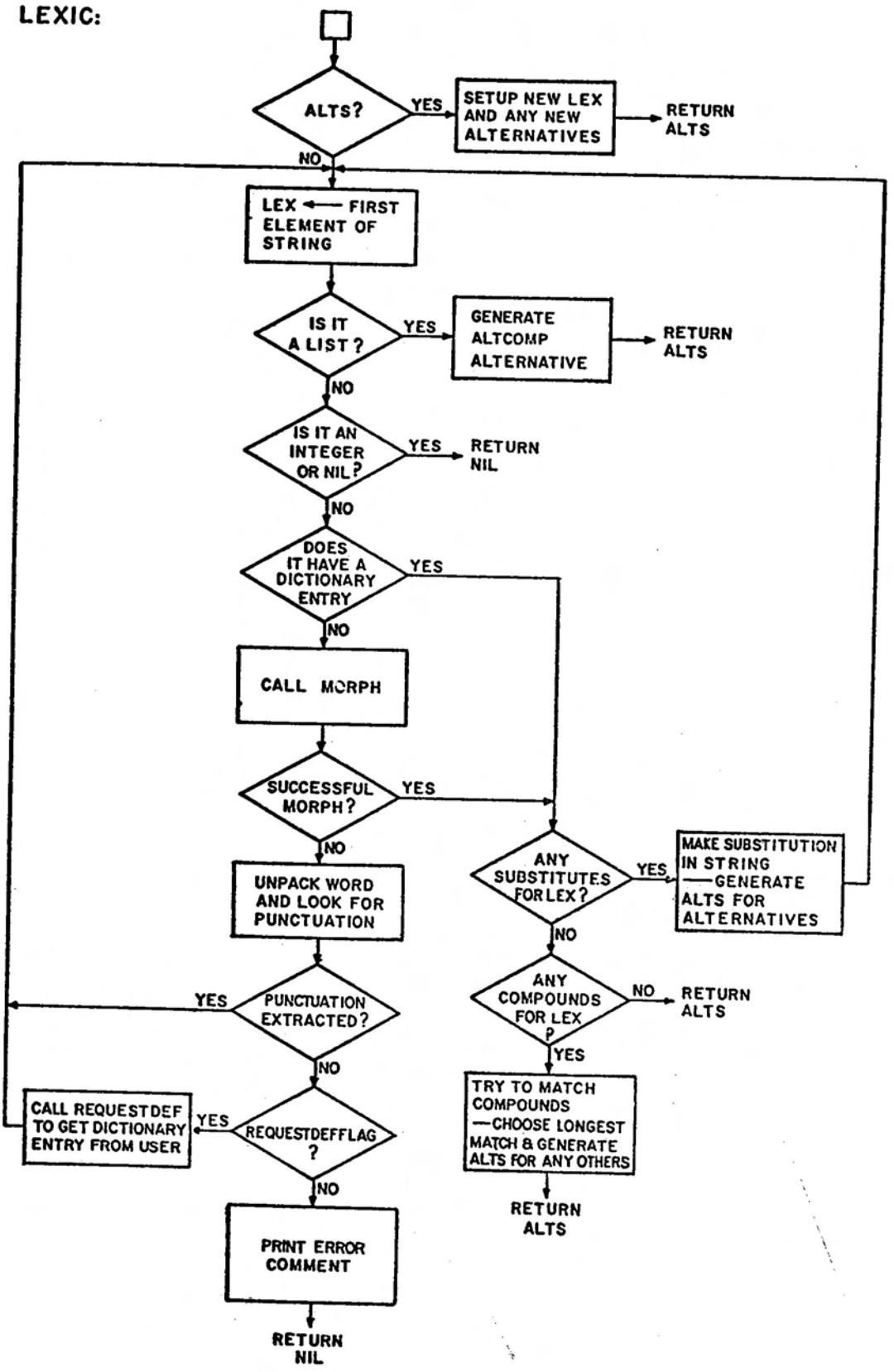


Figure 10b: The Function LEXIC

When PARSER runs out of active configurations which can be STEPed, a function called DETOUR is used to select the next alternative to be tried. If nothing special has been done to give the alternatives special weights, DETOUR will pickup alternatives in the usual backup order.

This parser has the advantage of allowing actions on the arcs to influence the order in which alternatives are chosen. This means that it is not necessary to completely exhaust the search space by following all possible parse paths; by careful ordering of the alternatives, the most likely parsing can be found first, leaving the others to be found later if necessary. Because many sentences and many partial sentences are ambiguous, any natural alnguage parser must be able to cope with alternatives. The ability to order them, together with the alternatives implicit in the factoring and merging of the grammar, help to reduce the number of alternatives which must be considered during a parse.

This parser has also proved to be a flexible testbed for modifications and special features (for examples, see the sections on conjunction and selective modifier placement below.) This parser (compiled in LISP) using the grammar of Figure 9 could parse 8 word sentences in about 2 seconds of cpu time on a PDP-10.

Several important efficiency issues for both parsers should be mentioned. The first is the problem of storing register contents in a space-conserving yet easily accessible manner. Although only a few registers are likely to be used on any one arc, the total number of registers used in the grammar may be quite large. A representation which fixes this number or which requires large numbers of unchanged registers to be copied would be wasteful of both space and time. This problem can be solved by representing a register list as a list of name/value pairs. The function GETR searches the list from front to back for the first occurrence of the named register and returns the associated value. SETR does not change the name/value pair in the current list but instead adds to the front a new name/value pair. This new pair will effectively hide from GETR any old pair with the same name. Thus to preserve the entire register list at a particular point, only a pointer to the current front of the list need be remembered. Other processes can add new information to the list and remember only their new front pointer. Thus the register lists resemble a tree structure with much information on commonly shared

branches; any one process is only able to see its unique path from leaf to root. This makes setting a register much faster than accessing one, but registers are generally set more often than they are retrieved for structure building.

Another way to speed up the parsing process, particularly in cases where a lot of backup is likely to be done, is the use of a well-formed-substring table (WFST). At every POP, the constituent just found together with the portion of input it spans is placed in the WFST. Then whenever a PUSH is encountered, the WFST is checked to see whether a constituent of the desired type has been found at the current place in the input. If so, the constituent may be used directly from the WFST instead of redoing the parsing at the lower level. (The use of the HOLD list complicates the WFST slightly, since a constituent can be taken from it only if the current HOLD list matches the one which was current when the constituent was originally parsed. A similar situation holds for the use of SENDRs.)

3b. Inside-Out Parsers

Two parsers have been developed for an experimental speech understanding system which are quite different from the preceding parsers. They illustrate the extraordinary flexibility of ATNs with respect to the environment where they are applied.

Current and all forseeable acoustic processors cannot uniquely identify all words in an acoustic signal. This is due to problems with homonyms (bear, bare), disputable word boundaries (tea meeting, team meeting, team eating), phonological effects (why choose, white shoes), incomplete pronunciation of most function words in context (of, have, a, the), low energy at the beginning and end of an utterance, and errors induced by the acoustic analysis and word matching process. Thus it is not necessarily the case that a speech understanding system should attempt to process an utterance left to right; it may be better to begin in the middle with a reliably identified long content word and work from the inside of the sentence out to the ends. In this way, the grammar can also provide predictions about what can be adjacent to a portion of the sentence already processed, and these expectations can be used by the rest of the system in its analysis of subsequent (or previous) portions of the utterance.

With these constraints in mind, a parser was developed [Bates, 1975a and b, 1976] which could start parsing anywhere in the input stream and could parse despite the lack of certainty as to the exact nature of the words at each point in the input. As partial parse paths were built up, their pieces were stored in tables so that any other parse that could use them did not need to reparse common sections of input. (This is like a WFST for partial paths rather than complete constituents.) Using the grammar, the parser could make predictions about the words of lexical classes that could be used to extend a sequence of words either to the right or to the left. If a gap between words was small enough to contain just one word, the parser could predict just the class or classes of words to fill the gap. The control structure of the parser could be modified fairly easily to allow experimentation with various combinations of backup, sequential, and parallel search. It used a combination of depth-first and breadth-first techniques, usually following a single path but splitting into parallel paths when desirable. Care was taken to allow the parser to interact frequently and easily with other components of the system (notably semantics) in order to receive guidance and to verify completed constituents.

