GENERALIZED AUGMENTED TRANSITION NETWORK GRAMMARS FOR GENERATION FROM SEMANTIC NETWORKS

Stuart C. Shapiro
Department of Computer Science, SUNY at Buffalo

1. INTRODUCTION

Augmented transition network (ATN) grammars have, since their development by Woods [7;8], become the most used method of describing grammars for natural language understanding and question answering systems. The advantages of the ATN notation have been 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" [1,p.191]. The usual method of utilizing an ATN grammar in a natural language system is to provide an interpreter which can take any ATN grammar, a lexicon, and a sentence as data and produce either a parse of a sentence or a message that the sentence does not conform to the grammar. A compiler has been written [2;3] which takes an ATN grammar as input and produces a specialized parser for that grammar, but in this paper we will presume that an interpreter is being used.

A particular ATN grammar may be viewed as a program written in the ATN language. The program takes a sentence, a linear sequence of symbols, as input, and produces as output a parse which is usually a parse tree (often represented by a LISP S-expression) or some "knowledge representation" such as a semantic network. The operation of the program depends on the interpreter being used and the particular program (grammar), as well as on the input (sentence) being processed.

Several methods have been described for using ATN grammars for sentence generation. One method [1,p.235] is to replace the usual interpreter by a generation interpreter which can take an ATN grammar written for parsing and use it to produce random sentences conforming to the grammar. This is useful for testing and debugging the grammar. Another method [5] uses a modified interpreter to generate sentences from a semantic network. In this method, an ATN register is initialized to hold a node of the semantic network and the input to the grammar is a linear string of symbols providing a pattern of the sentence to be generated. Another method [4] also generates sentences from a semantic network. In this method, input to the grammar is the semantic network itself. That is, instead of successive words of a surface sentence or successive symbols of a linear sentence pattern being scanned as the ATN grammar is traversed by the interpreter, different nodes of the semantic network are scanned. The grammar controls the syntax of the generated sentence based on the structural properties of the semantic network and the information contained therein.

It was intended that a single ATN interpreter could be used both for standard ATN parsing and for generation based on this last method. However, a special interpreter was written for generation grammars of the type described in [4], and, indeed, the definition of the ATN formalism given in that paper, though based on the standard ATN formalism, was inconsistent enough with the standard notation that a single interpreter could not be used. This paper reports the results of work carried out to remove those inconsistencies. A generalization of the ATN formalism has been derived which allows a single interpreter to be used for both parsing and generating grammars. In fact, parsing and generating grammars can be sub-networks of each other. For example an ATN grammar can be constructed so that the "parse" of a natural language question is the natural language statement which answers it, interaction with representation and inference routines being done on arcs along the way. The new formalism is a strict generalization in the sense that it interprets all old ATN grammars as having the same semantics (carrying out the same actions and producing the same parses) as before.

This material is based on work supported in part by the National Science Foundation under Grant #MCS78-02274.

2. GENERATION FROM A SEMANTIC NETWORK--BRIEF OVERVIEW

In our view, each node of a semantic network represents a concept. The goal of the generator is, given a node, to express the concept represented by that node in a natural language surface string. The syntactic category of the surface string is determined by the grammar, which can include tests of the structure of the semantic network connected to the node. In order to express the concept, it is often necessary to include in the string substrings which express the concepts represented by adjacent nodes. For example, if a node represents a fact to be expressed as a statement, part of the statement may be a noun phrase expressing the concept represented by the node connected to the original node by an AGENT case arc. This can be done by a recursive call to a section of the grammar in charge of building noun phrases. This section will be passed the adjacent node. When it finishes, the original statement section of the grammar will continue adding additional substrings to the growing statement.

In ATN grammars written for parsing, a recursive push does not change the input symbol being examined, but when the original level continues, parsing continues at a different symbol. In the generation approach we use, a recursive push often involves a change in the semantic node being examined, and the original level continues with the original node. This difference is a major motivation of some of the generalizations to the ATN formalism discussed below. The other major motivation is that, in parsing a string of symbols, the "next" symbol is well defined, but in "parsing" a network, "next" must be explicitly specified.

