Generalized Augmented Transition Network Grammars For Generation From Semantic Networks[1]

Stuart C. Shapiro

Department of Computer Science
State University of New York at Buffalo
Amherst, New York 14226

The augmented transition network (ATN) is a formalism for writing parsing grammars that has been much used in Artificial Intelligence and Computational Linguistics. A few researchers have also used ATNs for writing grammars for generating sentences. Previously, however, either generation ATNs did not have the same semantics as parsing ATNs, or they required an auxiliary mechanism to determine the syntactic structure of the sentence to be generated. This paper reports a generalization of the ATN formalism that allows ATN grammars to be written to parse labelled directed graphs. Specifically, an ATN grammar can be written to parse a semantic network and generate a surface string as its analysis. An example is given of a combined parsing-generating grammar that parses surface sentences, builds and queries a semantic network knowledge representation, and generates surface sentences in response.

1. Introduction

Augmented transition network (ATN) grammars have, since their development by Woods 1970,1973, 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" [Bates 1978, p. 191].

The usual method of utilizing an ATN grammar in a natural language system is to provide an interpreter that 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. The input sentence is assumed to be a linear sequence of symbols, while the parse is usually a tree (often represented by a LISP S-expression) or some "knowledge representation" such as a semantic network. Compilers have been written [Burton 1976; Burton and Woods 1976] that take an ATN grammar as input and produce a specialized parser for that grammar, but in this paper we assume that an interpreter is being used.

Several methods have been described for using ATN grammars for sentence generation. One method [Bates 1978, p. 235] is to replace the usual interpreter by a generation interpreter that 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.

Simmons 1973 uses a standard ATN 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. For example, the input string might be (CA1-LOCUS VACT THEME), where CA1-LOCUS and THEME are labels of arcs emanating from the semantic node, and VACT stands for "active verb." This pattern means that the sentence to be generated is to begin with a string denoting the CA1-LOCUS, then have the active form of the verb, and end with a string denoting the THEME. The method also assumes that semantic nodes have such syntactic information stored with them as number and definiteness of nominals, and tense, aspect, mood, and voice of propositions.

[1] This paper is a revised and expanded version of one given at the 17th Annual Meeting of the Association for Computational Linguistics. The work reported here was supported in part by the National Science Foundation under Grants MCS78-02274 and MCS80-06314.

0362-613X/82/010012-14$01.00

Shapiro 1979 also generates sentences from a semantic network. In this method, input to the grammar is the semantic network itself (starting at some node). That is, instead of 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, but bases specific decisions on the structural properties of the semantic network and on the information contained therein.

The original goal in Shapiro 1975 was that a single ATN interpreter could be used both for standard ATN parsing and for generation. However, a special interpreter was written for generation grammars; indeed, the semantics of the ATN formalism given in that paper, though based on the standard ATN formalism, were inconsistent enough with the standard notation that a single interpreter could not be used. For example, standard ATNs use a register named "*" to hold the input symbol (word) currently being scanned. Unlike other registers, whose values are set explicitly by actions on the ATN arcs, the * register is manipulated directly by the ATN interpreter. In Shapiro 1975 the * register was used to hold the string being generated rather than the input symbol being scanned. The interpreter written for Shapiro 1975 also manipulated the * register directly, but in a manner inconsistent with standard ATN interpreters.

This paper reports the results of work carried out to remove the inconsistencies mentioned above. A generalization of the ATN formalism has now been derived that supplies consistent semantics (and so allows a single interpreter to be used) for both parsing and generating grammars. In fact, one grammar can include both parsing and generating sub-networks that can call each other. For example, an ATN grammar can be constructed so that the "parse" of a natural language question is the natural language statement that 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.

The generalized ATN formalism can be used to write grammars for parsing labelled directed graphs. In this paper, however, we only discuss its use in parsing two particular kinds of labelled di-graphs. One is the kind that is generally called a semantic network. We consider parsing a semantic network, as viewed from some node, into a particular linear symbol structure that constitutes a surface string of English. The other kind of labelled di-graph is a linear graph all of whose arcs have the same label and whose nodes are successive words in a surface sentence. This kind of di-graph is so special that a subset of the generalized ATN formalism, namely the original formalism, has built-in facilities for traversing its arcs.

Since many people have implemented their own ATN interpreters, this paper is written to describe the extension to be made to any ATN interpreter to allow it to interpret generation grammars as well as parsing grammars. A key ingredient in such an extension is a systematic treatment of the input buffer and the * register. This is explained in Section 4, which is essentially a description of a set of program assertions for ATN interpreters.

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 analyze 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 that express the concepts represented by adjacent nodes. For example, if a node represents a proposition 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 normally continues at a different symbol. In the generation approach we use, a recursive push normally 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 defined by the system, but in "parsing" a network, "next" must be specified in the grammar.

3. The Generalization

The following sub-sections show the generalized syntax of the ATN formalism, and assume a knowledge of the standard formalism (Bates 1978 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.

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 Woods 1970 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 in Burton 1976 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." That our reintroduction of the JUMP terminal action does not bring back the confusion is explained 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. For symmetry, we 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.