As part of its initialization, the parser set up an index into the grammar so that, for example, all arcs which consume nouns could be easily located. Then, if a noun in the middle of the utterance were presented to the parser, it would retrieve those arcs and set up a partial parse path for each of them. By adding onto these paths as new words were added (and by using the grammar to make predictions to the right and the grammar index to make predictions to the left) the "island" of words could be grown until it spanned the entire utterance. Careful use of a well-formed-substring table and the maintenance of numerous alternate possible partial paths allowed a very general middle-out parsing algorithm.

A second, faster, more efficient parser was later built for the same system [Woods et al, 1976]. It too was designed for an island driven strategy. It did less interpretation of the grammar while parsing than the previous system because it pre-processed the grammar to obtain a set of useful relations between states. (For example, the relation (S1 J S2) holds if there is a path made up exclusively of JUMP arcs from state S1 to S2.) The pre-processor created arrays which described the grammar in a bi-directional way rather than the in the inherently left-to-right way it is written. The sets of relations

were used by the parser in a way similar to the L set in Earley's algorithm for context free grammars [Earley, 1970] and a technique similar to Earley's for eliminating the stack and performing indirect PUSHes was used.

This parser had the advantage of being able to follow efficiently all possible partial paths in parallel, thus eliminating the need to try to calculate which paths were most likely to be correct. It could also do more context sensitive checking and register setting, thus allowing very good predictions to be made at the ends of islands. These predictions could also be tightened as more information was added to the utterance.

The only accommodation that the grammar writer needed to make to this system was to mark actions which used register contents with the state or states where the original setting could have taken place. (These "scope" declarations were used to determine when sufficient context is present to perform a test or to do an action.) This was a small price to pay for the speed and accuracy of the parser's predictions.

3c. A Grammar-Compiler

A system has been written by R. Burton [Burton, 1976; Burton and Woods, 1976] which produces an extremely fast ATN parser by a two step process. In the first phase, an interpreter converts (compiles) the ATN grammar into a single LISP function. The second phase is to compile this function, thus producing a compiled program which is the parser/grammar combination. This process is schematically illustrated in Figure 11. The operation of the resulting program is shown in Figure 12. (Both figures are from [Burton, 1976].)

The LISP function which is produced by the first phase is a PROG in which the state names of the grammar become labels and the tests and actions are the "statements". The function looks like this:

```
(LAMBDA (ACF)
  (PROG (special variables like STATE, STACK, REGS, HOLD,  , LEX)
    SPREAD-ACF  (code to set up current configuration)
```

```
                (GO EVAL-ARC)
NEXTLEX         (if (another word?) then (advance input)
                                         (GO EVAL-ARC))
DETOUR          (if (another alternative?) then (ACF-alt)
                                                (GO SPREAD-ACF)
                                           else (RETURN failure))
EVAL-ARC        (BRANCH STATE arclabel1 arclabel2 ... arclabeln)
arclabel1       (code for arc)
arclabel2       (code for arc)
...
arclabeln       (code for arc))).
```

The arc code for JUMP, WRD, CAT, and VIR arcs generally looks like:

```
(if (arctype and test satisfied?)
    then (create alternative for remaining arcs)
         (do actions)
         (DOTO nextstate)
         (GO nextstate-arclabel1))
```

The function DOTO changes the state and advances the input. If the arctype or test is not satisfied, the function "falls through" to the next arclabel, which is the following arc. The arc code for the last arc of a state must have a "else (GO DETOUR)" clause.

PUSH arcs generate arc code to recursively invoke the grammar:

```
label  (if (test satisfied?)
           then (create alternatives for remaining arcs)
                (DOPUSH pushstate remaining-actions-label)
                (GO pushstate))

remaining-actions-label
       (do actions on PUSH arc)
       (DOPTO nextstate)
       (GO nextstate-1starclabel)
```

The function DOPUSH saves the current configuration (with the remaining-actions-label as the current state) on the stack and does any pre-actions to initialize registers. The lower level is started

