The Planes Interpreter and Compiler for Augmented Transition Network Grammars [1]

Timothy Wilking Finin [2]

CONTENTS

1.0	Introduction	3
1.1	The Planes Natural Language Query System	3
1.2	An Overview of this Chapter	5
1.3	Conventions	5
2.0	Augmented Transition Networks	6
2.1	Introduction	6
2.2	The ATN Formalism	7
2.3	Design Considerations	9
2.4	Implementation	10
3.0	The ATN Interpreter	11
3.1	Invoking the ATN Interpreter	11
3.2	Arc Types	12
3.2.1	Arcs which Test the Current Word	13
3.2.2	Arcs which Modify the Flow of Control	15
3.2.3	The Virtual Arc	18
3.2.4	Miscellaneous Arcs	18
3.3	Defining ATN Networks and States	19
3.4	Functions for Manipulating Registers	21
3.4.1	Setting Registers	22
3.4.2	Accessing Registers	23
3.4.3	Testing the Contents of a Register	24
3.5	Flow of Control	24

---

[1] This work was completed while the author was at the Coordinated Science Laboratory of the University of Illinois in Urbana-Champaign.

[2] Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19143 USA

L. Bloc (ed.), *The Design of Interpreters, Compilers, and Editors for Augmented Transition Networks*
© Springer-Verlag Berlin Heidelberg 1983

3.5.1 Effecting State Transitions 24
3.5.2 Invoking and Returning from Sub-computations 25
3.5.3 Controlling Backtracking and Failure 25
3.5.4 Suspending and Resuming Computations 26
3.5.5 The HOLD/VIRTUAL Facility 27
3.6 Useful Actions 30
3.6.1 Actions which Test the Current Word 30
3.6.2 Actions which Test Arbitrary Words 32
3.6.3 Miscellaneous Actions 32
3.7 Tracing Facilities 33
3.8 Global Variables 35

4.0 The ATN Compiler 37
4.1 Introduction 37
4.1.1 Compilation of a State 38
4.1.2 Compilation of an Individual Arc 40
4.1.3 Optimization 42
4.2 Using the ATN Compiler 44
4.2.1 Compiling Networks into LISP Code 47
4.2.2 Compilation in the LISP Compiler 47
4.2.3 Switches 48

5.0 The Lexicon 49
5.1 Introduction 49
5.2 The Dictionary Format 49
5.3 The Dictionary Manager 51
5.4 Auxiliary Functions 53

6.0 Auxiliary Functions 56
6.1 Interfacing with the LISP Editor 56
6.2 Displaying ATN Networks and States 57

7.0 Examples 58
7.1 A Sample ATN Network 58
7.2 A Sample Dictionary 60
7.3 Examples of Operation 61
7.4 Examples of Tracing States 63
7.5 Examples of Tracing Registers 65

8.0 References 67

1.0 INTRODUCTION

1.1 The PLANES Natural Language Query System

This chapter discusses the design and implementation of the ATN interpreter and compiler used in the PLANES system ([Walt75], [Walt76] and [Walt78]). This system, developed at the University of Illinois in the mid 70's, is an interactive question answering system which answers questions concerning a large data base of aircraft maintenance information. The intended scope of the PLANES system is to answer such questions as:

> How many Skyhawks required engine repairs in 1973?
> Did any of these log more than 200 hours in March?
> Show me the tail numbers of planes which were NOR during May.
> What kinds of aircraft are in the data base?

The processing of a user's request is divided into three main phases: parsing, interpretation, and evaluation (see figure 1 - The PLANES System).

The first phase, the parsing, is driven by a large ATN grammar. This ATN grammar defines a "semantic grammar" [Brow76] which maps the users request into an internal representation, the Paraphrase Language. The grammar is "semantic" in that it incorporates semantic and pragmatic knowledge as well as syntactic.

The Paraphrase Language representation of the user's request is translated by the second stage into a 'program' to generate the data to answer the request. This stage contains detailed knowledge about the data bases. It must know, for example, which sub-parts of the data base contain certain relations and the particular codes used to represent certain facts. The program is constructed out of primitives provided by the Query Language.

