AN IMPLEMENTATION OF

THE AUGMENTED TRANSITION NETWORK SYSTEM

OF WOODS

by
David Matuszek
June, 1971

as
revised by

Oct.,1972 Jonathan Slocum NL9

NATURAL LANGUAGE RESEARCH FOR COMPUTER-ASSISTED INSTRUCTION

Supported by:

THE NATIONAL SCIENCE FOUNDATION
Grant GJ 509 X

Department of Computer Sciences

and

Computer-Assisted Instruction Laboratory

The University of Texas
Austin, Texas

INTRODUCTION

English is a linear language. Every meaningful utterance in English consists of a sequence of symbols, called words, and even when these words are written on a two-dimensional sheet of paper they maintain their linear ordering. This apparent simplicity of form, however, does not reflect a corresponding simplicity in the meaning of the utterance, in the thought or thoughts which the words are intended to convey; rather, it seems to be an artifact resulting from the nature of verbal communication.

Of course, vocal sounds themselves may vary along a number of dimensions, but the possible variation is much too limited to be adequate for expressing complex ideas with a single sound. Rather, this variation is used to construct a basic set of speech sounds, called phonemes, which are meaningless when considered individually but which may be arranged in particular sequences to form words, which may in turn be combined sequentially to form sentences. Thus, while thoughts and ideas may not be constrained to a linear format, spoken languages are necessarily so constrained, and it is quite possible that many of the complexities of grammar are the result of linguistic signals which must be present to reflect the underlying structure of the thought conveyed. In other words, a sentence may contain a large number of cues whose function is to indicate to the hearer how to interrelate the more substantive aspects of the sentence in order to replicate the structure which exists in the mind of the speaker.

The nature of this underlying structure is the primary unknown in the field of linguistics. Most linguists, however, feel that a tree

structure is sufficiently general to represent the meanings of utterances; accordingly, a great deal of work has beer done in an attempt to devise a suitable means of mapping sentences into tree structures and back again. The development of suitable types of tree structures and of algorithms for performing the mapping is a difficult job, and one in which the aid of a computer can be of immense assistance.

Woods (1969) has recently described a LISP-based computer language for representing algorithms which parse sentences into tree structures. The language is more or less a description language for a particular type of graph structure, which Woods refers to as an augmented transition network. Woods shows that such graphs can be powerful tools for implementing various parsing strategies.

The primary purpose of this paper is to describe in detail a particular implementation of the sentence parsing system of Woods. While Woods gives a very clear overall view of the system, many of the details relevant to programming and/or using the system are ambiguous, to say the least. Accordingly, this paper may be taken as supplementary to that of Woods.

The attempt has been made to remain consistent with the system as described by Woods. Ambiguities have for the most part been resolved in favor of the interpretation deemed most natural or most useful, rather than the interpretation easiest to code. Some generalizations have been made in order to increase the power of the system, while two routines (BUILDQ and BUILD) have been discarded as they encourage what we feel is a too-restrictive approach to parse-construction -- that is, the full powers of LISP (or GROPE, as the case may be) permit a complete latitude of choice with regards to constructing a parse of a sentence. Therefore

the maintenance of such restricting functions was deemed inappropriate. Finally, an added capability allows the user to construct as many such "grammars" as he desires -- so long as each is uniquely named.