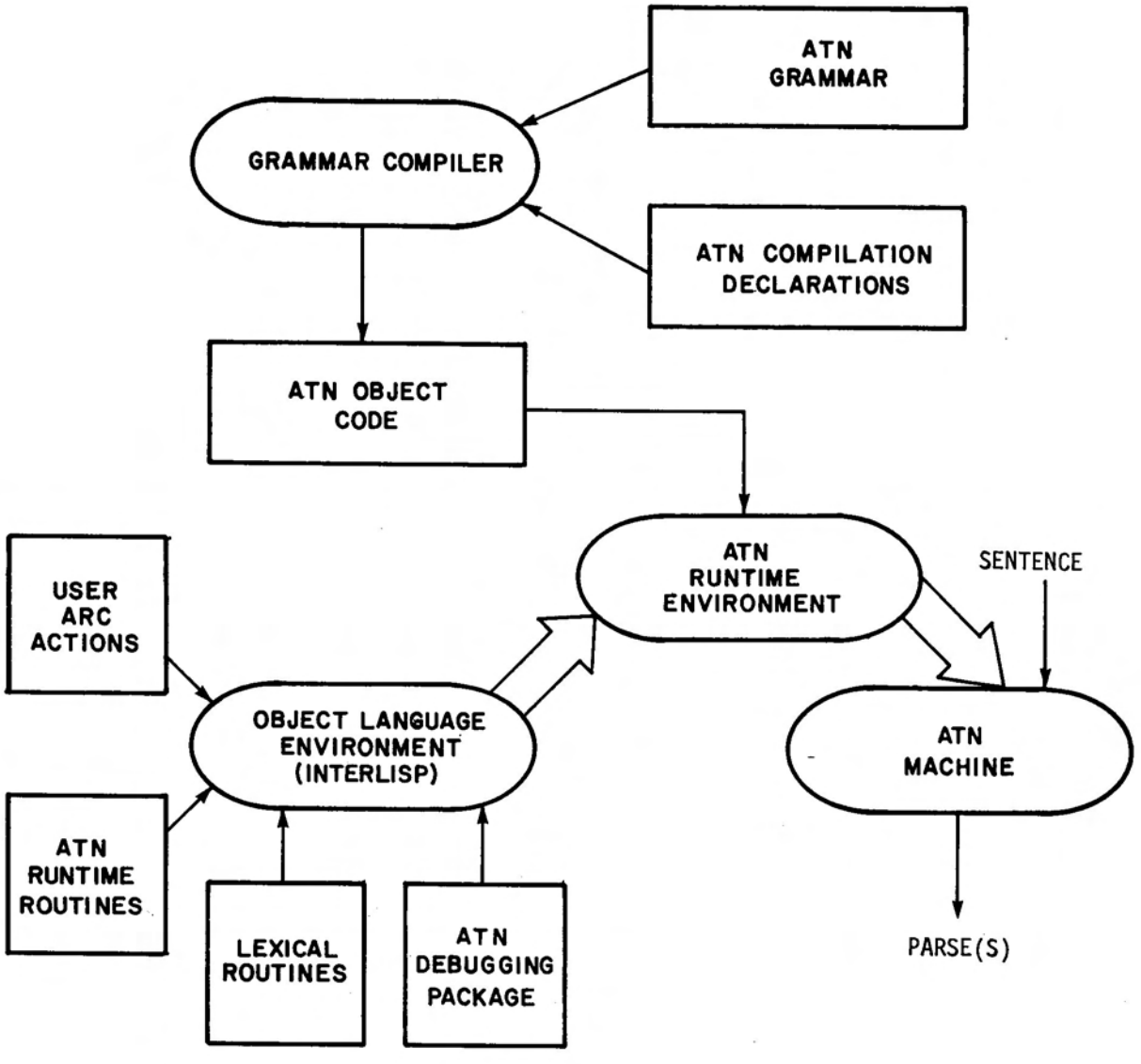


Figure 11: The Operation of a Grammar-Compiler

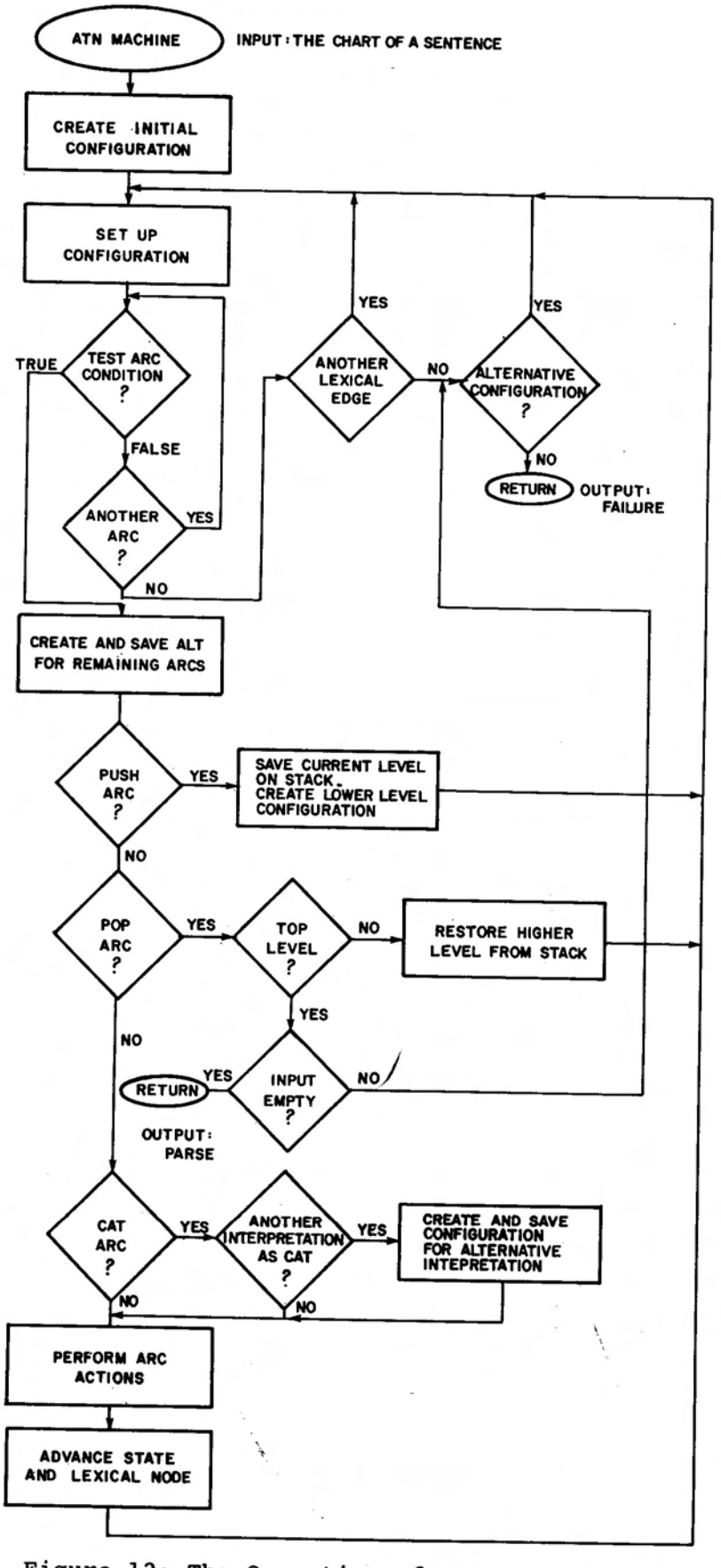


Figure 12: The Operation of an ATN Parser

by the GO to the arc label of the state. When and if the lower level finishes, control will transfer to remaining-actions-label where the actions are performed, the state (but not the input) changed by DOPTO, and the next state is begun.

POP arcs produce code like:

```
(if (test satisfied?)
    then (create alternative for remaining arcs)
         (DOPOP form)
         (GO EVAL-ARC))
```

DOPOP builds the structure to be returned, restores the higher level by getting information from the configuration on the top of the stack, and sets to the value of form. The branch at EVAL-ARC activates the state which was saved (remaining-actions-label).

This "parser" executes very much faster than the previously described methods, about 10 times faster than the compiled version of the LUNAR parser with the same grammar, parsing 8-12 word sentences in about 150 milliseconds.

For production programs where speed is essential, this method is extremely good. The compiler can check, for example, whether the test on an arc is T and if so it can omit the generation of code to evaluate the test; this sort of checking eliminates much inefficiency. The compiler allows unused features of the ATN formalism (recursion, WFST, alternate lexical interpretations, etc.) to be removed, thus improving the efficiency even more. It has the disadvantage that whenever a change is to be made to the grammar both phases of the compiler must be redone -- a time consuming process, especially if the grammar is large. There are also few tracing or debugging aids in the final version. A method similar to this one could be used to write an ATN system in a language which does not have an EVAL function.

4. VARIOUS TYPES OF ATN GRAMMARS

In this section we explore some of the tradeoffs and decisions which must be made by someone who wants to write an ATN grammar for a

particular purpose. As was noted earlier, there is no one grammar or style of grammar which suits every purpose. The linguist who is interested in testing a theory of language will write a very different grammar from a systems programmer who needs a small, fast natural language front end for a programming system, and both of those will differ from a grammar written for instructional purposes. Here are some of the issues involved.

4a. Parsing vs. Generation

Computational linguists and those who want to build a "natural language front end" for a system tend to think of a grammar as something which will be used exclusively for analysis, i.e. parsing. However there are many cases where it is useful to generate English text, as in some question answering systems and computer assisted instruction programs. It is not at all clear whether the same grammar can or should suffice for both analysis and generation, but because an ATN grammar is written separately from its parser, it can be thought of as a form which is independent of analysis or production.

A generator (rather than a parser) may be written which takes an ATN grammar and a dictionary as data and produces sentences. Beginning with the initial state of the grammar, the generator can randomly select an arc from that state to be followed. The following of PUSH, POP, VIR, and JUMP arcs would be nearly the same as in a parser, but WRD and CAT arcs would try to select a word from the dictionary which satisfies the conditions on the arc.

Of course, the method just outlined will produce random sentences since it is in no way guided by intentions or concepts. It may be useful, however, to test grammars which are supposed to be "tight" in the sense of being able to identify and reject incorrect input. Such a generator was written to help debug the ATN grammar for the BBN speech understanding system and was very helpful in the discovery of sentences which would (erroneously) have been accepted by the grammar.

One generation scheme [Simmons, 1973; Simmons and Slocum, 1972] has been proposed which is driven by a semantic network and a grammar very similar to a BTN grammar. A similar but more general system [Shapiro, 1975] uses an ATN grammar whose input is a node of a

semantic network and whose output is a linear string, a sentence describing the node.

If a grammar is to be written exclusively for parsing, one can usually make the assumption that the input will be correct according to the common rules of English syntax. (This may not always be the case, for example, in a computer assisted instruction system which is supposed to check and correct input from students or in a system which is expected to parse a naturally spoken dialog or transcript of such a dialog as opposed to written text.) If one can make the assumption of correct input then in many places the grammar can be simplified by allowing it to accept incorrect input which it will not encounter in practice. A grammar which is written to generate English must contain a great many checks to eliminate incorrect combinations. A grammar which is designed to help choose among many conflicting possible inputs must make extensive tests to screen out the incorrect input.

4b. Competence vs. Performance

Linguists have long made a distinction between language as people actually use it (performance) and language as one may ideally abstract it (competence). As an example, it is in the realm of competence that one knows that a reduced relative clause may be used in a noun phrase, but it is a fact of performance that this embedding is usually performed only once and in the subject position:

The girl the man kissed screamed.
The girl the man the mugger robbed kissed screamed.

Competence tells us that the verb of a sentence must agree in number with the subject and sometimes with the object, yet sentences such as the following are often spoken:

The boys in the band plays at the football game.
That's them!

There is a rule (competence) which says that a particle associated with a verb (as in "call up") may be moved beyond the object of the verb, yet if the object is very long or complex most people would say the sentence is bad.

I called up my neighbor.
I called my neighbor up.
?I called my neighbor in the town where I used to live before financial problems forced me to move to Chicago last month up.

Most systems must deal with some aspects of performance which are outside the formal competence model. For this reason, the term "ungrammatical" becomes ambiguous when referring to the input of a parsing system. A sentence may be conventionally ungrammatical ("A apples falls") but may still be accepted by the grammar, or it may be conventionally correct yet be rejected by the grammar.

4c. Syntactic vs. "Semantic" Grammars

For many particular applications, one can take advantage of key words or classes of words and the small range of likely syntactic constructions by restricting the grammar to accept sequences which are correct not only syntactically but also semantically and pragmatically.

It is possible to use a dictionary which classifies words not (or not only) by their syntactic parts of speech but also by relevant semantic groupings. For example, in a system to parse sentences about people at a zoo, lexical classes such as ANIMAL, PEOPLE, ANIMAL-ADJ, and PEOPLE-ADJ could be used. They would contain words like (bear, goat, lion), (boy, father, woman), (hungry, caged, fierce), and (hungry, naughty, educated, vain) respectively.