Finally, the Query Language evaluates the 'program' and passes the resulting data to the Response Generator. This module can display the answer in one of three forms: as a simple number or list, as a graph, or as a table. The choice of output form can be determined by the user thru a direct request (e.g. "Draw a graph of ...") or by a set of heuristics which attempt to find the most "natural" form. A graph, for example, will be generated only if the data consists of a set of tuples which

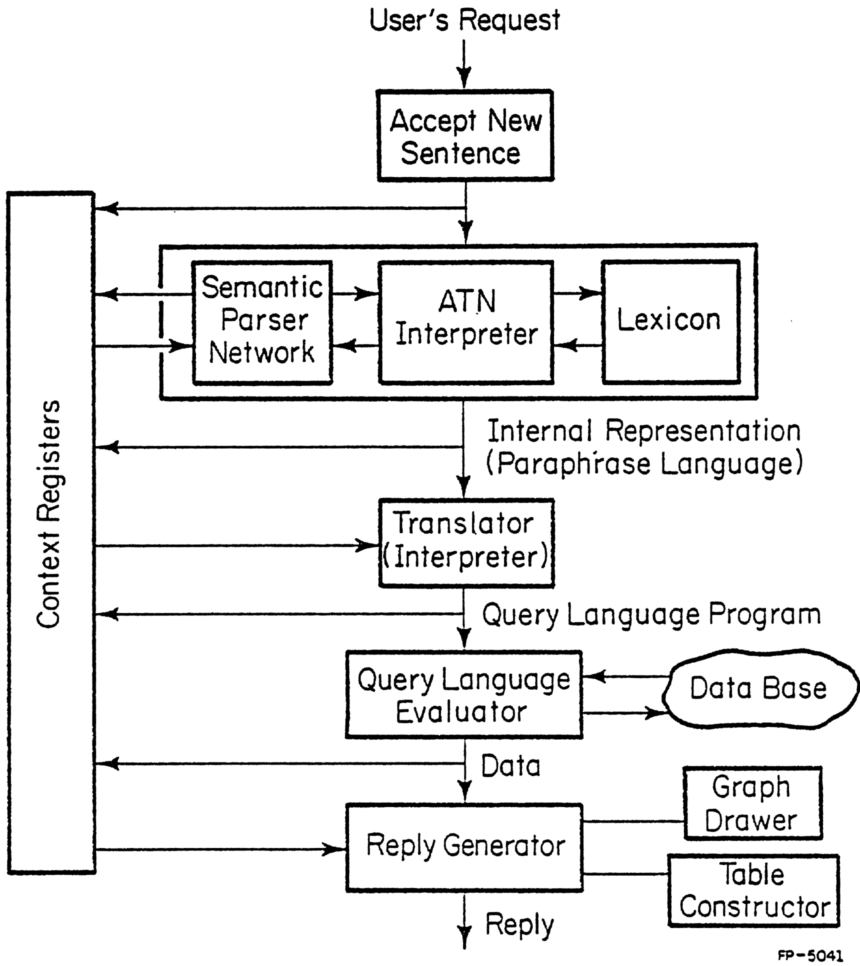


Fig.: 1 The PLANES System

can be interpreted as a function of two variables. Furthermore, the number of tuples must lie within certain bounds.

At each stage of the process, the results are sent to the History Keeper which manages a set of stacks of relevant information. These stacks contain the results of each stage (e.g. users request, paraphrase, etc.), syntactic components (e.g. subject, object, etc.), and semantic/contextual information (e.g. time specifications, plane specifications, etc.). This information is made available for resolving anaphoric reference, supplying phrases deleted thru ellipsis, and generating responses.

The entire process can be aborted by any of the major stages. If this is done, a suitable error message is generated and the user is invited to reform his request.

1.2 An Overview of this Chapter

This chapter describes the design and use of the PLANES ATN system. An earlier version ([Fini77a]) in fact, also served as a user's manual. Section 2 gives a brief description of ATN's and discusses some of the high level design considerations. Section 3 describes the interpreter and the auxiliary functions available to the user in some detail. Section 4 presents the compiler which can translate ATN networks into LISP code or machine language instructions. Section 5 describes the dictionary format expected by the interpreter. Also discussed are the various functions provided for creating and maintaining dictionaries. Section 6 documents several packages of auxiliary functions provided for interfacing the ATN system with the LISP editor and Pretty-printer. The chapter closes with examples of a very simple English grammar, a dictionary, and traces of the application of this grammar and dictionary to parsing some simple sentences.

1.3 Conventions

This chapter documents many LISP functions and "function-like" constructions. As part of the description for a function, a "syntax frame" will be given which specifies the number of arguments and whether or not they are evaluated. A "syntax frame" will look like a typical call to the function, i.e. it will be a list whose first element is the name of the function and whose remaining elements are the arguments. Arguments in the "syntax frame" will typically be enclosed in angle-brackets (<,>).

For example, the syntax frame for the LISP function LESSP might be:

```
(LESSP <number> <number>)
```

which states that LESSP takes two arguments, both of which are evaluated.

If a quote-sign (') precedes an argument, then that argument is not evaluated by the function. For example, the following describes the function DEFPROP:

```
(DEFPROP '<atom> '<value> '<property>)
```

If a function takes one or more optional arguments, then several "syntax frames" will be given. For example, the MacLISP function CATCH would be described as:

```
(CATCH <expression> '<tag>)
(CATCH <expression>)
```

Two conventions will be used to exhibit functions which take an indefinite number of arguments. In one, the arguments are simply partially enumerated with the elliptical "...", as in:

```
(PLUS <number1> <number2> ... <number n>)
```

The second form has the last argument to the function preceded by the "dotted pair" dot, as in:

```
(DEFUN '<name> '<argument list> . '<function body>)
```

The last argument then stands for an indefinite number of final arguments. Example calls to a function will usually be given with the "syntax frame" to clarify its usage. These will be preceded by an "eg:" indicator.