Neither the JUMP arc nor the TO arc are really required if the TST arc is retained (Bates 1978, however, does not mention it), since they are equivalent to the TST arc with the JUMP or TO terminal action, respectively. However, they require less typing and provide clearer documentation. They are used in the example in Section 6.

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 without the optional form, the PUSH arc (which we retain) is equivalent to the following CALL arc:

```
(CALL <state> * <test> <preaction>...
      * <action>... <terminal action> )
```

3.3 Forms

The generalized TO terminal action, the generalized TO arc, and the CALL arc all include a form whose value is to be the next symbol to be scanned. If this next symbol is a semantic network node, the primary way of identifying it is as the node at the end of a directed arc with a given label from a given node. This identification mechanism requires a new form:

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

where <node form> is a form that 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, Preactions, and Actions

The generalization of the ATN formalism to one that 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. Specifically, no new tests, preactions, or actions are required. 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.

Adequate treatment of the * register is crucial for the correct operation of a grammar interpreter that does both parsing and generation. This is dealt with in the present section.

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 CAT, VIR, and POP arcs. On a CAT arc, * is the root form of the top element of the input buffer. (Since the CAT arc is treated as a "bundle" of arcs, one for each sense of the word being scanned, and is the only arc so treated, it is the only arc on which (GETF <feature> *) is guaranteed to be well-defined.) 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 remain the same. This requires special processing for VIR, PUSH, and CALL arcs. Since a VIR arc gets the value of * from HOLD, rather than from the input buffer, after setting * the VIR arc pushes the contents of * onto the input buffer. The net effect is to replace the held constituent in a new position in the string. 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. The net effect of this is to replace a sub-string by its analysis. 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. (Recall that it was replaced before the transfer. See the previous paragraph.) 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 the 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 having the operational characteristics given in this section, 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 their syntactic categories and lexical features, as well as the inflectional forms of irregularly inflected words. Parsing and generating require different information, yet we wish to avoid duplication as much as possible. This section discusses how a lexicon might be organized when it is to be used both for parsing and for generation. Figure 1 shows the lexicon used for the example in Section 6.

During parsing, morphological analysis is performed. The analyzer is given an inflected form and must segment it, find the root in the lexicon, and modify the lexical entry of the root according to its analysis of the original form. Irregularly inflected forms, such as "seen" in Figure 1, must have their own entries in the lexicon. An entry in the lexicon may be lexically ambiguous, such as "saw" in Figure 1, 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 root, as well as other features needed by the grammar. The lexical routines we use supply certain default features if they are not supplied explicitly. These are as follows: the root is the lexeme itself; nouns have (NUM . SING); verbs have (TENSE . PRES). In Figure 1, BE and DOG get default features, while the entries for SAW override several of them.

```
(A      ((CTGY . DET)))
(BE     ((CTGY . V)))
(DOG    ((CTGY . N)))
(IS     ((CTGY . V) (ROOT . BE) (NUM . SING) (TENSE . PRES)))
(LUCY   ((CTGY . NPR)))
(SAW    ((CTGY . N) (ROOT . SAW1))
        ((CTGY . V) (ROOT . SEE) (TENSE . PAST)))
(SAW1   ((CTGY . N) (ROOT . SAW)))
(SEE    ((CTGY . V) (PAST . SAW) (PASTP . SEEN)))
(SEEN   ((CTGY . V) (ROOT . SEE) (TENSE . PASTP) (PPRT . T)))
(SWEET  ((CTGY . ADJ)))
(WAS    ((CTGY . V) (ROOT . BE) (NUM . SING) (TENSE . PAST)))
(YOUNG  ((CTGY . ADJ)))
```

Figure 1. Example Lexicon.

In the semantic network, some nodes are associated with lexical entries. In Figure 3, nodes SWEET, YOUNG, LUCY, BE, SEE, and SAW1 are. 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 roots, and so contain only a single lexical feature list. This feature list must contain any irregularly inflected forms. For example, the feature list for "see" in Figure 1 lists "saw" as its past tense and "seen" as its past participle. SAW1 represents the unambiguous sense of "saw" as a noun. It is used in that way in Figure 3. In Figure 1, SAW1 is given as the ROOT of the noun sense of SAW, but for purposes of morphological synthesis, the ROOT of SAW1 is given as SAW.

In summary, a single lexicon may be used for both parsing and generating under the following conditions. The entry of an unambiguous root can be used for both parsing and generating if its one lexical feature list contains features required for both operations. An ambiguous lexical entry (such as SAW) will only be used during parsing. Each of its lexical feature lists must contain a unique but arbitrary "root" (SEE and SAW1) 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 root (SAW for SAW1) as well as any irregularly inflected forms. Lexical entries for irregularly inflected forms will only be used during parsing. In the lexicon of Figure 1, the entries for A, DOG, LUCY, SEE, SWEET, and YOUNG are used during both parsing and generation. Those for BE, IS, SAW, SEEN, and WAS are only used during parsing. The entry for SAW1 is only used during generation. Our morphological synthesizer recognizes "be" as a special case, and computes its inflected forms without referring to the lexicon.

For the purposes of this paper, it should be irrelevant whether the "root" connected to the semantic network is an actual surface word like "give", a deeper sememe such as that underlying both "give" and "take", or a primitive such as "ATRANS".