3. THE GENERALIZATION

The following sub-sections show the generalized syntax of the ATN formalism, and assume a knowledge of the standard formalism ([1] is an excellent introduction). Syntactic structures already familiar to ATN users, but not discussed here remain unchanged. Parentheses and terms in upper case letters are terminal symbols. Lower case terms in angle brackets are non-terminals. Terms enclosed in square brackets are optional. Terms followed by "*" may occur zero or more times in succession. To avoid confusion, in the remainder of this section we will underline the name of the <u>*</u> register.

3.1 TERMINAL ACTIONS

Successful traversal of an ATN arc might or might not consume an input symbol. When parsing, such consumption normally occurs, when generating it normally does not, but if it does, the next symbol (semantic node) must be specified. To allow for these choices, we have returned to the technique of [6] of having two terminal actions, TO and JUMP, and have added an optional second argument to TO. The syntax is:

```
(TO <state> [<form>])
(JUMP <state>)
```

Both cause the parser to enter the given state . JUMP never consumes the input symbol; TO always does. If the <form> is absent in the TO action, the next symbol to be scanned will be the next one in the input buffer. If <form> is present, its value will be the next symbol to be scanned. All traditional ATN arcs except JUMP and POP end with a terminal action.

The explanation given for the replacement of the JUMP terminal action by the JUMP arc was that, "since POP, PUSH and VIR arcs never advance the input, to decide whether or not an arc advanced the input required knowledge of both the arc type and termination action. The introduction of the JUMP arc ... means that the input advancement is a function of the arc type alone." [2] That our reintroduction of the JUMP terminal action does not bring back the confusion is explained below in Section 4.

3.2 ARCS

We retain a JUMP arc as well as a JUMP terminal action. The JUMP arc provides a place to make an arbitrary test and perform some actions without consuming an input symbol. We need such an arc that does consume its input symbol, but TST is not adequate since it, like CAT, is really a bundle of arcs, one for each lexical entry of the scanned symbol, should the latter be lexically ambiguous. A semantic node, however, does not have a lexical entry. We therefore introduce a TO arc:

(TO (<state> [<form>]) <test> <action>*)

If <test> is successful, the <action>s are performed and transfer is made to <state>. The input symbol is consumed. The next symbol to be scanned is the value of <form> if it is present or the next symbol in the input buffer if <form> is missing.

The PUSH arc makes two assumptions: 1) the first symbol to be scanned in the subnetwork is the current contents of the * register; 2) the current input symbol will be consumed by the subnetwork, so the contents of * can be replaced by the value returned by the subnetwork. We need an arc that causes a recursive call to a subnetwork, but makes neither of these two assumptions, so we introduce the CALL arc:

(CALL <state> <form> <test> <preaction or action>* <register> <action>* <terminal action>)

where <preaction or action> is <preaction> or <action>. If the <test> is successful, all the <action>s of <preaction or action> are performed and a recursive push is made to the state <state> where the next symbol to be scanned is the value of <form> and registers are initialized by the <preaction>s. If the subnetwork succeeds, its value is placed into <register> and the <action>s and <terminal action> are performed.

Just as the normal TO terminal action is the generalized TO terminal action with a default form, the PUSH arc (which we retain) is the CALL arc with the following defaults: <form> is *; the <preaction or action>s are only <preaction>s; <register> is *.

3.3 FORMS

The only form which must be added is

(GETA <arc> [<node form>])

where <node form> is a form which evaluates to a semantic node. If absent, <node form> defaults to *. The value of GETA is the node at the end of the arc labelled <arc> from the specified node, or a list of such nodes if there are more than one.

3.4 TESTS, PREACTION, ETC.

The generalization of the ATN formalism to one which allows for writing grammars which generate surface strings from semantic networks, yet can be interpreted by the same interpreter which handles parsing grammars, requires no changes other than the ones described above. Of course, each implementation of an ATN interpreter contains slight differences in the set of tests and actions implemented beyond the basic ones.

4. THE INPUT BUFFER

Input to the ATN parser can be thought of as being the contents of a stack, called the input buffer. If the input is a string of words, the first word will be at the top of the input buffer and successive words will be in successively deeper positions of the input buffer. If the input is a graph, the input buffer might contain only a single node of the graph.