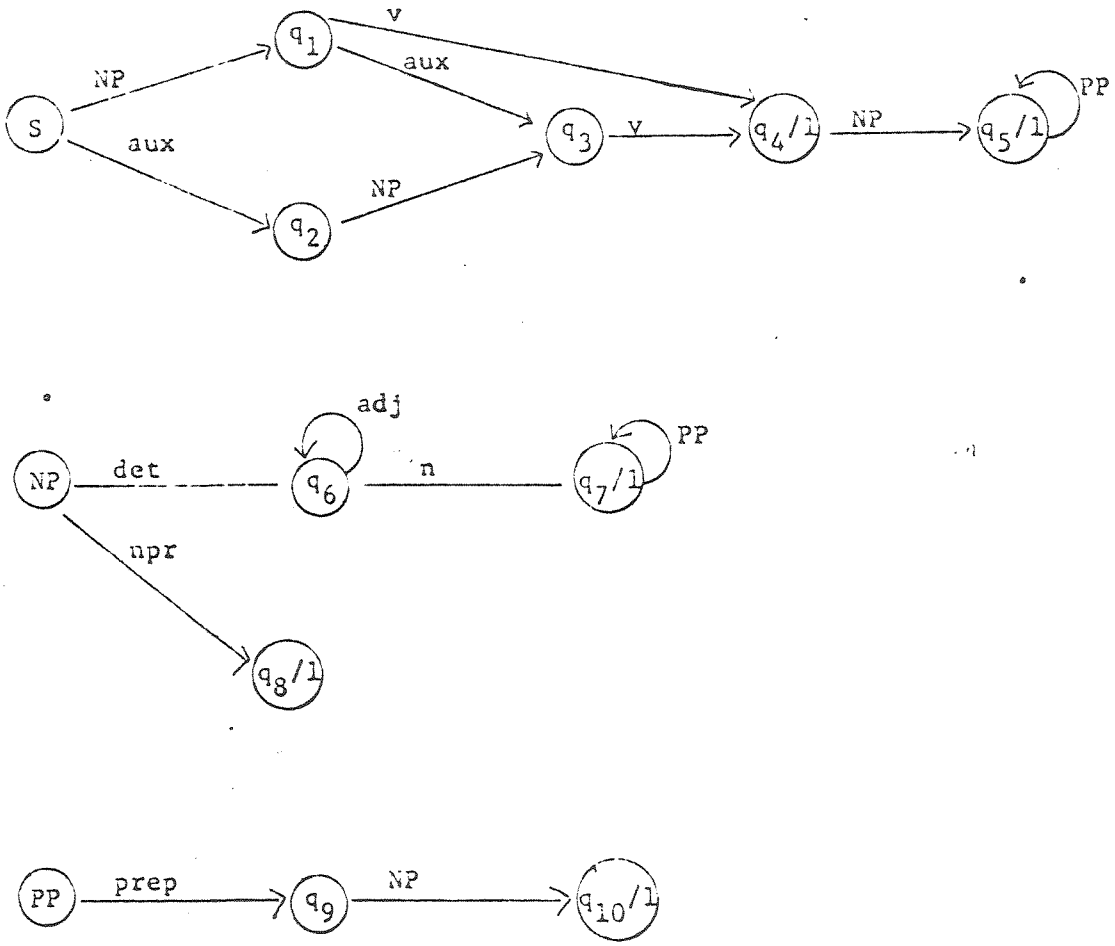
TRANSITION NETWORKS

A recursive transition network (see Fig. 1) is a directed graph with labelled nodes, called states, and labelled arcs. In each network, one particular node is designated the start state and is labelled with the name of the syntactic type which the network is to recognize, while the remaining nodes have arbitrary labels. In addition, one or more nodes (marked with a "/1" in Fig. 1) are called final states: when entered, they indicate that an instance of the syntactic type indicated by the label of the start state has been found.

Every node that is not a final state has one or more labelled arcs pointing from it, while final states may or may not have such arcs pointing from them. Each arc in the graph is labelled in one of two ways; either with the name of a category into which individual words of the input may fall, or with the name of a syntactic type which is also the label of the start state of another portion of the network.

A recursive transition network may be thought of as a mechanism which accepts as input a string of symbols in a source language (e.g. words in an English sentence), does computation over the input, and produces as output either a "yes" or a "no" response, indicating that the input string is or is not an instance of the specified syntactic type, respectively.

The computation performed by the network proceeds as follows. The network is initially in the start state; that is, the system maintains a marker to the node currently under consideration, and this marker initially points to the start state. Under control of the input, the marker advances from node to node along the arcs, checking at each node whether or not it is in the set of final states. If the marker points to a final state at the time the input string is exhausted, then the system accepts the string,



S is the start state

q_4, q_5, q_7, q_8, and q_{10} are the final states

Figure 1: A sample transition network

(Reproduced from Woods, 1969)

that is, it responds "yes". On the other hand, if the end of the input stream is reached while the marker is not pointing to a final state, or if at any point in the procedure a suitable outpointing arc cannot be found, then the input is not accepted as being an instance of the specified syntactic type.