Figure 13 shows portions of two different semantic grammars. Of course, a grammar may blend syntactic and semantic categories to any desired extent.

A purely syntactic grammar would be confined to the usual parts of speech and would accept a large number of syntactic constructions which were meaningless ("Colorless green ideas speep furiously"). In a limited domain of discourse where the input can be assumed to be meaningful, a semantic grammar is a very efficient solution to the parsing problem. It is also useful for the generation of meaningful sentences. See [Burton, 1976] for the description of a very efficient system using such a grammar.

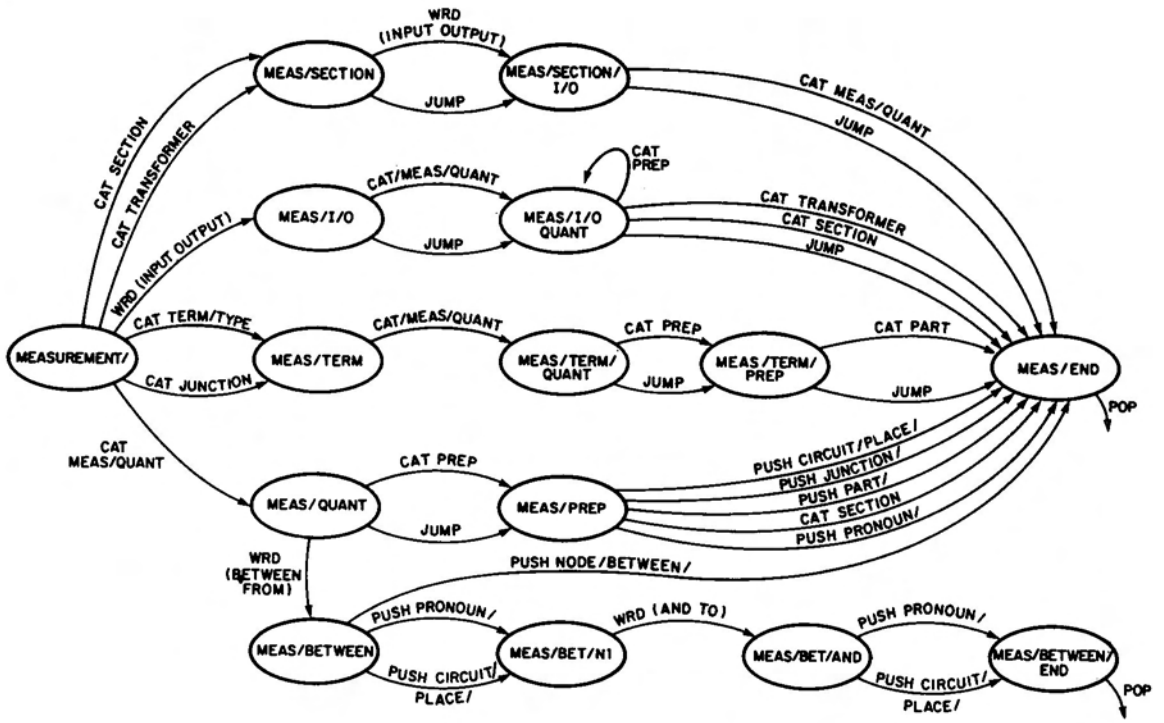


Figure 13a: A Semantic Grammar for Circuit Measurements

Semantic grammars tend to be much larger than syntactic grammars which accept the same set of sentences. The largest ATN grammar this author knows of is one she wrote for the BBN speech understanding system [Bates, 1975]; it contained 448 states, 881 arcs, and 2280 actions but was more limited in the variety of constructions it could accept than a 83 state, 202 arc, 386 action syntactic grammar for the same system. Another drawback to a semantic grammar is that it must be written anew if the domain of discourse is changed, and it would be extremely impractical to attempt to write such a grammar for anything but a limited application area.

4d. Semantics, Pragmatics, and Prosodics

Because syntax, semantics, and pragmatics interact in a very complex way in natural languages, it is desirable to attempt a similar fusion when modeling linguistic processing.