On entering an arc, the * register is set to the top element of the input buffer, which must not be empty. The only exceptions to this are the VIR and POP arcs. VIR sets * to an element of the HOLD register. POP leaves * undefined since * is always the element to be accounted for by the current arc, and a POP arc is not trying to account for any element. The input buffer is not changed between the time a PUSH arc is entered and the time an arc emanating from the state pushed to is entered, so the contents of * on the latter arc will be the same as on the former. A CALL arc is allowed to specify the contents of * on the arcs of the called state. This is accomplished by replacing the top element of the input buffer by that value before transfer to the called state. If the value is a list of elements, we push each element individually onto the input buffer. This makes it particularly easy to loop through a set of nodes, each of which will contribute the same syntactic form to the growing sentence (such as a string of adjectives).

While on an arc (except for POP), i.e. during evaluation of the test and the acts, the contents of * and the top element of the input buffer are the same. This requires special processing for VIR, PUSH, and CALL arcs. After setting *, a VIR arc pushes the contents of * onto the input buffer. When a PUSH arc resumes, and the lower level has successfully returned a value, the value is placed into * and also pushed onto the input buffer. When a CALL resumes, and the lower level has successfully returned a value, the value is placed into the specified register, and the contents of * is pushed onto the input buffer. The specified register might or might not be *. In either case the contents of * and the top of the input buffer are the same.

There are two possible terminal acts, JUMP and TO. JUMP does not affect the input buffer, so the contents of * will be same on the successor arcs (except for POP and VIR) as at the end of the current arc. TO pops the input buffer, but if provided with an optional form, also pushes the value of that form onto the input buffer.

POPping from the top level is only legal if the input buffer is empty. POPping from any level should mean that a constituent has been accounted for. Accounting for a constituent should entail removing it from the input buffer. From this we conclude that every path within a level from an initial state to a POP arc must contain at least one TO transfer, and in most cases, it is proper to transfer TO rather than to JUMP to a state that has a POP arc emanating from it. TO will be the terminal act for most VIR and PUSH arcs.

In any ATN interpreter which abides by this discussion, advancement of the input is a function of the terminal action alone in the sense that at any state JUMPed to, the top of the input buffer will be the last value of *, and at any state jumped TO it will not be.

5. THE LEXICON

Parsing and generating require a lexicon -- a file of words giving syntactic categories, features and inflectional forms for irregularly inflected words. Parsing and generating require different information, yet we wish to avoid duplication as much as possible.

During parsing, morphological analysis is performed. The analyzer is given an inflected form, must segment it, find the stem in the lexicon and modify the lexical entry of the stem according to its analysis of the original form. Irregularly inflected forms must have their own entries in the lexicon. An entry in the lexicon may be lexically ambiguous, so each entry must be associated with a list of one or more lexical feature lists. Each such list, whether stored in the lexicon or constructed by the morphological analyzer, must include a syntactic category and a stem, which serves as a link to the semantic network, as well as other features such as transitivity for a verb.

In the semantic network, some nodes are associated with lexical entries. During generation, these entries, along with other information from the semantic network, are used by a morphological synthesizer to construct an inflected word. We assume that all such entries are unambiguous stems, and so contain only a single lexical feature list. This feature list must contain any irregularly inflected forms.

In summary, a single lexicon may be used for both parsing and generating under the following conditions. An unambiguous stem can be used for both parsing and generating if its one lexical feature list contains features required for both operations. An ambiguous lexical entry will only be used during parsing. Each of its lexical feature lists must contain a unique but arbitrary "stem" for connection to the semantic network and for holding the lexical information required for generation. Every lexical feature list used for generating must contain the proper natural language spelling of its stem as well as any irregularly inflected forms. Lexical entries for irregularly inflected forms will only be used during parsing.

For the purposes of this paper, it should be irrelevant whether the "stems" connected to the semantic network are actual surface words like "give", deeper sememes such as that underlying both "give" and "take", or primitives such as "ATRANS".