An arc may be followed if one of the following two conditions holds: 1) the arc is labelled with the syntactic type of the current symbol in the input stream, or 2) the arc is labelled with the name of the start state of another portion of the network, and this other portion of the network accepts as input a substring of the input beginning with the current symbol.

The augmented transition network differs from the recursive transition network outlined above in two respects: 1) arbitrary tests may be put on the arcs, which must be satisfied before the arc can be followed, and 2) following an arc may cause a set of structure-building operations to be performed. Thus, while a recursive transition network is able to recognize syntactic types, an augmented transition network may in addition create a parse structure of the syntactic type which was input, as a side effect of tracing through the network.

THE SPECIFICATION LANGUAGE: PART I.

In order to work with augmented transition networks on the computer, Woods developed a LISP-based language for their specification. In this language, very powerful tests (in fact, arbitrary LISP expressions) may be placed on the arcs, although only a rather limited number of structure-building operations are available. A partial description of the syntax of the language as described by Woods is shown in Fig. 2 in a BNF-like metalanguage; a similar but somewhat more complete description of the extended language as described in this paper is shown in Fig. 3.

A program in this language is intended to be a complete description of an augmented transition network. Accordingly, the program itself is also referred to as a transition network.

Each program consists of a list of arc sets; the order in which the arc sets appear in the list is not important. Intuitively, each arc set consists of the name of a node and a description of all the arcs pointing out from that node; thus, there are exactly as many arc sets in the program as there are nodes in the network. The arc set is written as a list whose first element is the label of the node represented (e.g., "PP"), and the remaining elements of the list each describe an arc emanating from that node.

We have already noted that an arc of a recursive transition network may be labelled in one of two different ways. This is reflected in the specification language by the existence of two distinct types of arcs, the CAT arc and the PUSH arc. Each of these will be described in turn.

The CAT arc is used for those arcs of the network which are labelled with the name of a category into which the individual words of the input may fall. For example, the arcs labelled "v" in Fig. 1 are those which

```
<transition network> → (<arc set> <arc set>*)

<arc set> → (<state> <arc>*)

<arc> → (CAT <category name> <test> <action>* <term act>)  |
        (PUSH <state> <test> <action>* <term act>)  |
        (TST <arbitrary label> <test> <action>* <term act>)  |
        (POP <form> <test>)

<action> → (SETR <register> <form>)  |
           (SENDR <register> <form>)  |
           (LIFTR <register> <form>)

<term act> → (TO <state>)  |
             (JUMP <state>)

<form> → (GETR <register>)  |
         *  |
         (GETF  <feature>)  |
         (BUILDQ <fragment> <register>*)  |
         (LIST <form>*)  |
         (APPEND <form> <form>)  |
         (QUOTE <arbitrary structure>)
```

Figure 2: Specification of a language for representing augmented transition networks.

(Reproduced from Woods, 1969)

```
<transition network> → ((<arc set> <arc set>*) <network name>)

<arc set> → (<state> <arc>*)

<arc> → (CAT <category name> <test> <action>* <term act>) |
        (PUSH <state> <test> <action>* <term act>) |
        (EVPUSH <form> <test> <action>* <term act>) |
        (VIRT <syntactic type> <test> <action>* <term act>) |
        (TST <arbitrary label> <test> <action>* <term act>) |
        (POP <form> <test>)

<test> → <form>

<action> → <form>

<term act> → (TO <state>) |
             (TO* <state>) |
             (HOP <state>) |
             (JUMP <state>)

<form> → (GETR <register>) |
         (SETR <register> <form>) |
         (SENDR <register><form>) |
         (LIFTR <register> <form>) |
         (HOLD <form>) |
         (GETF  <feature>) |
         (GETM <feature>) |
         * |
         <any simple LISP-type S-expression which can be
          evaluated, not including COND, PROG, etc.>

<network name>, <state>, <category name>, <syntactic type>,  <register>,
<arbitrary label>,  <feature> → <any LISP-type atom>
```

Figure 3.

may be followed if the current input symbol is a verb. CAT arcs are written in the specification language as a list having the following elements:

1) The word "CAT", to indicate the type of arc.