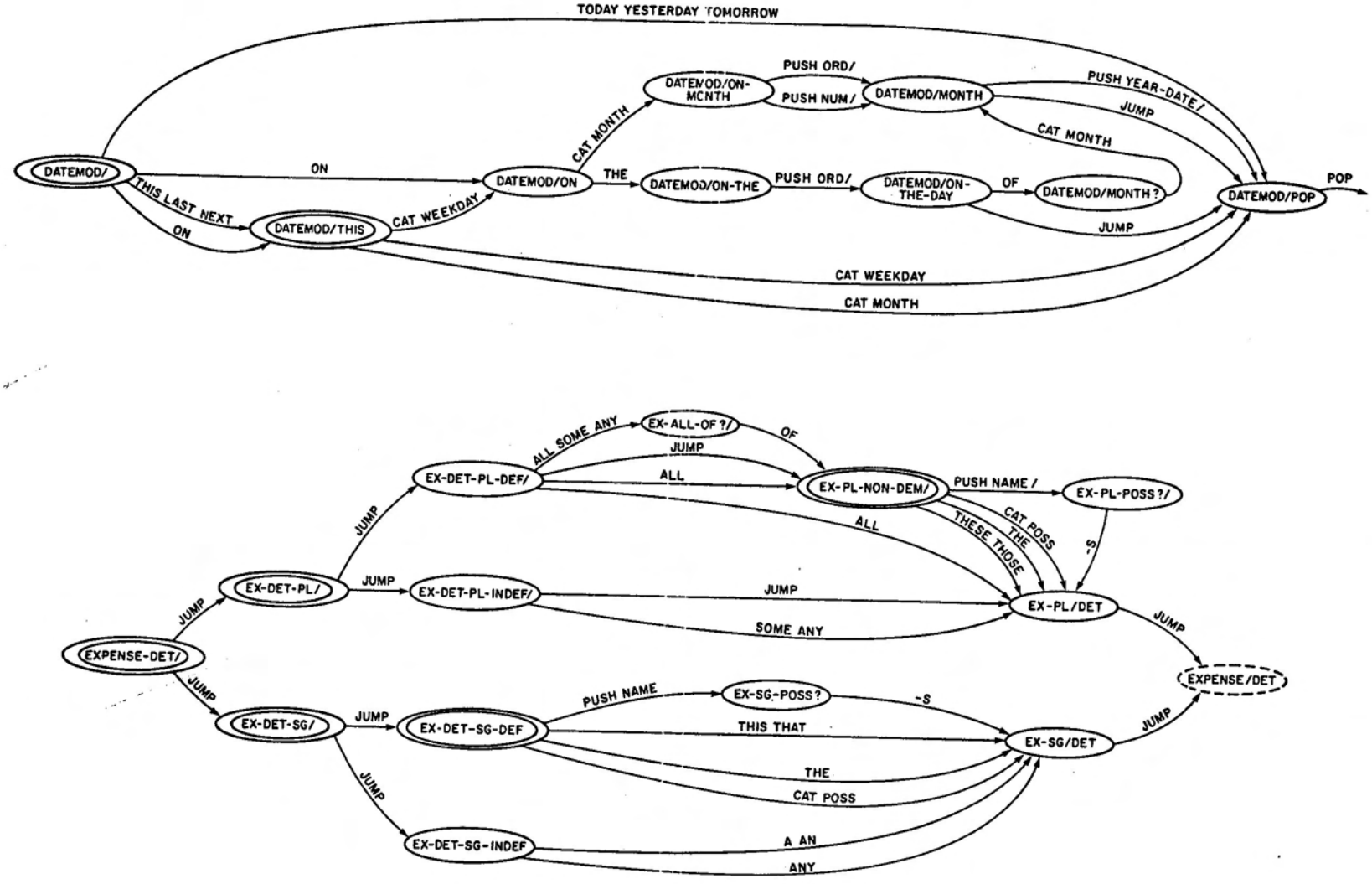


Figure 13b: A Semantic Grammar for Dates and Travel Expenses

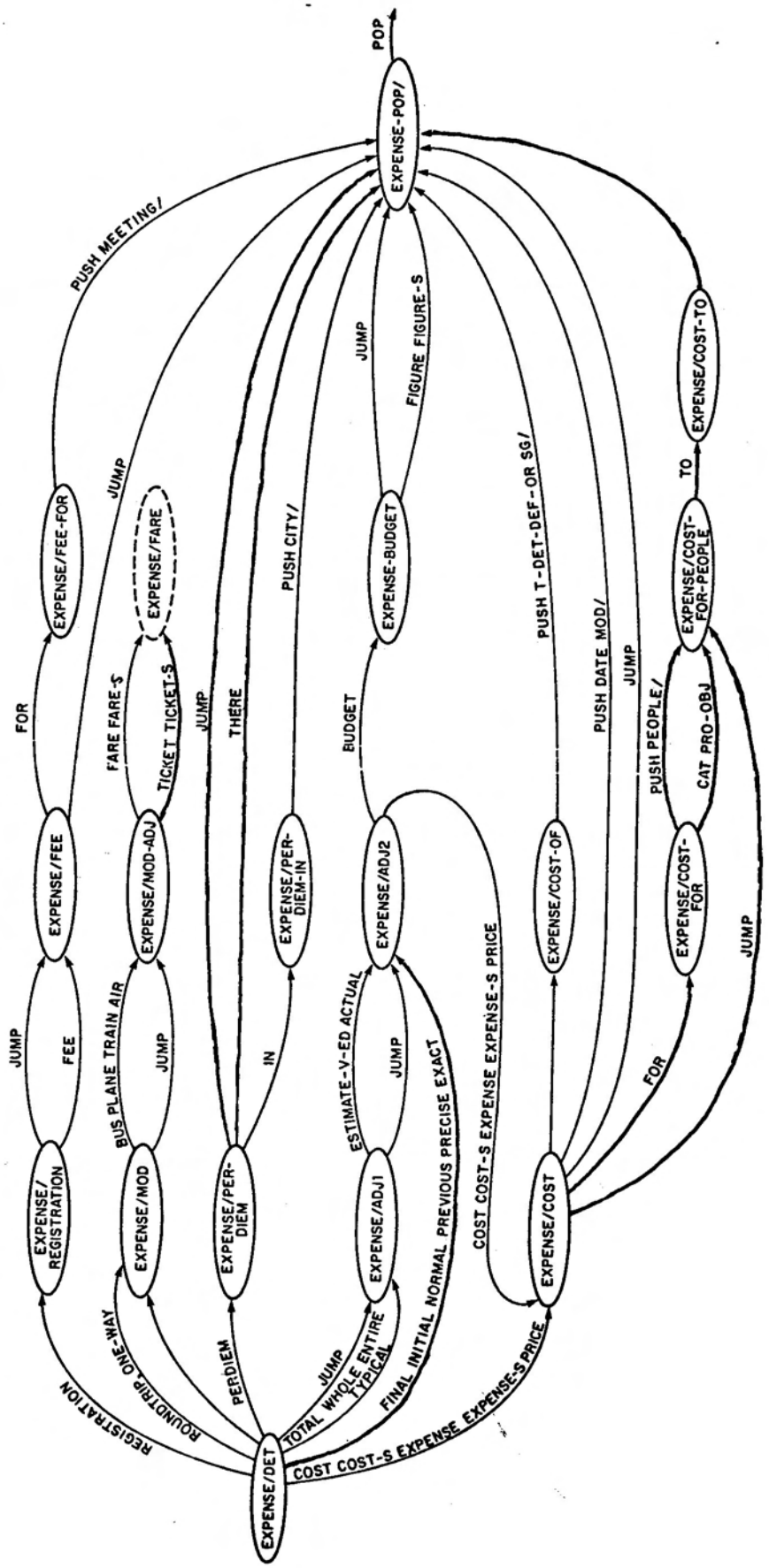


Figure 13b (con't): A Semantic Grammar for Travel Expenses

As was described above, it is possible to place semantic constraints on the grammar by using semantic lexical categories, but this is not the only way to achieve the desired synthesis. Tests and actions on arcs may call semantic routines to check any number of constraints such as the ordering of adjectives within a noun phrase and the agreement between modifiers and the things they modify. The parser may also be modified to call a semantic verification routine to test the completeness or meaningfulness of each constituent as it is produced by a POP arc.

If the parser is structured to allow the order of alternatives to be modified, the semantic functions called on the arcs can be used to effect the modification. Similar functions may be used to detect prosodic features in speech, punctuation in text, and even non-linguistic events (e.g., body actions) in an appropriate environment.

Weischedel [Weischedel, 1976] demonstrates an ATN to compute the presuppositions and entailments of fairly complex sentences. Consider the following sentences:

a. The professor doubted that John managed to translate an assignment.
b. The professor is human.
c. The professor believed that John attempted to translate some assignment.
d. The professor believed that it is not the case that John translated some assignment.

Sentences b and c are presupposed by a, while a entails d. Since presuppositions and entailments are determined by both the meaning of certain words and the syntactic constructs used, it is necessary to have special entries in the lexicon as well as an ATN which can accept the relevant structures.

For systems dealing with speech input, prosodic information is very important. In the BBN speech understanding system [Woods et al, 1976] a special pseudo action called PBDRY was placed on arcs which consumed words before which a prosodic boundary was expected. This function would in effect change the score of the process depending upon whether or not a prosodic boundary was actually detected in the expected location. Prosodic information provides good clues (for

humans) about syntactic boundaries, sentence type (for example imperative vs. a yes-no question), pronominal reference, and other things, but very little is currently known about how to automatically detect and use such information.

It is not always necessary to have a tradeoff between the efficiency of a semantic grammar and the compactness and generality of a syntactic one. A system [Bobrow and Bates, 1977] is currently being built which uses a very general syntactic ATN grammar together with a case-oriented dictionary (that also contains semantic interpretation rules). The grammar uses case information to help guide the parsing, and semantic interpretations can be done as soon as a constituent is complete. This grammar is of moderate size (75 states, 153 arcs) but, using Burton's grammar-compiler, has parsed and interpreted 20-word sentences in under one second.

5. LINGUISTIC ISSUES

There are a number of syntactic structures which have been studied in varying degrees of depth by linguists. The current paradigm of linguistic theory, transformational grammar, is not very suitable for either theoretical or applied computational purposes. ATNs provide a model which captures many of the ideas of transformational grammar in a computationally efficient and theoretically interesting way. While it is not necessary for every ATN grammar to account for all the structures which may be used in a natural language, and in fact most grammars written for practical use will have to include ad hoc methods to deal with those which have not been studied linguistically, it is important to khow that many of the generalizations captured by the theory of transformational grammar can be handled smoothly by an ATN grammar. The following are some such issues.

5a. Extraposition

In many English sentences, a constituent may be moved from its deep structure position up and to the left, out of its original

subtree to a place in a dominating tree. The following sentences are examples:

This is the cat that I was afraid someone was going to try to steal.
The man in the red shirt is the one who John wanted to speak to.

The problem with this construction is that the constituent when found during parsing cannot just be put in a register to be picked up and used later because the place at which it is likely to be needed is one or more levels down; by the time that the proper place in the input is reached, the constituent is hidden and inaccessible on the stack.

The HOLD action and VIR arcs may be used to handle this problem in an efficient, uncomplicated way. The HOLD action associates a constituent with a name and places this pair on a special list called the HOLD list. The HOLD list is in effect a global variable which is accessible at all lower levels. A VIR arc on the original level or at any lower level may remove a constituent from the HOLD list and use it just as if it had actually occurred at the current position in the input string. To insure that held constituents are used in constructions dominated by the one in which they are found, the parser must be modified so that every POP arc has an implicit test which will make the arc fail if any constituents which were placed on the HOLD list at the current level remain on it.

The HOLD-VIR mechanism is not absolutely necessary to deal with left extraposition. It would be possible to put the extraposed constituent in a register which was then sent down every time a PUSH were done. JUMP arcs could then test for that register and insert the constituent in its proper place. Every POP arc would have to lift an indication of whether or not the constituent had been used. This is a complicated, error-prone procedure which does not have the clairty of the previous mechanism.

Another way to avoid using the HOLD-VIR mechanism is to use in place of VIR arcs JUMP arcs which put an identifiable "dummy node" into the constituent which is being built. A tree with such a dummy node is given in Figure 14. At a higher level (on the PUSH arc on the level where the HOLD action would have been done) a copy of the structure returned from the lower level can be made, substituting the appropriate structure for the dummy node. An advantage of this method over the one just described is that the constituent with the dummy

node may be placed in the WFST for use by other paths, which may want to substitute a different structure for the dummy. However, there are numerous disadvantages compared to the HOLD method; an explicit test must be made to avoid returning a constituent with a dummy node from a level where it should have been replaced, much time may be wasted constructing constituents with dummy nodes which cannot be replaced at higher levels, and it is more difficult to do agreement tests between the extraposed portion and the rest of the constituent.