2.0 AUGMENTED TRANSITION NETWORKS

2.1 Introduction

The ATN formalism has been well received by the computational linguistics community in the past decade. A number of people were working with similar models in the late 60's (see [Thor68] and [Bobr69]), but the idea was crystallized and popularized by the work of William Woods in the early 70's [Wood71]. The ATN model has been heavily used to represent grammars for question answering systems, speech processing systems,

language generation systems, modeling learning and even for processing visual information [Loza77]. This section will provide a brief introduction to the ATN formalism. An excellent introduction to the theory and use of ATN grammars can be found in [Bate79].

2.2 The ATN Formalism

The ATN formalism is usually described in a evolutionary manner, showing how one can start with the notion of a simple finite state machine (FSM) and modify and embellish it to arrive at an Augmented Finite State Transition Network (ASFTN).

The FSM is the basis for ATN's

We start with a garden variety deterministic FSM consisting of a set of labeled states, a set of arcs between states, an initial state, and a set of distinguished final or accepting states. The first step might be to extend the formalism to allow for non-determinism. A simple grammar for English might be represented by the graph in figure 2 (A Simple Grammar for English). As a model for natural language processing this is inadequate. The most serious deficit is the lack of any mechanism to handle embedding of constituents to arbitrary depth. The next extension, then, is to provide a "push down" mechanism whereby one can suspend the processing of a constituent at one level while using the same network to process an embedded one.

Adding recursion extends the power of an FSM

We can represent this by labeling an arc from one state to another with the name of a third state. When such an arc is encountered, processing is suspended and the network restarted in the specified state. If an accepting state is eventually reached, the suspended processing is resumed. Such a capability not only allows one to achieve the power to parse a context free language, but also simplifies the representation of a grammar. One can collect common network fragments and specify them in one place, much as one can substitute a subroutine call for common program sequences in a programming language. Our simplistic English grammar might now be represented by the graph in figure 3 (A Better Grammar for English). Note that this grammar now accepts conjoined sentences and sentences embedded as relative clauses.

Augmentation realizes the power of ATN's

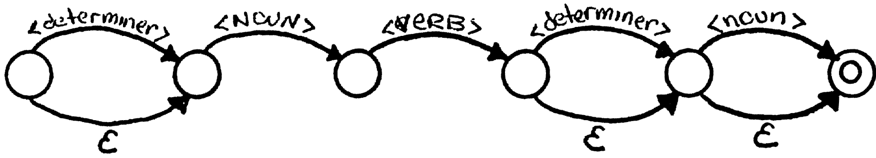
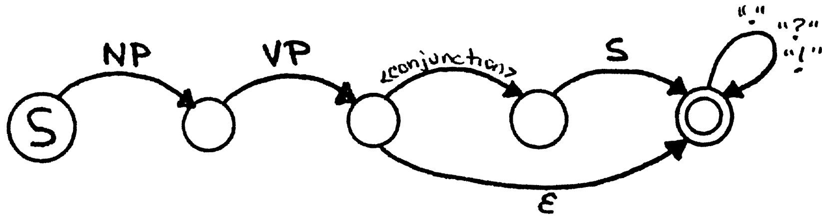
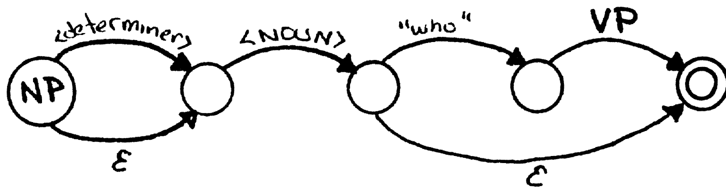


Fig. 2: A Simple Grammar for English





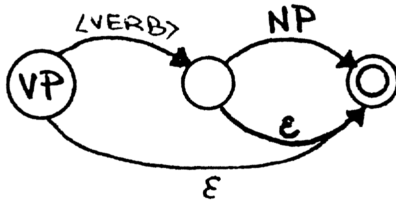


Fig. 3: A Better Grammar for English

Whether or not recursive transition networks provide a theoretically adaquate mechanism for parsing natural languages is an open question. It is clear, however, that from a practical viewpoint, the RTN formalism is not as powerful as one would like it to be. To further extend the power of our formalism we will allow arcs to be augmented with conditions and actions. Each arc can be augmented with:

1. An arbitrary condition which must be met before the arc may be taken.
2. A set of arbitrary actions which will be performed if the arc is taken. These can have side effects thru the setting of registers whose contents can later be accessed in other parts of the grammar.