2) The name of the syntactic category to which the input work must belong if the arc is to be followed.

3) A <test> which is also used to determine whether or not the arc may be followed: this <test> is evaluated after (and in conjunction with) the category check.

4) A set of <action> (structure-building operations) to be performed in the event that the arc is followed.

5) A <term act> which specifies the node to which the arc points, and is an indicator as to whether or not the input scanner should be advanced, and/or the * register affected.

The PUSH and EVPUSH arcs are used to represent those arcs of the network which are labelled with non-terminal syntactic type names which are also the names of other nodes. For example, an arc may be labelled "PP" for prepositional phrase, provided that "PP" is also the label of another node elsewhere in the network. Such an arc is not immediately followed: instead, control transfers to the node named - provided the <test> first evaluates true - and the subnetwork beginning with this node is followed in an attempt to recognize and parse a constituent of the type named. In this example, the arc having label "PP" would cause control to transfer to the node "PP"; if a prepositional phrase can be found, control would then be returned and the arc labelled "PP" would be followed, otherwize the arc labelled "PP" would fail and the next arc would be attempted. The EVPUSH arc is different only in that the <form> is evaluated to produce the name of the state to which control will be transferred — again, provided the <test> returns true.

The specification of the PUSH arc is very nearly identical to that of the CAT arc: only the first two elements of the list differ. The word "PUSH" (or "EVPUSH") is used in place of the word "CAT" as the first element,

while the second element is the name (or in EVPUSH, will evaluate to the name) of the required syntactic type, rather than the name of the category. Execution, however, is quite different. If the <test> evaluates true, and before transfer to the "named" node occurs, all registers in use (of which more will be said later), except for the special register "*", are pushed down to save their contents (hence the name PUSH), and when control is again returned the registers still contain the information they held before the PUSH occurred. The registers involved in this operation will be discussed in some detail in connection with structure-building operations.

A third type of arc, the VIRT arc, is available in the current implementation. While not specifically described by Woods, the existence of the VIRT arc or its equivalent in Woods' system may be inferred from his discussion of "virtual arcs" in connection with his second example. The specification of the VIRT arc is again very nearly the same as that of the CAT arc, differing only in the first two elements of the list. The word "VIRT" replaces the word "CAT" as the first element of the list; the second element is the name of a syntactic type, rather than the name of a category. Unlike the PUSH arc, the VIRT arc does not initiate an attempt to parse a constituent of the required type. Instead, the VIRT arc checks whether such a constituent has previously been found at the current level of processing and placed on a special list, called the HOLD list. If so, the constituent is removed from the HOLD list and transferred to the special system register "*". Then if the <test> on the arc evaluates true, the actions on the arc are performed. The VIRT arc tests only the HOLD list for this constituent; it does not examine the current input symbol pointed to by the scanner. (The HOLD list will be discussed in greater detail in a later section.)

A fourth type of arc described by Woods is the TST arc. The format of the TST arc is similar to that of the other arcs described thus far. The first element of the list is the word "TST", while the second element is an arbitrary label, serving only to maintain conformity with the other types of arcs, and otherwise completely ignored. (However, the current implementation requires that the label be present.) The TST arc may be followed if and only if the <test> on the arc (the third element of the list) is successful.

The final type of arc to be described, the POP arc, is to some extent the converse of the PUSH arc. The POP arc is used to indicate that the node from which it emanates is a final state, and it provides for arbitrary conditions which must be satisfied in order to allow a pop to a higher-level network. The POP arc, when followed, "pops" those registers which were pushed down as a result of following a previous PUSH arc. However, this is not a true pop in the usual sense of the word; the contents of the registers at the lower level are not lost, and will again become available should another PUSH arc be attempted subsequently. As with the PUSH operation, the system register "*" is a non-recursive register, so it is not affected by popping. Instead, it is used to communicate information from the current computation to higher level networks. A <form> or computation which returns a value, is a mandatory part of the specification of a POP arc; when the arc is followed, the computation is performed and the value which results is automatically placed in the special register "*", provided that value is not "false".