6. EXAMPLE

Figure 1 shows an example interaction using the SNePS Semantic Network Processing System [5] in which I/O is controlled by a parsing-generating ATN grammar. Lines begun by "**" are user's input, which are all calls to the function named ":". This function passes its argument list as the input buffer for a parse to begin in state S. The form popped by the top level ATN network is then printed, followed by the CPU time in milliseconds. (The system is partly compiled, partly interpreted LISP on a CYBER 173. The ATN grammar is interpreted.) Figure 2 shows the grammar in abbreviated graphical form, and Figure 4 gives the details of each arc. The parsing network, beginning at state SP is included for completeness, but the reader unfamiliar with SNePSUL, the SNePS User Language, [5] is not expected to understand its details.

The first arc in the network is a PUSH to the parsing network. This network determines whether the input is a statement (type D) or a question (type Q). If a statement, the network builds a SNePS network representing the information contained in the sentence and pops a semantic node representing the fact contained in the main clause. If the input is a question the parsing network calls the SNePS deduction routines (DEDUCE) to find the answer, and pops the semantic node representing that (no actual deduction is required in this example). Figure 3 shows the complete SNePS network built during this example. Nodes M74-M85 were built by the first statement, nodes M89 and M90 by the second.

When the state RESPOND is reached, the input buffer contains the SNePS node popped by the parsing network. The generating network then builds a sentence. The first two sentences were generated from node M85 before M89 and M90 were built. The third sentence was generated from M90, and the fourth from M85 again. Since the voice (VC) register is LIFTRed from the parsing network, the generated sentence has the same voice as the input sentence (see Figure 1).

Of particular note is the sub-network at state PRED which analyzes the proper tense for the generated sentence. For brevity, only simple tenses are included here, but the more complicated tenses presented in [4] can be handled in a similar manner. Also of interest is the subnetwork at state ADJS which generates a string of adjectives which are not already scheduled to be in the sentence. (Compare the third and fourth generated sentences of Figure 1.)

7. CONCLUSIONS

A generalization of the ATN formalism has been presented which allows grammars to be written for generating surface sentences from semantic networks. The generalization has involved: adding an optional argument to the TO terminal act; reintroducing the JUMP terminal act; introducing a TO arc similar to the JUMP arc; introducing a CALL arc which is a generalization of the PUSH arc; introducing a GETA form; clarifying the management of the input buffer. The benefits of these few changes are that parsing and generating grammars may be written in the same familiar notation, may be interpreted (or compiled) by a single program, and may use each other in the same parser-generator network grammar.

REFERENCES

[1] Bates, Madeleine. The theory and practice of augmented transition network grammars. In L. Bloc, ed. Natural Language Communication with Computers, Springer-Verlag, Berlin, 1978, 192-259.

[2] 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.

[3] Burton, Richard R. and Woods, Wm. A. A compiling system for augmented transition networks. Preprints of COLING 76: The International Conference on Computational Linguistics, Ottawa, June 1976.

[4] Shapiro, Stuart C. Generation as parsing from a network into a linear string. AJCL Microfiche 33 (1975) 45-62.

[5] Shapiro, Stuart C. The SNePS semantic network processing system. In N.V. Findler, ed., Associative Networks: Representation and Use of Knowledge by Computers, Academic Press, New York, 1979, 179-203.

[6] Simmons, R. and Slocum, J. Generating english discourse from semantic networks. CACM 5, 10 (October 1972), 891-905.

[7] Woods, W.A. Transition network grammars for natural language analysis. CACM 13, 10 (October 1970), 591-606.

[8] Woods, W.A. An experimental parsing system for transition network grammars. In R. Rustin, ed., Natural Language Processing, Algorithmics Press, New York, 1973, 111-154.

```
**(: A DOG KISSED YOUNG LUCY)
(I UNDERSTAND THAT A DOG KISSED YOUNG LUCY)
3769 MSECS

**(: WHO KISSED LUCY)
(A DOG KISSED YOUNG LUCY)
2714 MSECS

**(: LUCY IS SWEET)
(I UNDERSTAND THAT YOUNG LUCY IS SWEET)
2127 MSECS

**(: WHO WAS KISSED BY A DOG)
(SWEET YOUNG LUCY WAS KISSED BY A DOG)
3004 MSECS
```