This augmentation (among others) gives an ATN the computational power of a Turing machine. The final result is a powerful formalism which retains much of the elegance of the simple FSM.

2.3 Design Considerations

Since ATN's are based on non-deterministic FSM's, an implementation must be able to simulate all possible paths thru the network. We have chosen to implement a depth-first backtracking version for several reasons. Firstly, it is our belief that natural language processing may be most suited to this approach. At least, evidence suggests that our only example of competent natural language processors (i.e. humans) do not use highly parallel methods. Secondly, we believe that whatever gains might be made thru a parallel approach would be more than offset by the additional cost of processing. The phenomenon of natural language is such that it encompasses an extraordinarily large number of patterns and constructions. Most of these, however, are relatively rare, the greatest part of most language being drawn from a few common constructions. A non-parallel approach can easily take advantage of this fact in the ordering of its attempts to find a path thru the ATN network.

To simulate the non-deterministic nature of an ATN processor (or a FSM for that matter) one needs to maintain an encoding of the current configuration (i.e. state, input word, register environment, etc.) as well as a stack of past configurations which represent the path so far. In our ATN system, we have chosen a recursive implementation in which this information is stored on LISP's internal stack. The alternative is to

maintain an explicit configuration stack built from CONS cells from the general pool. The advantages of a recursive implementation include:

- o Simplicity of design

  Backing up to a previous configuration is achieved by "returning" to the environment of that configuration. The undoing of side-effects is automatic.

- o Efficiency

  Previous configurations are stored on LISP's internal stack and thus do not consume CONS cells from the general pool. This reduces the amount of LIST space required and, more importantly, results in fewer and shorter garbage collections.

The chief disadvantage of a recursive implementation is that we give up the possibility of giving the user complete control over processing. Earlier configurations are inaccessible to the user as well as to the system itself (*). This problem is alleviated by the inclusion of several flow-of-control modifiers thru which one can directly fail back to a specified configuration.

2.4 Implementation

The PLANES ATN interpreter, Dictionary Manager, and ATN compiler are written in the LISP dialect MacLisp, which runs on a variety of machines and operating systems (**). PDP-10 TOPS10 timesharing system. The following gives approximate values for memory requirements:

	LIST space	BPS space
Interpreter	5100	4000
Dictionary Manager	1200	1000
Compiler	2500	1500

In this table, LIST space refers to the number of memory cells used to

---

(*) This makes certain kinds of parallel processing regimes difficult, if not impossible, to implement.

(**) Versions of MacLisp exist for PDP-10s under the TOPS-aO, TOPS-20, ITS, and WAIT operating systems and for MULTICS. Two other Lisp dialects are sufficiently close to MacLisp to allow the PLANES ATN system to run with minor modifications: FranzLisp (for a PDP11/780 Vax) and Lisp Machine Lisp.

store lists, atoms, etc. and BPS space refers to 'Binary Program Space', the number of memory cells used to store compiled code.

3.0 THE ATN INTERPRETER

This section describes the principal functions in the PLANES ATN interpreter.

3.1 Invoking the ATN Interpreter

PARSE and PARSE1 invoke the interpreter

PARSE and PARSE1 are the main functions used to invoke the ATN interpreter on an input string. The function PARSE is intended to be used at LISP's toplevel and PARSE1 used by other programs.

(PARSE '<string> '<initial state> '<number of parses>)

eg: (PARSE (which skyhawks required repairs) s:start 3)

The function PARSE starts the ATN interpreter in state <initial state> with the input string <string>. It takes from 0 to 3 arguments, none of which is evaluated. If any argument is omitted or is NIL, it is defaulted as specified below.

The first argument specifies the input string to be parsed. If it is a list of atoms, then that list is used as the input string. If it is an atom which has a value, then the value of the atom is used as the input string. If it is omitted or is NIL, then the function READTEXT is called which reads the input string from the terminal. The second argument to PARSE specifies the initial state of an ATN network. If it is omitted or NIL, it defaults to the value of the global variable @INITIALSTATE. The third argument determines how many parses are produced. If it is omitted or NIL, it defaults to one. If it is a positive number, then the interpreter will attempt to produce that many parses. If it is the atom ALL, then all possible parses will be produced.

The function PARSE always returns the value "*", so it is run for its side-effects. The principle side-effect is that it sets the variable PARSE to:

- The atom FAIL if the ATN was unable to reach a final state.

- o The value POPped by the ATN if <number of parses> is 1.
- o A list of the values POPped if <number of parses> is greater than one or the atom ALL.

In addition, this value is "pretty printed" on the terminal and the atom AGAIN is set to the input string.