A POP arc is written in the specification language as a list composed of the following three elements:

1) The word "POP".

2) The <form> to be evaluated and placed in the "*" register, provided the <test> returns true.

3) A <test> to determine whether or not the arc may by followed.

There is no <term act> in the specification because, unlike a true arc, the POP arc does not point to anything. Control returns to the PUSH (or EVPUSH) arc whose evaluation was most recently initiated but not yet completed.

This completes the description of the types of arcs available in the specification languages.

It is possible for a node to be encountered which has no applicable arcs emanating from it. In this event, the action taken is the same as that described for the POP arc. The false value is returned; control returns to the PUSH arc most recently initiated and not yet completed, and the false value prevents this arc from being followed. Note that this does not necessarily, or even usually, cause the entire parse to fail. It should also be noted that the above feature allows explicit "failure POP arcs" to be written, merely by using NIL as the <form> and T as the <test>on the POP arc.

The next section describes in some detail the special system operations and the registers over which they are defined.

THE SPECIFICATION LANGUAGE: PART II.

The Woods system uses a number of registers for temporary storage of information. Woods, unfortunately, does not go into detail regarding the nature, use, or mode of operation of these registers. The description which follows is therefore a description of the current implementation only; while no known incompatibilities with Woods' system exist, it cannot be guaranteed that the two systems are in fact compatible.

As can be seen from Fig. 4, a program written in Woods' language uses a number of registers whose names are common English words or abbreviations, such as SUBJ, TYPE, and AUX. These registers, which we will call programmer registers, are all recursive, each capable of holding a stack of arbitrary S-expressions. The execution of a PUSH, EVPUSH, or POP arc (or automatic popping due to failure to find an applicable arc) results in the pushing or popping of all programmer registers. Thus, if we consider the PUSH arcs as initiating a computation at a deeper level of recursion, and popping as returning to the next higher level of recursion, then we may consider each level of a computation as having its own private set of registers, each capable of holding a single S-expression.

When the registers are considered to exist independently at every level of recursion, it can be seen that popping from a level need not destroy the contents of the registers at that level. Thus, registers are not actually pushed down or popped, but rather each PUSH arc or POP arc merely changes the set of registers which are available. As will be seen, this rather unusual interpretation of the form of the registers is necessitated by the SENDR action, which transmits information to the next lower level of recursion.

It appears from Woods' paper that there is a (relatively) fixed set

```
(S/ (PUSH NP/ T
        (SETR SUBJ *)
        (SETR TYPE (QUOTE DCL))
        (TO Q1))
    (CAT AUX T
        (SETR AUX *)
        (SETR TYPE (QUOTE Q))
        (TO Q2)) )
(Q1 (CAT V T
        (SETR AUX NIL)
        (SETR V *)
        (TO Q4))
    (CAT AUX T
        (SETR AUX *)
        (TO Q3)) )
(Q2 (PUSH NP/ T
        (SETR SUBJ *)
        (TO Q3)) )
(Q3 (CAT V T
        (SETR V *)
        (TO Q4)) )
(Q4 (POP (BUILDQ (S + + + (VP +)) TYPE SUBJ AUX V) T)
    (PUSH NP/ T
        (SETR VP (BUILDQ (VP (V +) *) V))
        (TO Q5)) )
(Q5 (POP (BUILDQ (S + + + +) TYPE SUBJ AUX VP) T)
    (PUSH PP/ T
        (SETR VP (APPEND (GETR VP) (LIST *)))
        (TO Q5)) )
```

Figure 4: An illustrative fragment of an augmented transition network. (Reproduced from Woods, 1969)

of programmer registers available, although a list of register names was not given in the paper. Instead of attempting to compile such a list, the current implementation makes use of the generalized high-speed storage features of GRASPE 1.5, a LISP extension - or GROPE, a FORTRAN extension - to maintain the registers. With this elaboration, any legal atom may be used, without prior definition or binding, as the name of a programmer register. These registers are defined by their occurrance in an AFSTN program; initially, each such register is empty at every level of recursion. As has been noted, no automatic clearing of registers occurs during a POP operation, so that at any level of recursion the registers maintain their contents until explicitly altered by the program.