6. Example

In this section, we discuss an example of natural language interaction (in a small fragment of English) using an ATN parsing-generating grammar and SNePS, the Semantic Network Processing System [Shapiro 1979]. The purpose of the example is to demonstrate the use of the generalized ATN formalism for writing a parsing-generating grammar for which the "parse" of an input sentence is a generated sentence response, using a knowledge representation and reasoning system as the sentence is processed. Both the fragment of English and the semantic network representation technique have been kept simple to avoid obscuring the use of the generalized ATN formalism.

Figure 2 shows an example interaction using SNePSUL, the SNePS User Language. The numbers in the left margin are for reference in this section. The string "**" is the SNePSUL prompt. The rest of each line so marked is the user's input. The following line is the result returned by SNePSUL. The last line of each interaction is the CPU time in milliseconds taken by the interaction. (The system is running as compiled LISP on a CDC CYBER 170/730. The ATN grammar is interpreted.) Figure 3 shows the semantic network built as a result of the sentences in Figure 2.

The first interaction creates a new semantic network node, shown as B1 in Figure 3, to represent the instant of time "now". The symbol "#" represents a SNePSUL function to create this node and make it the value of the variable NOW. From then on, the ex-

```
     ** #NOW
      (B1)
      4 MSECS

(1)  ** (: YOUNG LUCY SAW A SAW)
      (I UNDERSTAND THAT YOUNG LUCY SAW A SAW)
      2481 MSECS

(2)  ** (: WHO SAW A SAW)
      (YOUNG LUCY SAW A SAW)
      875 MSECS

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

(4)  ** (: WHAT WAS SEEN BY LUCY)
      (A SAW WAS SEEN BY SWEET YOUNG LUCY)
      862 MSECS
```

Figure 2. Example interaction.

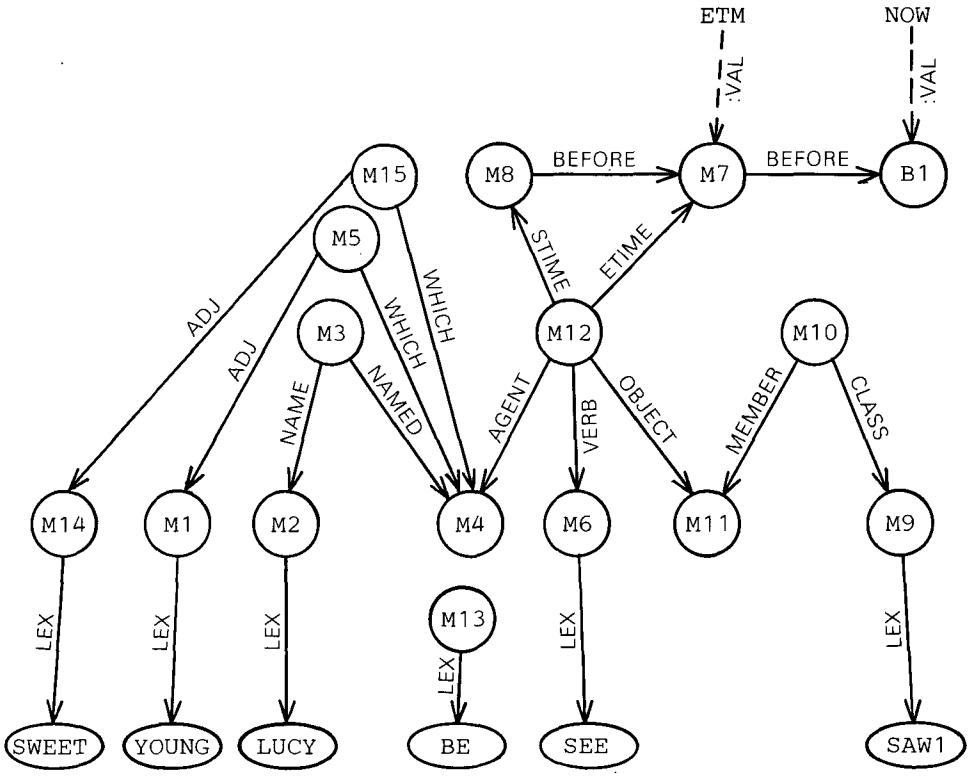

Figure 3. The semantic network built by the example interaction.

pression *NOW evaluates to B1. We will see *NOW used on some arcs of the grammar.

The rest of the user inputs are calls to the SNePSUL function ":". This function passes its argument list to the parser as the input buffer. The parser starts in state S. The form popped by the top level ATN grammar is returned as the value of the call to :, and is then printed as mentioned above. Thus, the line following the call to : may be viewed as the "parse" of the sentence passed to :.