(PARSE1 <string> <initial state> <number of parses>)

eg: (PARSE1 '(show me the list) 's:start 'all)

The function PARSE1 requires all three arguments, each of which is evaluated. It returns the value(s) POPped by the ATN from the final state(s). PARSE1 has the side-effect of binding the global variable AGAIN to <input string>.

(READTEXT)

If the first argument to PARSE is omitted or NIL, the function READTEXT is called to read the input string from the TTY. This function reads a string of characters from the TTY until a carriage-return is typed and returns it in the form of a list of atoms. Characters which have a "special" meaning to LISP (e.g. . , ; , etc.) are automatically "slashified" (i.e. taken literally).

The function incorporates three special features:

1. If the null string is entered by typing only a carriage-return, then READTEXT returns the last input string given to PARSE (i.e. the value of AGAIN).
2. If the first character on the line is a "(" then READTEXT reads a LISP s-expression, evaluates it, and prints the resulting value. It then waits for a string of characters to be typed as the input string.
3. Whenever a line-feed is typed, READTEXT takes the next character of the input string from the last input string (AGAIN again). Subsequent line-feeds produce subsequent characters from the old input string.

3.2 Arc Types

The PLANES ATN interpreter provides a total of 13 varieties of arcs.

These can be broken down into the following categories:

1. Flow of Control (PUSH, POP, TO, JUMP, FAIL)

   These are arcs which effect transitions from one state to another or invoke or return from sub-computations.

2. Testing the Current Word (CAT, WRD, ROOT, PHRASE)

   These arcs are taken only when the current word meets a specified condition.

3. Virtual Arc (VIR)

   This arc provides a natural mechanism for handling constituents which are parsed "out of place" in the input string.

4. Miscellaneous Arcs (TST, DO, AND)

   These three arcs provide a mechanism for miscellaneous computation.

All of the arcs except the AND arc have the basic form:

```
    (<type> <head> <test> <action1> ... <actionn>)
```

where <type> is one of the arc types (PUSH, CAT, TST, etc.), <head> is an argument for that arc type, <test> is an arbitrary condition which must be met before the arc can be taken, and the <actions>'s are arbitrary lisp expressions which are evaluated if the arc is taken. Typically the last action will specify destination - the next state to go to.

3.2.1 Arcs which Test the Current Word

<u>The WRD arc compares the current word</u>

```
(WRD '<word(s)> <test> ... <actions> ... <destination>)
eg: (WRD (the a) T (SETR det *)(TO np:det))
    (WRD please t (SETR polite t)(TO s:start))
```

The WRD arc compares the current word against <word(s)>, which can be an atom or a list of atoms. If the current word matches, then <test> is evaluated and, if true, the actions are evaluated from left to right.

<u>The CAT arc checks for a lexical category</u>

```
(CAT '<head> <test> ... <actions> ... <destination>)
eg: (CAT adj T (ADDR adj *)(TO np:adj))
    (CAT (n npr) t (SETR n *)(TO np:n))
```

A CAT arc is taken if the current word has a lexical entry under the category <head> (if it is an atom) or any of the categories <head> if a list. Within the CAT arc, the variable * is bound to the root form of the current word. For example, if the current word is CRASHING, then the following arc would set the register VERB to CRASH:

```
    (CAT V T (SETR verb *)(TO s:verb))
```

Note that this arc may generate several possibilities if the current word has more than one dictionary entry under the specified category. For example, the word SAW might have the dictionary entry:

```
    (SAW V  (SEE (tns past))
            (SAW (tns present)(untensed)))
```

In this case the CAT arc above will first try parsing SAW as the past tense of SEE and, if that fails, will try parsing SAW as the present/ untensed form of the verb SAW.

<u>The ROOT arc checks the root form</u>

```
(ROOT '<root(s)> <test> ... <actions> ... <destination>)
eg: (ROOT BE T (SETR v *)(to S:BE))
    (ROOT (have be) (NULLR aux)(SETR aux *)(TO s:aux))
```

A ROOT arc compares the root form of the current word to <root(s)>. Again, <root(s)> can be an atom or a list of atoms. The arc is taken if the current word matches one of the forms in <root(s)> . As with the CAT arc, the variable * is bound to the root form of the current word within the ROOT arc.

<u>The PHRASE arc checks the next several words</u>

```
(PHRASE '<phrase> <test> ... <actions> ... <destination>)
eg: (PHRASE (more or less) T (SETR fuzzy T)(TO quant:end))
    (PHRASE (please (give show tell) (me {})) T (TO s:imperative))
```

The PHRASE arc is used to compare the next several words in the input string to <phrase>, which should be a list. Successive elements of <phrase> are matched against successive words from the input string.

The possibilities for phrase elements, and what they represent are given below:

- o A simple word (e.g. MORE)

  The given word must match the corresponding word in the input stream exactly.