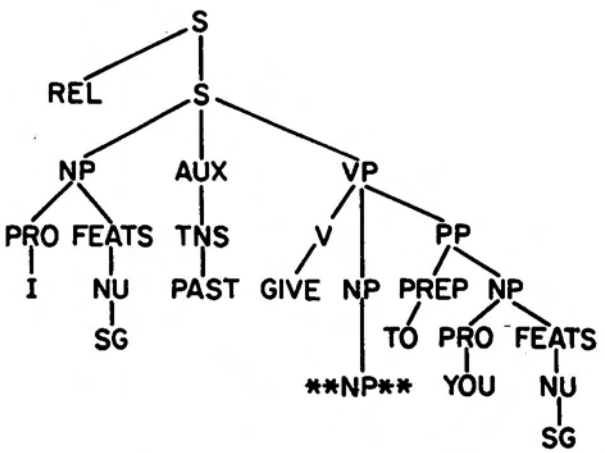


Figure 14: A Relative Clause Tree with a Dummy Node

Extraposition may happen to the right as well as the left. In sentences like

The resort I went to which was on the Mediterranean was pleasant.
How many chickens were there which crossed the road?

a complete constituent is moved out of another constituent (which is still completely well-formed despite the loss) to a position farther to the right and above the original constituent.

To handle this right extraposition, two new actions are used. The action (RESUMETAG <state>) creates a marker combining the current register list with the named state at which parsing could continue at some later time. Then at some later time, if the action (RESUME) is encountered on an arc, the marker is retrieved and the configuration the parser was in when the RESUMETAG was executed is re-established.

The extraposed text can be parsed as usual, and when a POP is done the completed constituent is used as the current input item on the arc containing the RESUME.

5b. Conjunction

Conjunction in English (and many other languages) is a problem for both the theoretical linguist and the applied computational linguist. In its simplest form, two complete constituents are conjoined; at its most complex, numerous constituents can be conjoined in an overlapping manner. The following sentences illustrate the range of complexity.

The quick brown fox and the lazy dog are famous.
Enclose this message with the red, orange and blue packages.
Will you plant the roses or cut the grass?
I groomed and brushed my cat.
The easy homework and exams made the course hard to fail.
She often fought with, later married, and soon divorced the boy next door.

If only complete constituents could be conjoined (either by a single conjunction, a list with commas having the conjunction before the last conjunct, or a list with the conjunction between all conjuncts) then any type of constituent X could be made conjoinable by having a new level of the grammar called XLIST. XLIST/ would be PUSHed to by all the arcs which formerly PUSHed to X/. The XLIST/ level would simply PUSH to X/, accumulate the result in a register, and either POP or find a conjunction (or comma). In the latter case control would transfer to XLIST/ to look for the next conjunct. If more general conjunction is allowed, for example, "and," "or," and "not" with different precedence, then a more complex grammar would be required.

However, general conjunction which can violate constituent boundaries cannot be conveniently handled this way. It is possible to add a facility to the ATN parser which provides some insight into the general process. This facility, called SYSCONJ in [Woods, 1973] handles reduced conjunctions and constructs the appropriate unreduced deep structures without requiring any changes to the grammar.

The most general kind of conjunction is the reduced conjunction as seen in "John seldom talked about and tried to forget the war." In this case the conjoined fragments "... seldom talked about" and "tried to forget ..." are not constituents, nor are they similar at either their beginning or their end. There is, however, a great deal of structure to conjunctions of this type. There is a place in the first conjunct (after "seldom") where the parser could begin parsing the second conjunct, and there is a place in the second (after "forget") where the parser is in the same state it was in at the end of the first.

Because of this regularity, the SYSCONJ facility may be built into the parser to intercept a conjunction before the grammar processes it. At this point the parser has available to it the entire history of the parse up to that time in the form of configurations on the path and stack. SYSCONJ selects one of these configurations (how it decides which one can be a deterministic, non-deterministic, or semantically motivated decision) and restarts it using the string following the conjunction. The configuration which the parser was in when the conjunction was encountered is saved. The restarted configuration will allow the parser to proceed until it reaches some portion of the input which is shared with the suspended configuration. Then the paths can be combined and the constituent completed. If the restarted configuration cannot find a point at which the suspended configuration can join it, it is an indication that the original selection of a restart configuration was in error, and another one may be tried.

This method is potentially very combinatorially explosive, but it may be guided by some syntactic heuristics such as restarting just before a word which is identical to or of the same category as the word after the conjunction (e.g., "The dogs in the yard, on the porch, and in the house barked"). Semantic guidance may be useful also, but this has not been investigated in depth. It also will not work for embedded complex structures. Further study and experimentation with this feature are necessary to show the limits of this approach to the problem.

5c. Punctuation and Contraction

Usually punctuation is an aid to parsing because it can be used to delimit the scope of constituents, as in

John, the plumber, fixed it.
He eats fried eggs, over-ripe bananas, and soft custard.

It is useful to have a prepass to scan the input to separate punctuation from the preceding word. Then an arc like WRD , or WRD ; or CAT PUNCT may be put in the grammar.

It is tempting to separate endings like "n't," "'ve," and "'s" and turn them into the separate words "not," "have," and "is" in the prepass, but here the problem is more complex. For example, "'s" can be a contraction for "is" or it can be the possessive morpheme ("Helen's jewelry"), and "n't" can change the form of its predecessor ("won't" -> "will not"). There are also restrictions on where contraction can occur ("Mary's eating and Joe is too," but not "Mary's eating and Joe's too"), so information useful to the grammar may be lost by expanding the contraction out of context. One way to handle the latter problem is to attach a feature CONTRACTED to words which have been so expanded.

5d. Modifier Placement

One of the most common sources of syntactic ambiguity in English is the problem of what head is modified by a modifier. This is particularly true of a series of prepositional phrases, and it also occurs with adverbs. Often, but not always, semantics determines the correct attachment:

I saw the man in the { park / dark suit } with { a telescope. / pigeons. }

Paul borrowed the book that { belonged to / mentions } Jane on Friday.

The orchestra performed the music Walter { wrote / likes } recently.

Woods' parser [Woods, 1973] incorporated a selective modifier placement facility that was invoked by a special type of POP arc called SPOP. When an SPOP arc was encountered, the parser would search the stack for other configurations that could use the constituent about to be SPOPed. For the second sentence shown above, an SPOP arc at the end of the PP/ network would find configurations PUSHing for a prepositional phrase (after "Jane"), for a relative clause (after "book"), and for a noun phrase (after "borrowed"). At the configuration for the most recent PUSH, the SPOP process determines that a POP could have been done instead of the PUSH. Then it finds out that the next higher level (the PUSH for a relative clause) could also PUSH for a prepositional phrase. This means that the configuration PUSHing for a relative clause is a candidate for the prepositional modifier.

Continuing up the stack in the same way, a list of candidate configurations is made. Then semantic information associated with the head of the level represented by the candidates is examined to see which ones may be associated with the semantic head of the modifier. (This semantic information may come from the dictionary or from special-purpose functions.) Checks which may be made include: heads which forbid modifiers of that type ("sincerity on the table") heads which require modifiers of that type to make sense ("consort with criminals"), and heads which may use such a modifier ("see with a telescope"). The chosen configuration is the closest one which needs the modifier most. Alternatives are created for other placements (in case of backup or to ensure the eventual production of the less likely ambiguities), and the prefered one is continued.

6. DEVELOPING AN ATN GRAMMAR

This final section is designed for those who wish to write an ATN grammar. Remember that it is not necessary to have a parser in order to effectively use an ATN grammar. One can learn a lot about the structure of a language (even a restricted subset) by going through the exercise of expressing the language in an ATN form. Careful hand simulation of parses is usually sufficient for testing portions of the grammar.

The first step in writing a grammar is to have a clear idea in mind of the types of sentences one would like to handle. What aspects of competence must be handled? Of performance? Is the grammar going to be used to generate or to parse? Make a list of ten or twenty sample sentences. Then decide on the general type of grammar which is desired, for example, decide on a syntactic grammar which will produce stratificational structures.