We will trace the first example sentence through the ATN grammar, referring to the other example sentences at various points. The parse starts in state S (Figure 4) with the input buffer being (YOUNG LUCY SAW A SAW). The one arc out of state S pushes to state SP, which is the beginning of a network that parses a sentence. The first arc out of state SP recognizes that inputs (2) and (4) are questions. In these cases the register TYPE is set to Q both on this level (by the SETR action) and on the top level (by the LIFTR action). The register SUBJ is also set to %X, which is a free variable in SNePSUL. In the cases of sentences (1) and (3), however, the second arc from SP is taken. This is a PUSH to the network beginning at state NPP, which will parse a noun phrase. Register TYPE is initialized to D (for "declarative") at the lower level by SENDR. When the noun phrase is parsed, the TYPE register is set to D at this and the top level, and the SUBJ register is set to the parse of the noun phrase.

At state NPP (Figure 5) the JUMP arc is followed since the * register is YOUNG rather than A. At state NPDET, the CAT ADJ arc is followed and a semantic node representing the concept of YOUNG is put in the HOLD register for later use. The SNePSUL form (FINDORBUILD LEX (^(GETR *))) finds in the network a node with a LEX arc to YOUNG (the contents of the * register) or builds such a node if one does not exist, and returns that node as the value of the form. In this case, node M1 of Figure 3 is built and placed in HOLD. The parser then loops TO NPDET with the input buffer being (LUCY SAW A SAW). This time, the * register contains LUCY, so the fourth arc is followed. This time the SNePSUL form builds nodes M2, M3, and M4 of Figure 3, and places M4 in the register NH. Node M4 represents someone named LUCY. Node M3 represents the proposition that this person is named LUCY. Node M2 represents the name LUCY. (When this arc is taken while parsing LUCY in sentences (3) and (4), these semantic nodes will be found.) The parser then transfers TO state NPA at which the modifying properties are removed from the HOLD register and asserted to hold of the concept stored in NH. In this case, there is only one property, and node M5 is built. Node M4, representing someone who is named LUCY and is YOUNG is popped to the PUSH arc emanating from state SP, and is placed in the SUBJ register as mentioned earlier. The parser then transfers TO state V with an input buffer of (SAW A SAW).

The CAT arc from state V (Figure 6) wants a word of category V. The first word in the input buffer is SAW, which is two-ways lexically ambiguous (see Figure 1), so we can think of the CAT arc as being two arcs, on one of which * contains the singular of the Noun SAW1, and on the other of which * contains the past tense of the Verb SEE. The second of these arcs can be followed, setting the register VB to node M6, and the register TNS to PAST. The parser than goes TO state COMPL with input buffer (A SAW). At COMPL, neither CAT arc can be followed, so the parser JUMPs to state SV. The first CAT arc is followed in sentence 4, while the second CAT arc is followed in sentence 3.

At state SV (Figure 7), a semantic network temporal structure is built for events. Each event is given a starting time and an ending time. Present tense is interpreted to mean that the present time, the value of *NOW, is after the starting time and before the ending time. Past tense is interpreted to mean that the ending time is before *NOW. In this case, the tense is past, so the third arc is taken and builds nodes M7 and M8. M7 is made the SNePSUL value of *ETM, and M8 is placed in the ATN register STM. For simplicity in this example, the first arc ignores the tense of questions. Control then passes to state O.

The first arc of state O (Figure 8) recognizes the beginning of a "by" prepositional phrase in a passive sentence. This arc will be followed in the case of sentence 4 to the state PAG where the object of BY will replace the previous contents of the SUBJ register. In the case of sentence (1), the second arc will be taken, which is a PUSH to state NPP with input buffer, (A SAW).

```
(S ; Parse a sentence and generate a response.
   (PUSH SP T (JUMP RESPOND)))

(SP ; Parse a sentence.
   (WRD (WHO WHAT)
       ; If it starts with "Who" or "What", it's a question.
       T (SETR TYPE 'Q) (LIFTR TYPE) (SETR SUBJ %X) (TO V))
   (PUSH NPP ; A statement starts with a Noun Phrase -- its subject.
       T (SENDR TYPE 'D) (SETR TYPE 'D) (LIFTR TYPE) (SETR SUBJ *)
       (TO V)))
```

Figure 4. ATN Grammar.

At state NPP (Figure 5), the first arc is taken, setting register INDEF to T, and transferring TO state NPDET with input buffer (SAW). The second arc is taken from NPDET interpreting SAW as a noun, the singular form of SAW1. A semantic node is found or built (in this case M9 is built) to represent the class of SAW1s, M11 is built to represent a new SAW1, and M10 is built to assert that M11 is a SAW1. In the case of a question, such as sentence (2), the third arc finds all known SAW1s using the SNePSUL function DEDUCE, so that whatever can be inferred to be a SAW1 is found. In either case the SAW1(s) are

```
(NPP ; Parse a noun phrase.
   (WRD A T (SETR INDEF T) (TO NPDET))
   (JUMP NPDET T))

(NPDET ; Parse a NP after the determiner.
   (CAT ADJ T ; Hold adjectives for later.
       (HOLD 'ADJ (FINDORBUILD LEX (^(GETR *)))) (TO NPDET))
   (CAT N (AND (GETR INDEF) (EQ (GETR TYPE) 'D))
       ; "a N" means some member of the class Noun,
       (SETR NH ; but not necessarily any one already known.
           (BUILD MEMBER-
               (BUILD CLASS (FINDORBUILD LEX (^(GETR *))))))
       (TO NPA))
   (CAT N (AND (GETR INDEF) (EQ (GETR TYPE) 'Q))
       ; "a N" in a question refers to an already known Noun.
       (SETR NH
           (FIND MEMBER-
               (DEDUCE MEMBER %Y CLASS (TBUILD LEX (^(GETR *))))))
       (TO NPA))
   (CAT NPR T ; A proper noun is someone's name.
       (SETR NH (FINDORBUILD NAMED-
                   (FINDORBUILD NAME (FINDORBUILD LEX (^(GETR *))))))
       (TO NPA)))

(NPA ; Remove all held adjectives and build WHICH-ADJ propositions.
   (VIR ADJ T
       (FINDORBUILD WHICH (^ (GETR NH)) ADJ (^ (GETR *)))
       (TO NPA))
   (POP NH T))
```