- o A root form (e.g. {eat})

  A word between "{" and "}" specifies a root form match. The root form of the corresponding word in the input must match the given root. If the input word has several different root forms (corresponding to several lexical categories) then the given root may match any one of them.

- o A lexical category (e.g. {.prep})

  A word between "{." and "}" indicates a lexical category to compare the next input word to. If the input word is a member of the specified category, then the phrase match continues.

- o A list of Phrase elements (e.g. ({give} {show} {tell}))

  An element which is a list enumerates several possibilities for the corresponding word in the input string. If any one of them matches the next input word, then the match continues.

The phrase arc is taken if and only if each element of <phrase> is successfully matched against the input string.

3.2.2 Arcs which Modify the Flow of Control

These arc types can be further divided into arcs which involve sub-computations (PUSH, POP, POP!), arcs effecting state transitions (TO, JUMP) and an arc which controls backtracking (FAIL).

<u>The PUSH arc invokes a sub-computation</u>

```
(Push '<state> <test> ... <actions> ... <destination>)
eg:  (PUSH np t (SENDR goal 'subj)(SETR subj *)(TO s:subj))
```

The PUSH arc is used to invoke a sub-computation. If the expression <test> is true, then the ATN interpreter is recursively called, starting in state <state>. All registers at the current level are saved and made invisible to the lower level. Before actually invoking the lower level, the list of actions is scanned for "preactions", i.e. those which

should be evaluated before the PUSH is made. These "preactions" are SENDR's or any action which begins with a "!" (e.g. (! (OR (EQ Sv 'be) (SENDR v))). The SENDR actions, and any previously done, are typically used to set register values in the lower computation.

If the sub-computation fails, i.e. does not reach a state which can take a POP arc, then the PUSH fails. If a POP from the lower level is taken, then control is returned to the PUSH arc. The variable * is then set to the value POPped by the lower network, the values of any LIFTRed registers set, and the "post-actions" evaluated. These "post-actions" are just the actions on the arc minus the "pre-actions".

The POP arc returns from a sub-computation

(POP <value> <test> . <actions>)

The POP arc returns from a sub-computation. If <test> is true, the <actions> are evaluated from left to right and then <value> is computed and returned to the most recent PUSH arc or PUSHATN action. Both <test> and <actions> can be omitted. If the <test> argument is omitted, it defaults to T.

Two conditions affect the taking of a POP arc. If any items were on the HOLD list at this level and have not yet been used, then the POP arc can not be taken (see section 2.5.5). In addition, a POP arc will not be taken if it would be returning to the "top-level" and words remain to be parsed (i.e. STRING is non-NIL). This last condition can be turned off if the global variable USE-ALL-WORDS? is set to NIL.

Some examples of POP arcs are:

```
(POP (build-np) T (LIFTR head-noun $noun))
(POP (BUILDQ (pp + +) prep np))
```

The POP! arc really returns

(POP! <value> <test> . <actions>)

The POP! arc is almost identical to the POP arc except for one important difference: one can not back-up into the sub-computation after leaving it via a POP! arc. Under our implementation of the ATN interpreter, a POP! arc does a RETURN to the environment of the most recent PUSH arc or call to the PUSHATN action. If failure later backs up to this point, the entire sub-computation will, in effect, be backed up over as well.

The grammar writer may find this a useful arc in that it gives him greater control over the automatic backtracking done by the ATN interpreter.

The TO and JUMP arcs effect state transitions

```
(TO '<state> <test> . <actions>)
(JUMP '<state> <test> . <actions>)
```

As a convenience, special arcs are provided which duplicate the TO and JUMP actions. Both arc types cause the ATN interpreter to advance to the state specified by their first "argument". The TO arc causes the current word to be advanced and the JUMP arc does not. In each case, the <test> argument and the <actions> arguments are optional. If the <test> is omitted, it defaults to T. Some examples are:

```
(JUMP s:end (out-of-words?))
(JUMP np:det)
(TO np:det t (ADDR ignored-words *))
```

The FAIL arc controls backtracking

```
(FAIL '<where> <test> . <actions>)
```

The FAIL arc can be used to gain some control over backtracking. If the <test> is true, then this arc will evaluate the <actions> and cause failure to propagate backwards to a point specified by <where>. Again, the <test> and <actions> arguments may be omitted.

The <where> argument may specify that the ATN interpreter fail from the current arc, the current state, the current sub-computation, an arbitrary named state, or to the initial top-level call. The precise options for the <where> argument are:

```
STATE   : fail from the current state
PUSH    : fail from the current subcomputation
          (i.e. from the most recent PUSH)
<state> : fail from the named state
TOP     : fail altogether
```

Note that the setting of registers in the <actions> will have no effect since their side effects will be immediately undone by the failure. Some examples of FAIL arcs are:

```
(FAIL state)
(FAIL np:adj (CAT v))
(FAIL push (NOT (CAT prep)))
```

3.2.3 The Virtual Arc

```
(VIR '<cat> <test> ... <actions> ... <destination>)
(VIR '(<cat> <level>) <test> ... <actions> ... <destination>)
```

The VIR or virtual arc is taken if a constituent is found on the HOLD list indexed under the category <cat>. Again, the <test> argument must evaluate to non-NIL.