Next, sketch an ATN diagram of the most common constructions which you find in your sample sentences. After the surface structure has been drawn, add a few tests and actions. It is a good idea to use carefully chosen state and register names and to record on every arc (as a comment) its purpose together with a sample phrase or two which will use the arc. Like completely commenting a computer program as it is being written, this is almost never actually done, but the closer one can come to this ideal the easier it will be to debug and modify the grammar later.

Look for portions of the diagram which are identical. There are two ways to consolidate these: merging by loops and making a new level to be reached by PUSH arcs. Every grammar writer is eventually faced with the choice of whether to create a new level of the network to PUSH to or to use a longer, more complex set of arcs in the original net.

For inexperienced grammar writers, deciding where to separate a grammar into levels can be a problem. It seems natural to say that a noun phrase is a constituent, but what about a verb phrase? a determiner? an auxilary? a reduced relative clause?

Some linguists have fairly fixed ideas about what constitutes a constituent. To them it is a convenient grouping of words which can be moved as a whole (but not in part unless the part is also a constituent) to another place in the sentence and which obeys certain rules with respect to transformational rules. This definition is not very helpful to writers of ATN grammars for applications, however, except that it does indicate that certain groups of words may appear in several places in a sentence. It is more efficient to have a single level of the grammar to process such units rather than to repeat the arcs and nodes comprising them. Sometimes a PUSH is required by the nature of the structure to be parsed, as in noun phrases which may contain prepositional phrases which must contain

noun phrases. In other cases a PUSH may be used merely for convenience when a section of input can be processed relatively independently of any information preceding it.

Now look for parts of the network which are very similar, but not quite identical. Again, there are two alternatives: either merge the nets using registers and tests to keep track of which path is being followed, or make a new level to be reached by PUSHs which use SENDRs to convey the necessary information about the differences. Merging similar networks by using tests is usually desirable since it cleanly and concisely expresses some generalization about the language. If this method is carried to extremes, however, the clarity is lost in the complexity of the tests. One could merge an entire network into one arc with a huge number of conditional tests, but this would not express very much about the language!

The merging of common portions of the network does more than permit a more compact representation; it eliminates the necessity of redundant processing when parsing. When two rules have common parts, by matching (or even attempting to match) the first, one has already performed some of the tests required for the matching of the second. Thus one can take advantage of this information to avoid redoing the work when trying the second rule.

Next, for each of the sample sentences decide on the structure which you want the parser to return. Try to sketch the structure of the major constituents within the sentence. Add more actions to the grammar to produce those structures. When conflicts arise, use a conditional action or use two arcs of the same type but with different tests and actions.

Try to consolidate the network by using self loop arcs with tests to prevent them from being taken more than once. For example, the grammar fragment in Figure 15 may be reduced by one state and one arc if it is represented as it was in Figure 6.

The use of "look-ahead" tests on PUSH arcs is a great time saver, since a lot of work is wasted if a recursive call is set up but fails before it consumes any input.

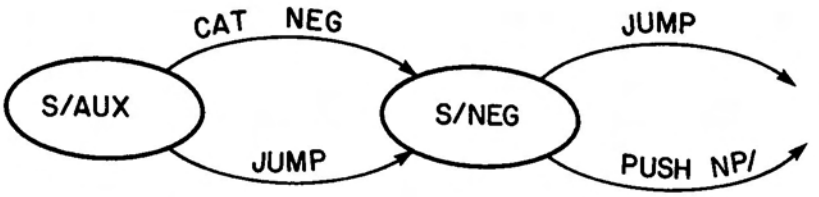


Figure 15: A Grammar Fragment Which May be Reduced

The best way to debug a grammar is to parse a number of sentences and examine the opertion of the parser on those which fail. It is extremely useful to have a tracing facility in the parser which will print out each state entered, each arc taken (not the whole arc, just enough to identify it), each register set, each structure POPed, and each state which blocks.

The trace is useful not only to debug sentences which didn't parse or which parsed and returned the wrong structure, but also to find inefficiencies in the processing of correct sentences. If a sentence parses correctly but requires much backup, there are several things to look for between the point where the parser started down the wrong path and the point where it blocked: Can a test be made at the beginning of the erroneous path that would prevent the arc from being taken? Should the arcs be reordered so that the correct one is taken first? Can the right and wrong paths be merged?

The overriding consideration in a decision among several adequate but different network representations should be the clarity of the method. After all, the ultimate purpose of writing a grammar is to communicate something about the structure of language to other human beings. A grammar writer who always sacrifices clarity for the sake of efficiency will waste extraordinary amounts of time modifying and explaining his work. The programmer's maxim, "Make it work, then make it fast," should be heeded.

One portion of the grammar interacts with many others, so as the grammar gets large it becomes harder to keep track of (or even to discover) the implications of additions or changes. This is where copious commenting of the grammar pays off. It is also a good idea to keep a list of sentences which thoroughly exercise all parts of the grammar. Add to the list whenever new capabilities are added to the grammar and occasionally parse the entire list, just to be sure that what used to parse still does.

As the grammar grows, the dictionary will probably grow at the same time. It is important to keep track of what features the grammar will be testing and how to decide whether a word is to get that feature or not. See Appendix II for a description of the information which may be kept in the dictionary.

The reader who has carefully studied the concepts presented here should now be able to design an ATN grammar and/or parser with which to experiment. It is a rewarding experience to use such a simple yet powerful mechanism. The author would greatly appreciate receiving comments, suggestions, and reports of others' experiences with ATN grammars.

I would like to express appreciation to William A. Woods for his reading of a draft of this paper; the responsibility for errors is my own.

References

Bates, M. "The Use of Syntax in a Speech Understanding System," IEEE Transactions on Speech and Signal Processing, Vol. ASSP-23, No. 1, Feb. 1975, pp. 112-117.

Bates, M. "Syntactic Analysis in a Speech Understanding System," BBN Report No. 3116, Bolt Beranek and Newman Inc., Cambridge, Ma., 1975.

Bates, M. "Syntax in Automatic Speech Understanding," American Journal of Computational Linguistics, Microfiche 45, 1976.

Bobrow, D.G. and Fraser, J.B. "An Augmented State Transition Network Analysis Procedure." Proc. IJCAI, 557-567, 1969.

Bobrow, R. and Bates, M. "The Efficient Integration of Syntactic Processing with Case-Oriented Semantic Interpretation," submitted to the Annual Meeting of the Association for Computational Linguistics, Georgetown University, Washington D.C., March 1977.

Burton, R.R. "Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems." BBN Report No. 3453, Bolt Beranek and Newman Inc., Cambridge, Ma., December 1976.

Burton, R.R. and Woods, W.A. "A Compiling System for Augmented Transition Networks," presented at the International Conference on Computational Linguistics, Ottawa, Canada, June 1976.

Earley, J. "An Efficient Context-Free Parsing Algorithm." Communications of the ACM. 13, 1970, 94-102.

Grebe, K. "Verb Clusters of Lamnsok," in Network Grammars, Grimes, J., ed., 1975.

Grimes, J. "Transition Network Grammars: A Guide," in Network Grammars, Grimes, J., ed., 1975.

Grimes, J., ed. Network Grammars, a publication of the Summer Institute of Linguistics of the University of Oklahoma, 1975.

Leal, W.M. "Transition Network Grammars as a Notation Scheme for Tagmemics," in Network Grammars, Grimes, J., ed., 1975.

Rustin, R., ed. Natural Language Processing. Algorithmics Press, N.Y., 1973.

Shapiro, Stuart C., "Generation as Parsing from a Network into a Linear String," American Journal of Computational Linguistics, Microfiche 33, 1975.

Simmons, R.F. "Semantic Networks: Their Computation and Use for Understanding English." in Computer Models of Thought and Language. Eds. R.C. Schank and K.M. Colby. San Francisco: W.H. Freeman and Company. 1973.

Simmons, R. and Slocum, J. "Generating English Discourse from Semantic Networks," CACM, 15:10 (Oct. 1972) pp. 891-905.

Teitelman, W. INTERLISP Reference Manual. Xerox Palo Alto Research Center, Palo Alto, California, 1974.

Thorne, J.P., Bratley, P., and Dewar, H. "The Syntactic Analysis of English by Machine," in Michie, Machine Intelligence 3, pp. 281-309, 1968.

Weischedel, R.M. "A New Semantic Computation While Parsing: Presupposition and Entailment." Technical Report 76, Department of Information and Computer Science, University of California, Irvine, California, 1976.