Figure 5. ATN Grammar (continued).

```
(V (CAT V T ; The next word must be a verb.
       (SETR VB (FINDORBUILD LEX (^(GETR *)))) (SETR TNS (GETF TENSE))
       (TO COMPL)))

(COMPL ; Consider the word after the verb.
   (CAT V (AND (GETF PPRT) (OVERLAP (GETR VB) (GETA LEX- 'BE)))
       ; It must be a passive sentence.
       (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))
       ; a predicate adjective.
       (SETR ADJ (FINDORBUILD LEX (^(GETR *)))) (TO SVC))
   (JUMP SV T))
```

Figure 6. ATN Grammar (continued).

placed in register NH, and the parser transfers TO state NPA with an empty input buffer, and the value of NH is popped back to the PUSH arc from state O or PAG where it becomes the value of the * register and the first (and only) item on the input buffer. After setting the SUBJ or OBJ register, these PUSH arcs LIFTR the proper voice to the VC register on the top ATN level and transfer TO state SVO (Figure 9).

When dealing with sentence (1), the first POP arc at state SVO builds node M12 and pops it to the top level. The second POP arc finds M12 for both sentences (2) and (4). In the case of sentence (3), the CAT ADJ arc is followed from state COMPL to state SVC, on the first arc of which node M15 is built. The second arc from SVC is not used in these examples.

The pop from state SVO or SVC returns to the PUSH arc from state S which JUMPs to the state RESPOND with the input buffer containing either the node built to represent an input statement or the node that represents the answer to an input question. In our example, this is node M12 for inputs (1), (2), and (4), and node M15 for input (3). Remember that nodes M14 and M15 do not exist until sentence (3) is analyzed.

The state RESPOND (Figure 10) is the initial state of the generation network. In this network the register STRING is used to collect the surface sentence being built. The only difference between the two arcs in state RESPOND is that the first, responding to input statements, starts the output sentence with the

```
(SV ; Start building the temporal structure.
   (JUMP O (EQ (GETR TYPE) 'Q)) ; Ignore the tense of a question.
   (JUMP O (EQ (GETR TNS) 'PRES)
       ; Present means starting before and ending after now.
       (SETR STM (BUILD BEFORE *NOW BEFORE (BUILD AFTER *NOW) = ETM)))
   (JUMP O (EQ (GETR TNS) 'PAST)
       ; Past means starting and ending before now.
       (SETR STM (BUILD BEFORE (BUILD BEFORE *NOW) = ETM))))
```

Figure 7. ATN Grammar (continued).

```
(O ; Parse what follows the verb group.
   (WRD BY (EQ (GETR VC) 'PASS) ; Passive sentences have "by NP".
       (TO PAG))
   (PUSH NPP T ; Active sentences have an object NP.
       (SENDR TYPE) (SETR OBJ *) (LIFTR VC) (TO SVO)))

(PAG (PUSH NPP T ; Parse the subject NP of a passive sentence.
         (SENDR TYPE) (SETR SUBJ *) (LIFTR VC) (TO SVO)))
```

Figure 8. ATN Grammar (continued).

```
(SVO ; Return a semantic node.
   (POP (BUILD AGENT (^(GETR SUBJ)) VERB (^(GETR VB))
           OBJECT (^(GETR OBJ))  STIME (^(GETR STM)) ETIME *ETM)
       (EQ (GETR TYPE) 'D)) ; An Agent-Verb-Object statement.
   (POP (DEDUCE AGENT (^(GETR SUBJ)) VERB (^(GETR VB))
               OBJECT (^(GETR OBJ)))
       (EQ (GETR TYPE) 'Q))) ; An Agent-Verb-Object question.

(SVC (POP (EVAL (BUILDQ (FINDORBUILD WHICH + ADJ +) SUBJ ADJ))
         (EQ (GETR TYPE) 'D)) ; A Noun-be-Adj statement.
     (POP  (DEDUCE WHICH (^(GETR SUBJ)) ADJ (^(GETR ADJ)))
         (EQ (GETR TYPE) 'Q))) ; A Noun-be-Adj question.
```

Figure 9. ATN Grammar (continued).

```
(RESPOND
   ; Generate the response represented by the semantic node in *.
   (JUMP G (EQ (GETR TYPE) 'D)
       ; The input was a statement represented by *.
       (SETR STRING '(I UNDERSTAND THAT)))
   (JUMP G (EQ (GETR TYPE) 'Q)))
       ; The input was a question answered by *.
```

Figure 10. ATN Grammar (continued).

phrase I UNDERSTAND THAT. Then both arcs JUMP to state G. We follow the generation process assuming that the input buffer is now (M12).