Figure 1. Example Interaction

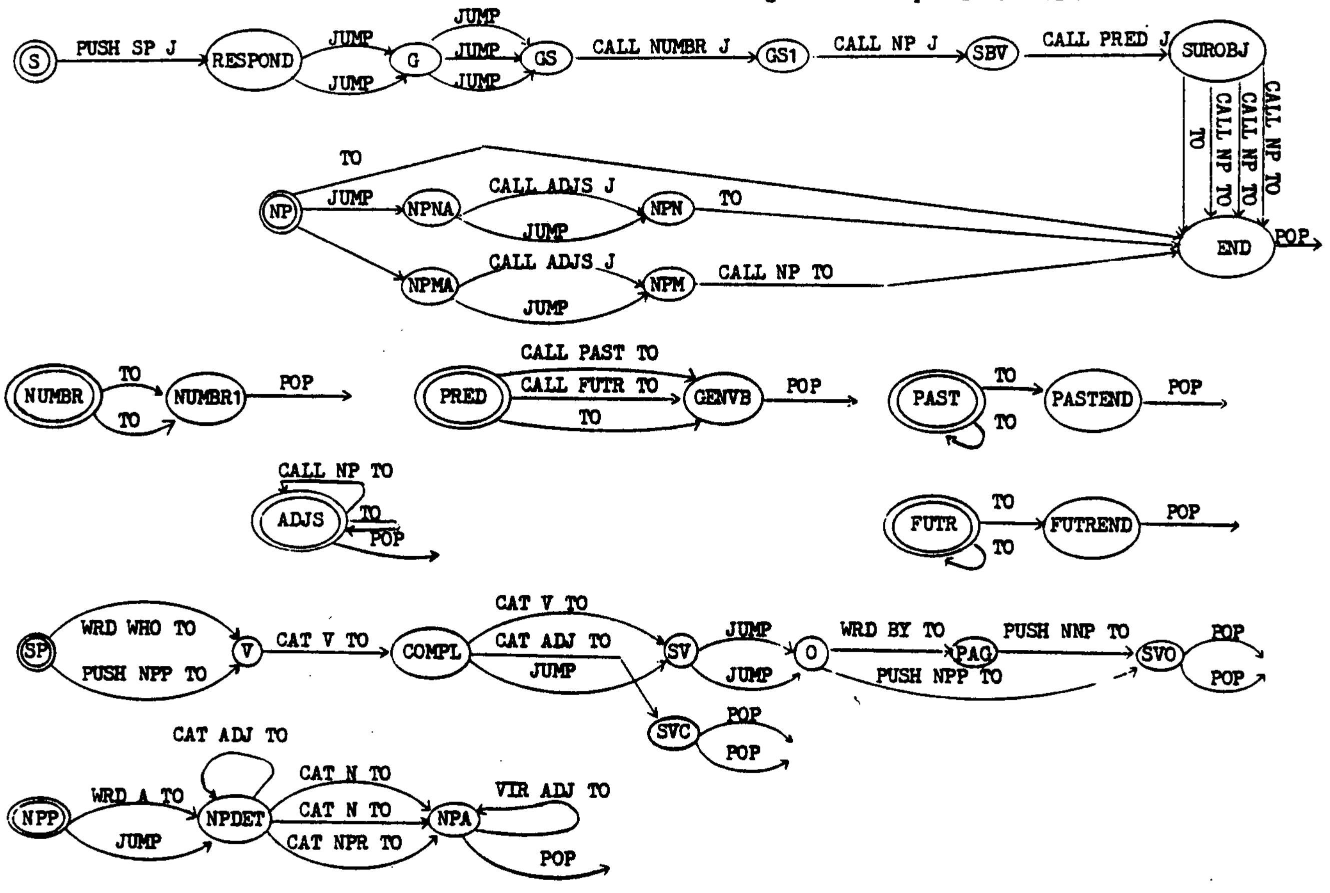

Figure 2. A Parsing-Generating Grammar
Terminal acts are indicated by "J" or "TO"