Weissman, C. LISP 1.5 Primer, Dickenson Publishing Co, Belmont, Calif., 1967.

Woods, W.A. "Augmented Transition Networks for Natural Language Analysis." Harvard Computation Laboratory Report No. CS-1, Harvard University, Cambridge, Ma., 1969. (Available from the National Technical Information Service 5285 Port Royal Rd., Arlington, Va., 22209, USA, as Microfiche PB-203-527; also available from ERIC, PO Box 0, Bethesda, Md., 20014, USA as publication ED-037-733)

Woods, W.A. "Transition Network Grammars for Natural Language Analysis." Communications of the ACM. 13(1970), 591-606.

Woods, W.A. "An Experimental Parsing System for Transition Network Grammars." Natural Language Processing, Randall Rustin, ed., New York: Algorithmics Press, 1973.

Woods, W.A., R.M. Kaplan, and B. Nash-Webber, "The Lunar Sciences Natural Language Information System: Final Report." BBN Report No. 2378, Bolt Beranek and Newman Inc., Cambridge, Ma., 1972. (available from the National Technical Information Service as publication N72-28984)

Woods, W.A. et al, "Speech Understanding Systems, Final Report Vol. IV (Syntax and Semantics)," BBN Report No. 3438, Bolt Beranek and Newman Inc., Cambridge, Ma., 1976.

APPENDIX I

SOME COMMENTS ON LISP

For those readers not familiar with LISP, a brief explanation may be necessary to make the text of the ATN grammar fragments intelligible.

Since programs and data are interchangeable in LISP, we will speak of either of them as "forms." Forms in LISP are either atoms (alphanumeric sequences like NOUN, COND, 23, and MOD4) or lists of forms like (NOUNTYPE ABSTRACT), (NP (N MILK)), and (JUMP X T).

A form which is a function call is written as a list whose first element is the name of the function and whose subsequent elements are the arguments. Thus (EQUAL (LENGTH FOO) 7) is the function call that in some other language might be written as EQUAL(LENGTH(FOO), 7) or as LENGTH(FOO) = 7. In this case FOO must be evaluated before LENGTH is evaluated just as the length of FOO must be calculated before EQUAL can operate. Most functions require their arguments to be evaluated in this way; to pass an argument unevaluated the special function QUOTE is used. Thus the form (LENGTH (QUOTE (FE FI FO FUM))) has the value 4 but (LENGTH (FE FI FO FUM)) assumes that there exists a function named FE which takes three arguments and that FI, FO, and FUM are variables with values.

Some functions have the quality of not evaluating one or more of their arguments (i.e. their arguments are implicitly QUOTEd). Such functions sometimes (but not always) have names ending with the letter Q to remind the user of this property (e.g. SETRQ).

Since all operations in LISP are functions and hence must be called in prefix form, some rather unusual forms result from fairly common operations, for example, conditional expressions. In LISP the form:

```
(COND (p1 e11 e12 e13 .. e1n)
      (p2 e21 e22 .. e2m)
      ...
      (pi ei1 ei2 .. eij))
```

is equivalent to

```
if p1 then begin e11 e12 ... e1n end
     else if p2 then begin e21 e22 .. e2m end
                else if ...                .
                        else if pi then begin ei1 ... e1j end.
```

If none of the predicates (p1, p2, ...) are true, then the conditional is simply a no-op. Only one of the expression lists is evaluated, the first one which has a true predicate. The value of the conditional (since it is a function, it has a value) is the value of the last expression e evaluated.

When functions are defined in LISP they are written as Lambda-expressions. This simply means that the defining form is a list which begins with the atom LAMBDA followed by a list of the formal parameters followed by the body of the function. The body is often a PROG form, which is a list beginning with the atom PROG followed by a list of local variables and a sequence of forms to be evaluated. The forms to be evaluated may be prefaced by labels, which are (unevaluated) atoms. Thus the definition of the factorial function in LISP is the following (SETQ is the assignment operator):

```
(LAMBDA (N)
  (PROG  (X)
         (SETQ X 1)
   LOOP  (COND((EQUAL N 1)
               (RETURN X)))
         (SETQ X (TIMES N X))
         (SETQ N (SUBTRACT N 1))
         (GO LOOP)))
```

APPENDIX II

THE DICTIONARY

The dictionary is an important part of any parsing system. It contains information about the lexical categories of words, their features, and perhaps their morphological properties.

Table II shows some of the common syntactic categories of English, and Table III shows a selection of features. Neither of these lists are definitive; many other classification schemes and features may be used.

SYMBOL	MEANING	EXAMPLE
ADJ	adjective	actual, special, nice
ADV	adverb	quickly, again, too
ART/DET	article	a, an, the, this
CONJ	conjunction	and, or, both, either
MODAL	modal	can, could, may, will
N	noun	air, group, stone, year
NEG	negative	not
NPR	proper noun	Jimmy, France, TWA
POSS	possessive	her, my, whose, your
PREP	preposition	by, for, from, in, under
PRO	pronoun	someone, he, it, what, I
QADV	question adverb	when, where, why
QDET	question determiner	what, which, whose
QUANT	quantifier	all, enough, many, much
QWORD	question word	how, what, which, who
V	verb	kiss, want, see, jump

Table II: Typical Syntactic Categories

When the parser asks the dictionary if a particular word is in a particular category (as on a CAT arc or via the function CATCHECK) the answer is not just yes or no. If yes, it is also necessary to return the root form of the word and the set of features which is implied by that morphological form. As examples, the word "tallest" has the root "tall" and the feature SUPERLATIVE; "talks" has the root "talk" and the features (TNS PRESENT), (PNCODE 3SG) when considered as a verb, but as a noun its features are (NUMBER PL).

Feature	Meaning	Sample Words	Sample Sentence
FORCOMP	may take a FOR---TO-- complement	afford,need,plan,want	I'll arrange for him to go.
TOCOMP	may take a TO-- complement	try,begin,continue,need	The boy started to cry.
THATCOMP	may take a THAT--- complement	assume,estimate,show,know	She forgot that the iron was hot.
INDOBJLOW	indirect object may be sent down to lower level	arrange,cost,get	The car cost him $50 to fix.
OBJLOW	direct object may be sent down to lower level	afford,allow,need,send	I planned a trip for John to take.
SUBJLOW	subject may be sent down to lower level	arrange,forget,need	I need to learn to swim.
INDOBJ	may take an indirect object	cost,get,send,show,tell	He gave the dog a bone.
(INDOBJ FOR)	indirect object goes with FOR, not TO	allow,arrange,plan	Find the guests a room.
INTRANS	does not need an object	begin,fly,leave,travel	The snow won't last.
TRANS	may take an object	add,compute,give,know	We finished the work.
PASSIVE	may be passivized	arrange,cancel,do,give	The story was told.
(ROLE OBJ)	may not appear as subject	him,them,us,her	She liked him.
(ROLE SUBJ/OBJ)	may be either subject or object	it,who,you	Who liked it?
(NUMBER SG/PL)	may be singular or plural	deer,you	The deer played.
(PNCODE X13SG)	goes with all except 1st and 3rd person singular	are,were	You are sleepy.
(PNCODE 3SG)	goes with 3rd person singular only	does,goes,is,spends	The child knows a game.

TABLE III: Sample Syntactic Features

A very detailed presentation of the format of the LUNAR dictionary is given in [Woods et al, 1972]. One if its most interesting characteristics was the inclusion of morphological codes with the root forms of regularly inflected words. (Irregular words appeared as separate entries, the root forms of which has a morphological code indicating the irregularity.) Using this code, a function (named MORPH, called from LEXIC) took a word of input and attempted to strip off regular endings in such a way that the remaining root was a word in the dictionary whose morphological code agreed with the endings which had been removed. This mechanism made for considerably more compact dictionaries than if all inflected forms had to be stored.

Unfortunately, for languages with very complex morphological structures this method may not be adequate. It has been suggested [Grimes, 1975] that for such languages the input words be hyphenated and that separate levels of the grammar be written to do the detailed morphological decompositions.

Besides the parts of speech and features, the dictionary may also contain information about substitutions which should be made in the input string or compounds which should be collapsed. For example, the following may be considered equivalent:

United States
United States of America
USA
United-States

If the last form is chosen to be the "standard" one, the first two can be collapsed to it and the third can be expanded to it by either a prepass on the input or by the function LEXIC which gets the next word of input from the string.