In state G (Figure 11), the node representing the statement to be generated is analyzed to decide what kind of sentence will be produced. The first arc, for a passive version of M12, sets the SUBJ register to M11 (the saw), the OBJ register to M4 (Lucy), and the PREP register to the word BY. (Note the use of GETA, defined in Section 3.3.) The second arc, for an active version of M12, sets SUBJ to M4, OBJ to M11, and leaves PREP empty. It also makes sure VC is set to ACT, since active voice is the default if VC is empty. The third arc is for generating sentences for nodes such as M15. In that case it sets SUBJ to M4, OBJ to M14 (the property SWEET), VC to ACT, and leaves PREP empty. All three arcs then JUMP to state GS.

The CALL arc at state GS (Figure 12) sets the NUMBR register to SING or PL to determine whether the subject and verb of the sentence will be singular or plural, respectively. It does this by CALLing the network beginning at state NUMBR, sending it the contents of the SUBJ register, and placing the form returned by the lower network into the NUMBR register. Let us assume we are generating an active version of M12. In that case when state NUMBR (Figure 12) is reached, the input buffer will be (M4), and when the parser returns to the CALL arc the input buffer will again be (M12).

At state NUMBR, the semantic network attached to the node in the * register is examined to determine if it represents a (singular) individual or a (plural) class of individuals. The first arc decides on PL if the node has a SUB-, SUP-, or CLASS- arc emanating from it. These arcs are the converses of SUB, SUP, and CLASS arcs, respectively. The first would occur if the node represented the subset of some class. The second would occur if the node represented the superset of some class. The third would occur if the node represented a class with at least one member. In our example, the only semantic node that would be recognized by this arc as representing a class would be M9. The second arc from state NUMBR decides on SING if none of the three mentioned arcs emanate from the node in *, and in our case this is the successful arc. The decision is made by placing SING or PL in the NUMBR register, and transferring TO state NUMBR1. There the input buffer is empty and the contents of NUMBR is popped to the CALL arc in the state GS as discussed above. The last thing the CALL arc in state GS does is set the DONE register to the node in *. This register is used to remember the node being expressed in the main clause of the sentence so that it is not also used to form a subordinate clause or description. For example, we would not want to generate "Lucy, who saw a saw, saw a saw." This is used

```
(G ; Generate a sentence to express the semantic node in *.
   (JUMP GS (AND (GETA OBJECT) (OVERLAP (GETR VC) 'PASS))
       ; A passive sentence is "OBJECT VERB by AGENT".
       (SETR SUBJ (GETA OBJECT)) (SETR OBJ (GETA AGENT))
       (SETR PREP 'BY))
   (JUMP GS (AND (GETA AGENT) (DISJOINT (GETR VC) 'PASS))
       ; An active sentence is "AGENT VERB OBJECT".
       (SETR SUBJ (GETA AGENT)) (SETR OBJ (GETA OBJECT))
       (SETR VC 'ACT))
   (JUMP GS (GETA WHICH) (SETR SUBJ (GETA WHICH))
       ; A WHICH-ADJ sentence is "WHICH be ADJ".
       (SETR OBJ (GETA ADJ)) (SETR VC 'ACT)))
```

Figure 11. ATN Grammar (continued).

effectively in the response to statement 3 to prevent the response from being "I UNDERSTAND THAT SWEET YOUNG LUCY IS SWEET". We will see where DONE is used in the ATN network shortly. The parser then JUMPs to state GS1, where NP is CALLed with input buffer (M4), DONE set to M12, and NUMBR set to SING.

State NP (Figure 13) is the beginning of a network that generates a noun phrase to describe the concept represented by the semantic node in the * register (in this case, M4). The first arc just uses the node at the end of the LEX arc if one exists, as it does for nodes M1, M2, etc. WRDIZE is a LISP function that does morphological synthesis for nouns. Its first argument must be SING or PL, and its second argument must be a non-ambiguous lexeme in the lexicon. Nouns whose singular or plural forms are irregular must have them explicitly noted in the lexical feature list. The regular rule is to use the ROOT form as the singular, and to pluralize according to rules built into WRDIZE that operate on the ROOT form. For example, the singular of SAW1 is its ROOT, SAW, and its plural is SAWS.