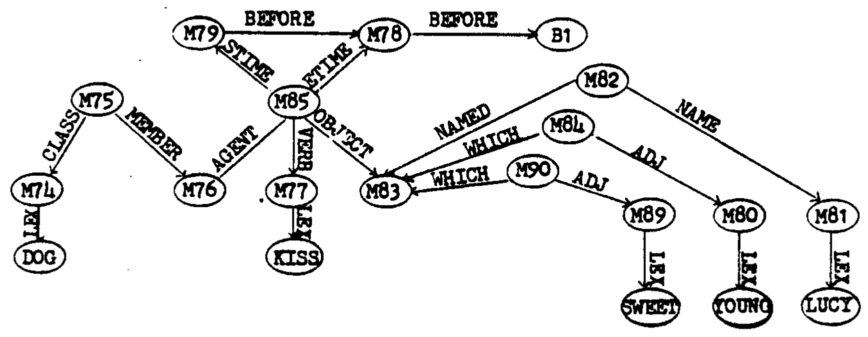

Figure 3. Semantic Network Build by Sentences of Figure 1

```
(S (PUSH SP T (JUMP RESPOND)))
(RESPOND (JUMP G (EQ (GETR TYPE) 'D) (SETR STRING '(I UNDERSTAND THAT)))
         (JUMP G (EQ (GETR TYPE) 'Q)))

(G (JUMP GS (AND (GETA OBJECT) (OVERLAP (GETR VC) 'PASS)) (SETR SUBJ (GETA OBJECT)))
   (JUMP GS (AND (GETA AGENT) (DISJOINT (GETR VC) 'PASS)) (SETR SUBJ (GETA AGENT)) (SETR VC 'ACT))
   (JUMP GS (GETA WHICH) (SETR SUBJ (GETA WHICH)) (SETR VC 'ACT)))
(GS (CALL NUMBR SUBJ T NUMBR (SETR DONE *) (JUMP GS1)))
(GS1 (CALL NP SUBJ T (SENDR DONE) (SENDR NUMBR) REG (ADDR STRING REG) (JUMP SVB)))
(SVB (CALL PRED * T (SENDR NUMBR) (SENDR VC) (SENDR VB (OR (GETA LEX (GETA VERB)) 'BE)) REG (ADDR STRING REG)
          (JUMP SUROBJ)))
(SUROBJ (CALL NP (GETA AGENT) (AND GETA AGENT) (OVERLAP VC 'PASS)) (SENDR DONE) * (ADDR STRING 'BY *) (TO END))
        (CALL NP (GETA OBJECT) (AND (GETA OBJECT) (OVERLAP VC 'ACT)) (SENDR DONE) * (ADDR STRING *) (TO END))
        (CALL NP (GETA ADJ) (GETA ADJ) * (ADDR STRING *) (TO END))
        (TO (END) T))
(END (POP STRING T))
(NUMBR (TO (NUMBR1) (OR (GETA SUB-) (GETA SUP-) (GETA CLASS-)) (SETR NUMBR 'PL))
       (TO (NUMBR1) (NOT (OR (GETA SUB-) (GETA SUP-) (GETA CLASS-))) (SETR NUMBR 'SING)))
(NUMBR1 (POP NUMBR T))
(PRED (CALL PAST (GETA ETIME) T TENSE (TO GENVB))
      (CALL FUTR (GETA STIME) T TENSE (TO GENVB))
      (TO (GENVB) T (SETR TENSE 'PRES)))
(GENVB (POP (VERBIZE (GETR NUMBR) (GETR TENSE) (GETR VC) (GETR VB)) T))
(PAST (TO (PASTEND) (OVERLAP * *NOW))
      (TO (PAST (GETA BEFORE)) T))
(PASTEND (POP 'PAST T))
(FUTR (TO (FUTREND) (OVERLAP * *NOW))
      (TO (FUTR (GETA AFTER)) T))
(FUTREND (POP 'FUTR T))
(NP (TO (END) (GETA LEX) (SETR STRING (WRDIZE (GETR NUMBR) (GETA LEX))))
    (JUMP NPNA (AND (GETA NAMED-) (DISJOINT (GETA NAMED-)DONE)))
    (JUMP NPMA (AND (GETA MEMBER-) (DISJOINT (GETA MEMBER-) DONE))))
(NPNA (CALL ADJS (GETA WHICH-) (GETA WHICH-) (SENDR DONE) REG (ADDR STRING REG) (JUMP NPN))
      (JUMP NPN T))
(NPN (TO END) (ADDR STRING (WRDIZE (GETR NUMBR) (GETA LEX (GETA NAME (GETA NAMED-)))))))
(NPMA (CALL ADJS (GETA WHICH-) (GETA WHICH-) (SENDR DONE) REG (ADDR STRING 'A REG) (JUMP NPM))