Note that two syntax frames are given for this arc. In the first, the "head" of a VIR arc is an atomic category name. In the second, it is a tuple whoes first element is a category name and whose second is a number which specifies the level at which the constituent is to be found on the HOLD list. See section 3.5.5 for details.

If the VIR arc can be taken, * is bound to the constituent found on the HOLD list and FEATURES is bound to the (optional) feature list associated with that constituent.

3.2.4 Miscellaneous Arcs

The TST arc applies an arbitrary test

```
(TST '<label> <test> ... <actions> ... <destination>)
```

The TST arc is taken if the arbitrary LISP expression <test> evaluates to non-NIL. The <label> argument is not evaluated and is not used in any way by the ATN interpreter. Typically it is used for a mnemonic label describing the function of the arc. Some examples are:

```
(TST end-of-sent? (AND (NULL *)(NULL string))(JUMP s:end))
(TST t (MEMQ   cuss-words)(TO s:complain))
```

The DO arc has no destination

```
(DO '<label> <test> . <actions>)
```

The ATN interpreter handles the DO arc in an identical manner to the TST arc. If the <test> is true, the <actions> are evaluated. Its intended use, however, is somewhat different. The TST arc is intended to have a "destination action" as its final action (i.e. a call to TO or JUMP). The DO arc is intended to be used without a "destination action". Thus, the DO arc might be used to initialize a set of registers. Some sample DO arcs are given below:

```
(DO initialize t (SETR type 'dcl)(SETR context NIL)
(DO warn verbose? (print '|That is not very grammatical!|))
```

The AND arc conjoins other arcs

```
(AND <arc1> <arc2> ... <arcn>)
```

The AND arc is the only one which does not fit the general arc syntax. The AND arc takes an indefinite number of arguments, each of which can be any of the legal arc types (including another AND arc). The effect of an AND arc is to evaluate each of the "sub-arcs" from left to right until one of them fails or the last one is reached. Consequently, none of the "sub-arcs" should contain a "destination action" except the last one, which should have a "destination" as its last action.

Note that after evaluating each of the "sub-arcs", the current word is always advanced. This arc is useful in eliminating many states which contain only a single arc. This can greatly increase the readability of a network by keeping together arcs which form a single path. For example, to handle two-word comparative adjective, one might use the following arc:

```
(AND (WRD more t)
     (CAT adj t (SETR adj (BUILDQ (adj: * comparative)))
                (TO np:adj)))
```

3.3 Defining ATN Networks and States

The PLANES ATN system provides functions for defining ATN networks, states, individual arcs and special word-triggered interrupt functions.

DEFATN defines an ATN network

```
(DEFATN '<name> '<default-arcs> '<default-registers>
        '(<state> ... <state>))
```

The function DEFATN takes from 2 to 4 arguments and defines an ATN network. The first argument is the name of the ATN network and the last argument is a list of the states in the network. Intervening arguments are optional and, if included, can specify initial values for registers and special "default arcs" which are assumed to be the initial arcs of each state in the network.

If a register default argument is supplied, its form should be:

```
("default-register" (<register name> <value>)
                    (<register name> <value>)
                    ... ))
```

This argument causes the named registers to be set to the specified initial values whenever a state in the network is PUSHed to.

If a "default-arc" argument is given to DEFATN, its form should be:

```
("default-arc"  <arc1> <arc2> ... <arcn>)
```

where an arc has one of the forms:

```
<arc name>
(<arc type> <head>.... )
```

If the arc is an atom, it is assumed to be the name of separately defined arc (see DEFARC). If it is a list, then it should be in the form of a legally defined arc. The effect of this argument is to cause the specified arcs to be added at the beginning of each state in the network.

DEFSTATE defines an individual state

```
(DEFSTATE '<name> . '<arcs>)
```

Individual ATN states can be defined with the function DEFSTATE. This function takes an indefinite number of arguments, the first of which is the name of the state. The remaining arguments specify the arcs leading from the state. Again, an arc can be represented either by an atomic name or by a complete list. An example is:

```
(DEFSTATE np:start
     (CAT det (SETR det *)(TO np:det))
     (CAT pro (SETR noun *)(TO np:end))
     (JUMP np:det))
```

DEFARC defines an individual arc

```
(DEFARC '<name> '<arc>)
```

An individual arc can be defined with the function DEFARC. Arcs defined in this way can be of two types: Regular arcs which have a destination

(i.e. a call to TO or JUMP) as their last action and degenerate arcs which do not have a destination as their last action. When a degenerate arc is inserted into an ATN state, a destination is supplied which causes the ATN to loop back to that state. For example:

```
(DEFARC timephrase (PUSH timepp T (SETR time *)))
```

is an arc which will attempt to parse a time phrase and then reenter the current state.