The second arc in the state NP uses a proper name to describe *, if it has one, and if the proposition that this name is *'s name is not the point of the main

```
(GS ; Set the NUMBR register to the number of the subject,
    ; and the DONE register to the proposition of the main clause.
    (CALL NUMBR SUBJ T NUMBR (SETR DONE *) (JUMP GS1)))

(GS1 (CALL NP SUBJ T ; Generate a NP to express the subject.
         (SENDR DONE) (SENDR NUMBR) REG
         (ADDR STRING REG) (JUMP SVB)))

(NUMBR
   ; The proper number is PL for a class, SING for an individual.
   (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)) ; Return the number.
```

Figure 12. ATN Grammar (Continued).

```
(NP ; Generate a NP to express *.
   (TO (END) (GETA LEX)
       ; Just use the word at the end of the LEX arc if present.
       (SETR STRING (WRDIZE (GETR NUMBR) (GETA LEX))))
   (CALL ADJS (GETA WHICH-) ; If it has a name,
       (AND (GETA NAMED-) (DISJOINT (GETA NAMED-) DONE))
       (SENDR DONE) REG
       (ADDR STRING REG) ; add an adjective string,
       (TO NPGA (GETA NAME (GETA NAMED-)))) ; and consider its name.

   (CALL ADJS (GETA WHICH-) ; If it has a class,
       (AND (GETA MEMBER-) (DISJOINT (GETA MEMBER-) DONE))
       (SENDR DONE) REG     ; add 'A and an adjective string,
       (ADDR STRING 'A REG) ; and consider its class.
       (TO NPGA (GETA CLASS (GETA MEMBER-)))))

(NPGA ; Generate a noun phrase for the name or class.
   (PUSH NP T (SENDR DONE) (ADDR STRING *) (TO END)))

(END (POP STRING T)) ; Return the string that has been built.
```

Figure 13. ATN Grammar (continued).

clause. The third arc uses the phrase "a <class>" if * is known to be a member of some class, and if that fact is not the main clause of the sentence. Both arcs first call ADJS to form an adjective string to be included in the noun phrase.

The network starting at ADJS (Figure 14) is CALLed to generate a string of adjectives. For this purpose, it is "passed" the set of property assertion nodes, and its DONE register is set. Let us consider the four cases in which a noun phrase is being generated to describe M4. In sentences (1) and (2), the input buffer at state ADJS is (M5) and DONE contains (M12). In sentence (3), the input buffer is (M15 M5) and DONE is (M15). In sentence (4), the input buffer is (M15 M5) and DONE contains (M12). The CALL arc in state ADJS calls the NP network to generate a description of the property at the end of the ADJ arc from the node in * (the first node in the input buffer) as long as the node in * is not also in DONE. It adds this description to the register STRING and loops back TO ADJS, consuming the node in * from the input buffer. We have already seen how the NP network will generate SWEET for M14 and YOUNG for M1. The second arc in state ADJS consumes the first node in the input buffer without generating a description for its property. The third arc POPs back to the CALLing arc, returning the constructed adjective string in STRING. If we view the ATN as a non-deterministic machine, the result of the ADJS network is a string of zero or more of the adjectives that describe the individual of the noun phrase, but not the adjective, if any, in the predicate of the higher clause. Viewed deterministically, since most ATN interpreters try arcs in strict order, the network will generate a string of all appropriate adjectives.

Returning to our main example of sentence (1), the string YOUNG is POPped into the REG register on the first CALL arc in state NP, where it is added to the end of the register STRING (previously empty). The parser then jumps TO state NPGA, and, because of the form in the terminal action, the input buffer is changed from (M4) to (M2). At state NPGA, the parser PUSHes to state NP where, as we have seen, LUCY will be generated and POPped back into the * register. This is added to STRING, forming (YOUNG LUCY), and the parser goes TO state END, emptying the input buffer. At END, the contents of STRING, (YOUNG LUCY) is POPped to the register REG in the CALL arc of state GS1 (Figure 12), and added to the top level of STRING, which is now (I UNDERSTAND THAT YOUNG LUCY). The parser then JUMPs to state SVB with the input buffer restored to (M12).

At state SVB (Figure 15), the network beginning at state PRED is CALLed. At that level the input buffer is (M12), * contains M12, NUMBR contains SING, VC contains ACT (PASS in the case of sentence 4), and VB contains SEE. Notice that if the main proposition node has no VERB arc, BE is placed in VB. This is the situation in the case of sentence (3), reflecting the theory that in copulative sentences BE is only a dummy verb used to carry tense, mood, and aspect.

The arcs at state PRED (Figure 15) determine and place in the TENSE register the tense of the sentence being generated. For simplicity in this example, we only consider simple present, past, and future. This is one of the most interesting sections of this generation grammar, because it is a grammar that analyzes (parses) a piece of the semantic network. The first CALL arc of state PRED calls a network that recognizes the temporal structure indicating past tense. The second CALL arc calls a network that recognizes the temporal structure indicating future tense. The third, TO, arc chooses present tense as the default. The CALL arcs pass to the lower network the appropriate semantic temporal nodes. Since past tense is indicated by the action ending before *NOW, the first arc passes the node at the end of the ETIME arc from the proposition node. Since future tense is indicated by the action starting after *NOW, the second arc passes the node at the end of the STIME arc. In our case, the tense will be past, so we turn to the PAST subnetwork (Figure 16).