In addition to the programmer registers, there is a special system register, * , whose function it is to hold the current constituent of the sentence under consideration. This system register is apparently intended to be nonrecursive, capable of holding only a single S-expression, and has been so implemented in the current program. Because it is not recursive, it may not be interrogated or altered by the same instructions which interrogate and alter programmer registers. (In this connection, it should be noted that the defined type <register> in the syntax tables of Figures 2 and 3 refer only to programmer registers.) System register operations will be discussed at the end of this section.

A <form> is an expression which results in a value. Certain of the forms retrieve the contents of a register; others may perform arbitrary computation over the contents of zero or more registers. The syntax of forms is given in Figure 3; the following discussion concentrates on their semantics.

The GETR instruction fetches the contents of the designated register at the current level of computation. GETR may be used on any programmer register. However, since the system register * is not recursive, the instruction "(GETR *)" is not meaningful; instead, the "*" standing alone is used as a complete form which fetches the contents of the system register.

The GETM instruction takes as its single parameter the name of a "morphological feature" and returns true if the last item processed by a CAT arc was that morphological variant of its root form. (Along with the CAT arc operation, this is dependent upon the logical structure of the lexicon -- an appropriate lexical input routine is provided.

The GETF form acts as a lexical retrieval operation; it takes as its single parameter the name of a "feature", and returns as its value the value of the feature -- retrieved with respect to the last input symbol processed by a CAT arc. This too, then, is dependent upon the lexical structure.

The instruction SETR takes as its two arguments a programmer register name and a <form>, evaluates the <form> , and stores the result in the designated register at the current level of recursion. Thus, SETR is normally used to save information which may later be retrieved by GETR. The value of the function SETR is the evaluated result.

Two additional forms, SENDR and LIFTR, also take as their two arguments a programmer register and a <form>, and evaluate the <form>. However, SENDR stores the result of this evaluation in the designated register at the next lower level of recursion, while LIFTR stores it at the next higher level of recursion. These also return that result.

The HOLD list, like the programmer registers, has an independent existence at every level of recursion. Unlike the registers, which may each hold only a single S-expression, the HOLD list is capable of maintaining an entire list of values. One final type of action, the HOLD action, is used

for inserting new items onto the HOLD list. HOLD takes as its one argument a <form> to be evaluated. An important restriction is that the evaluated <form> must be a list whose first element is an atom representing the name of a syntactic type. After evaluation, the list is placed on the HOLD list at the current level of recursion. Values may be retrieved from the HOLD list only by execution of a VIRT arc, as described earlier.

Normally, an atom in a <form> is evaluated before being used as a parameter to a function -- that is, its value is passed. The programmer may bind a value to an atom by executing the function SETQ <atom> <form>. This also applies to the system atom "*" -- that is, the star register "*" is just a special atom, whose value may be automatically affected when the scanner is moved during the course of an application of the program to an input sentence.

Finally, any LISP-type S-expression (function call, etc.) may now be used as a form. This is perhaps the most important extension of the language as described by Woods, which only allowed APPEND and LIST, but no other system or user-supplied functions. With the current implementation, the user may use any LISP (or GROPE) function which evaluates its arguments, including functions which he has written himself. Moreover, within a form, it is possible to use other forms, at any level of parenthesis nesting, exactly as if they were LISP functions. User-supplied functions should be written in pure LISP - or GROPE, as the case may be.

THE SPECIFICATION LANGUAGE: PART III.

During the attempted parse of an input string, a special device called the scanner is used to point to that symbol in the input string currently being processed. As in Woods' system, no back up is allowed -- the scanner can only advance or remain in place. In the current implementation, any attempt to advance past the end of the input string results in the scanner pointing to the false value; this is normal procedure and does not constitute an error. The means of moving the scanner is by the execution of a <term act> , which is the last element of every arc except POP arc. There are four different <term acts> available: TO, TO*, JUMP, and HOP, each of which takes as its single argument the name of the node (state) of the network to which the arc points.