      (JUMP NPM T (ADDR STRING 'A)))
(NPM (CALL NP (GETA CLASS (GETA MEMBER-)) T (SENDR DONE) REG (ADDR STRING REG) (TO END)))
(ADJS (CALL NP (GETA ADJ) (DISJOINT * DONE) (SENDR DONE) * (ADDR STRING *) (TO ADJS))
      (TO (ADJS) T)
      (POP STRING T))

(SP (WRD WHO T (SETR TYPE 'Q) (LIFTR TYPE) (SETR SUBJ  %X (TO V))
    (PUSH NPP T (SENDR TYPE 'D) (SETR TYPE 'D) (LIFTR TYPE) (SETR SUBJ *) (TO V)))
(V (CAT V T (SETR VB (FINDORBUILD LEX (↑(GETR *)))) (SETR TNS (GETF TENSE)) (TO COMPL)))
(COMPL (CAT V (AND (GETF PPRT) (OVERLAP (GETR VB) (GETA LEX- 'BE))) (SETR OBJ (GETR SUBJ)) (SETR SUBJ NIL)
            (SETR VC 'PASS) (SETR VB (FINDORBUILD LEX (↑(GETR *)))) (TO SV))
       (CAT ADJ (OVERLAP (GETR VB) (GETA LEX- 'BE)) (SETR ADJ (FINDORBUILD LEX (↑(GETR *)))) (TO SVC))
       (JUMP SV T))
(SV (JUMP O (EQ (GETR TNS) 'PRES) (SETR STM (BUILD BEFORE *NOW (BUILD AFTER *NOW) = ETM)))
    (JUMP O (EQ (GETR TNS) 'PAST) (SETR STM (BUILD BEFORE (BUILD BEFORE *NOW) = ETM))))
(O (WRD BY (EQ (GETR VC) 'PASS) (TO PAG))
   (PUSH NPP T (SENDR TYPE) (SETR OBJ *) (LIFTR VC) (TO SVO)))
(PAG (PUSH NPP T (SENDR TYPE) (SETR SUBJ *) (LIFTR VC) (TO SVO)))
(SVO (POP (BUILD AGENT (↑(GETR SUBJ)) VERB (↑(GETR VB)) OBJECT ( ↑(GETR OBJ)) STIME (↑(GETR STM)) ETIME *ETM)
          (EQ (GETR TYPE 'D))
     (POP (EVAL (BUILDQ (DEDUCE AGENT + VERB + OBJECT +) SUBJ VB OBJ)) (EQ (GETR TYPE) 'Q)))
(SVC (POP (EVAL (BUILDQ (FINDORBUILD WHICH + ADJ +) SUBJ ADJ)) (EQ (GETR TYPE) 'D))
     (POP (EVAL (BUILDQ (DEDUCE WHICH + ADJ +) SUBJ ADJ)) (EQ (GETR TYPE) 'Q)))
(NPP (WRD A T (SETR INDEF T) (TO NPDET))
     (JUMP NPDET T))
(NPDET (CAT ADJ T (HOLD (FINDORBUILD LEX (↑(GETR *)))) (TO NPDET))
       (CAT N (AND (GETR INDEF) (EQ (GETR TYPE) 'D))
            (SETR NH (BUILD MEMBER- (BUILD CLASS (FINDORBUILD LEX ( ↑(GETR *)))))) (TO NPA))
       (CAT N (AND (GETR INDEF) (EQ (GETR TYPE) 'Q))
              (SETR NH (FIND MEMBER- (DEDUCE MEMBER %Y CLASS (TBUILD LEX (↑(GETR *)))))) (TO NPA))
       (CAT NPR T (SETR NH (FINDORBUILD NAMED- (FINDORBUILD NAME (FINDORBUILD LEX (↑(GETR *)))))) (TO NPA)))
(NPA (VIR ADJ T (EVAL (BUILDQ (FINDORBUILD WHICH + ADJ *) NH)) (TO NPA))
     (POP NH T))
```

Figure 4. Details of the Parser-Generator Network