The first arc in state PAST transfers TO state PASTEND, which simply POPs the atom PAST if the contents of * OVERLAPs the value of *NOW, that is, if the * register contains the semantic node representing now. If it doesn't, the second arc in PRED replaces the first node in the input buffer by the node(s) representing times known to be after it, and loops back to PRED. This sub-network can only succeed, returning

```
(ADJS
   ; Generate a string of adjectives, one for each WHICH-ADJ node in *.
   (CALL NP (GETA ADJ) (DISJOINT * DONE) (SENDR DONE) *
       (ADDR STRING *) (TO ADJS))
   (TO (ADJS) T)
   (POP STRING T))
```

Figure 14. ATN Grammar (continued).

```
(SVB (CALL PRED * T
         ; Generate a verb group.  Use "be" if no other verb.
         (SENDR NUMBR) (SENDR VC)
         (SENDR VB (OR (GETA LEX (GETA VERB)) 'BE))
         REG (ADDR STRING REG) (JUMP SUROBJ)))

(SUROBJ (CALL NP OBJ OBJ
           ; Generate a NP to express the OBJ if there is one.
           (SENDR DONE) * (ADDR STRING PREP *) (TO END))
        (TO (END) T))

(PRED ; Figure out the proper tense.
   (CALL PAST (GETA ETIME) T TENSE (TO GENVB))
       ; Past tense depends on ending time.
   (CALL FUTR (GETA STIME) T TENSE (TO GENVB))
       ; Future tense depends on starting time.
   (TO (GENVB) T (SETR TENSE 'PRES))) ; Present tense is the default.

(GENVB ; Return the verb group.
   (POP (VERBIZE (GETR NUMBR) (GETR TENSE) (GETR VC) (GETR VB)) T))
```

Figure 15. ATN Grammar (continued).

```
(PAST ; If we can get to *NOW by BEFORE arcs, it is past tense.
   (TO (PASTEND) (OVERLAP * *NOW))
   (TO (PAST (GETA BEFORE)) T))
(PASTEND (POP 'PAST T))

(FUTR ; If we can get to *NOW by AFTER arcs, it is future tense.
   (TO (FUTREND) (OVERLAP * *NOW))
   (TO (FUTR (GETA AFTER)) T))
(FUTREND (POP 'FUTR T))
```

Figure 16. ATN Grammar (continued).

PAST, if there is a path of BEFORE arcs from the node representing the ending time of the action to *NOW. If there isn't, the sub-network will eventually block, causing the CALL PAST arc in state PRED to fail. The FUTR sub-network works in a similar fashion. Similar sub-networks can easily be written to recognize the temporal structure of future perfect ("Lucy will have seen a saw."), which is a path of BEFORE arcs followed by a path of AFTER arcs from the ending time to now, and the temporal structures of other tenses.

In our example, the CALL PAST arc succeeds, TENSE is set to PAST, and the parser transfers TO state GENVB (Figure 15), where the appropriate verb group is generated and POPped to the CALL arc in state SVB. The verb group is constructed by VERBIZE, which is a LISP function that does morphological synthesis on verbs. Its arguments are the number, tense, voice, and verb to be used.

Back on the CALL arc in state SVB (Figure 15), the verb group POPped into the register REG is added to the STRING, which is now (I UNDERSTAND THAT YOUNG LUCY SAW), and the parser JUMPs to state SUROBJ. There, the NP sub-network (Figure 13) is CALLed to generate a noun phrase for the contents of OBJ (M11). Since M11 has neither a LEX arc nor a name, but does have a class, the third arc is used, and (A SAW) is generated and POPped. This noun phrase is added to STRING preceded by the contents of PRED, which is empty in sentences (1), (2), and (3), but which contains BY in sentence (4). The CALL arc then transfers TO state END, emptying the input buffer at the top level. The POP arc at state END POPs the contents of STRING, which is finally printed by the system, and the interaction is complete.

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 that 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.

Acknowledgments

The help provided by the following people with the ATN parsing and generating interpreters and compiler, the morphological analyzer and synthesizer, and moving the software across several LISP dialects is greatly appreciated: Stan Kwasny, John Lowrance, Darrel Joy, Don McKay, Chuck Arnold, Ritch Fritzson, Gerard Donlon. Associate editor Michael McCord and the reviewers provided valuable comments on earlier drafts of this paper.

References

Bates, M. 1978 The theory and practice of augmented transition network grammars. In Bolc, L., Ed., *Natural Language Communication with Computers.* Springer Verlag, Berlin: 191-259.

Burton, R.R. 1976 Semantic grammar: an engineering technique for constructing natural language understanding systems. BBN Report No. 3453, Bolt Beranek and Newman, Inc., Cambridge, MA., December.

Burton, R.R. and Woods, W.A. 1976 A compiling system for augmented transition networks. Preprints of COLING 76: The International Conference on Computational Linguistics, Ottawa (June).

Shapiro, S.C. 1975 Generation as parsing from a network into a linear string. *AJCL* Microfiche 33, 45-62.

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

Simmons, R.F. 1973 Semantic networks: their computation and use for understanding English sentences. In Schank, R.C. and Colby, K.M., Ed., *Computer Models of Thought and Language.* W. H. Freeman and Co., San Francisco: 63-113.

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

Woods, W.A. 1973 An experimental parsing system for transition network grammars. In Rustin, R., Ed., *Natural Language Processing.* Algorithmics Press, New York: 111-154.

Stuart C. Shapiro is an associate professor in the Department of Computer Science at the State University of New York at Buffalo. He received the Ph.D. degree in computer sciences from the University of Wisconsin in 1971.