The JUMP <term act> does not advance the scanner. The symbol currently under the scanner is copied into the * register, and control is transferred to the named node. HOP only transfers control to the named node, without moving the scanner or affecting the * register.

Normally, the TO <term act> both advances the scanner and enters the new symbol scanned into the * register, thus making it available to the program, then transfers control to the named node. This clearly coincides with Woods' intentions, but in our implementation this is instead the definition of TO*, since a more commonly useful definition of TO was found: with the former definition of the action of TO, unfortunately, it is impossible to make Woods' first example network parse in the manner specified. Accordingly, the following peculiarity has been added to the definition of the TO <term act> : when the * register does not contain the symbol currently pointed to by the scanner (as for example when a PUSH has just been successful), the scanner is not advanced, and the symbol currently pointed

to by the scanner is entered into the * register. (Again, TO* conforms to the original Woods definition of TO.)

It is now possible to describe the operation of the scanner and the * register in more detail. At the beginning of a "parse", the scanner is set to point to the first symbol of the input string, and this symbol is also entered into the * register. A PUSH to node S/ on level 0 is then executed, thus turning control over to the program in the specification language.

The scanner is interrogated whenever a TO <term act> is executed. The * register is interrogated and/or altered at the following times:

(1) Whenever a JUMP, TO*, or TO < term act> is executed, the scanner either is advanced or remains in place, as described above, following which the symbol then under the scanner is entered into the * register.

(2) Whenever a POP arc is successfully executed, the value returned by the < form > (the second element in the description of the POP arc) is entered into the * register, provided that value is not "false" -- NIL in LISP, zero in GROPE.

(3) Whenever it is determined that a VIRT arc might be followed, but before the test or actions on the arc are executed, the expression removed from the HOLD list is entered into the * register.

(4) Whenever a CAT arc is attempted, a test is made to determine whether the symbol in the * register belongs to the specified category.

(5) The "*" standing alone may be used as a complete form whose value is the current contents of the * register.

USING THE SENTENCE PARSING SYSTEM

In order to use the sentence parsing system, it is necessary to be able not only to define a transition network, but also to input to the system any and all information required by the network during a parse. A number of functions which have been made available to the user for these purposes are described below.

LISP version

DEFINE* <transition network> -- compiles the network into a (CRASPE-type) graph, whose name is the <network name>. This function may be used to amend previously-compiled networks, a node at a time.

TRACEPARSE (flag) -- will enable or suppress the trace feature according to whether flag is true (not NIL) or false (NIL).

CLEARREGS ( ) -- empties all registers at all levels.

LEXICON <lexicon> -- compiles the <lexicon>.

GRAMMAR (<network name>) -- identifies to the AFSTN interpreter which (grammar) network is to be interpreted in subsequent execution of an AFSTN network.

Note: the syntax of the <lexicon> appears in Fig. 5.

< lexicon> → (< lexical entry>*)

< lexical entry> → (< word> (< category> < morphological feature>)

< rest of lexical information>*)

< rest of lexical information> → (< morphological feature> < word>) |

(< feature> < feature value>)

< word> → < any atom of the source (English) language>

examples: PRODUCE, BOY, IT, RED.

< category> → < any atomic form-class name>

examples: N, V, ADJ, ADV, PREP, CONJ, PRON.

< morphological feature> → < any atomic morphological-class name>

examples: S, PL, INF, PST, :EN, :ING, SG3, POS, COM, SPR, PI, NOM,

OBJ, POSS, RFL.

< feature> → < any atom not a < morphological feature> name>

< feature value> → < any LISP-type S-expression>

Figure 5

The syntax of the input lexicon

GROPE version

GRAMMIN <transition network> -- compiles the network into a <graph>, whose label is the <network name>, and returns the <graph> as the value of GRAMMIN. This function may be used to amend previously-compiled networks, a node at a time.

TRACEP flag -- will enable or suppress the trace feature according to whether flag is true (non-zero) or false (zero).

CLEAREG -- empties all registers at all levels.

LEXICON <lexicon> -- compiles the <lexicon>.

START <node> <list> -- begins the interpretation (execution) of a network at location <node>, using <list> as the control string (input sentence).