Word interrupts are created by DEFINTERUPT

The PLANES ATN system provides a mechanism whereby certain computations can be performed whenever a certain word is encountered in the input string. The function DEFINTERUPT associates an interrupt function with a word.

```
(DEFINTERUPT '<word(s)> '<function>)
 eg: (DEFINTERUPT please cdr)
     (DEFINTERUPT (in on before)
                  (lambda (x)(parse-time-pp (car x)(cdr x))))
```

The first argument to DEFINTERUPT should be a word or a list of words. The second should be a function of one argument, expressed as either the atomic name of the function or as a lambda expression. Whenever one of the specified words is encountered in the input string, the associated interrupt function is called with the input string as its argument. Its return value is used as the new input string. Thus the first example would cause the word PLEASE to be ignored (since it returns the CDR of the input string). The second example would call the function PARSE-TIME-PP whenever the words IN, ON, or BEFORE are discovered in the input string.

3.4 Functions for Manipulating Registers

A register in an ATN grammar plays the role of a variable in a higher level programming system. The interpreter provides functions for assigning a register a value (SETR, SENDR, LIFTR), accessing the value of a register (GETR,$), and testing the contents of a register (NULLR). Registers need not be declared and are "created" when they are assigned a value for the first time. The contents of a register which has never been assigned a value is defined to be NIL.

3.4.1 Setting Registers

Registers can be set at the current level (SETR) in the next lower level (SENDR) or in any of the higher levels (LIFTR).

SETR sets a register at the current level

(SETR '<register name> <value>)
 eg: (SETR verb 'be)

The function SETR assigns a value to a register at the current level. SETR assigns the register <register name> the value <value>.

SENDR sets a register at a lower level

(SENDR '<register name> <value>)
(SENDR '<register name>)

SENDR is used to set the contents of a register at a lower level, typically just before a PUSH action. Whenever a SENDR is evaluated, the register name and value are stored on the SENDLIST. When a PUSH occurs, the SENDLIST is used to initialize the registers to the specified values. If the second argument is omitted, it defaults to the current contents of the register named by the first argument. Thus (SENDR noun) is the equivalent to (SENDR noun (GETR noun)).

LIFTR sets a register at a higher level

The function LIFTR is used to set the contents of a register at a higher level. It takes from one to three arguments, the syntax being:

(LIFTR '<register name> <value> <level>)
(LIFTR '<register name> <value>)
(LIFTR '<register name>)

If all three arguments are given, LIFTR assigns <register name> the value <value> <level> levels above the current one. For example, (LIFTR stype 'q 1) would set the register STYPE to Q in the next higher level. If the <level> argument is the atom TOP, then it refers to the highest level. If the <level> argument is omitted, it defaults to 1 (i.e. the next higher level).

The second argument, <value>, specifies the value to be assigned to the register. If it is omitted, the current contents of the register are used. Thus (LIFTR noun) is equivalent to (LIFTR noun (GETR noun)).

When a LIFTR action is evaluated, it adds the complete three tuple (<register name><value><level>) to the special register LIFTLIST. When a POP action is taken, the LIFTLIST is processed by setting those registers associated with a <level> of 1 and "re-lifting" the rest of the elements with a decremented <level>. Thus, the effect of a LIFTR is only seen once we have POPped to the appropriate level.

3.4.2 Accessing Registers

GETR gets the contents of a register

The function GETR is used to access the contents of a register at the current level or, if an optional argument is supplied, at any higher level. The syntax is:

(GETR '<register name>)
(GETR '<register name> <level>)
(GETR '<register name> 'nearest <test>)

If only one argument is supplied, GETR retrieves the contents of that register at the current level. An optional second argument instructs GETR to get the contents of the register at a higher level. In addition, if <level> evaluates to the atom NEAREST, then GETR will return the contents of the register at the lowest level in which it has a value. If <level> is NEAREST, an optional third argument, <test>, can specify an arbitrary test which must be met by the candidate register value. While evaluating <test>, the variable * is bound to the contents of the candidate register. For example, the expression:

(GETR verb 'nearest '(TRANSITIVE *))

would search upwards thru the chain of levels until it found a VERB register which contained a transitive verb (assuming that TRANSITIVE is a user-defined predicate which is true of transitive verbs only).

The character $ acts as prefix operator

The character $ (i.e. dollar sign) is defined to be a "read macro character" which acts as a prefix operator identical to GETR. Thus $noun is the equivalent to the function call (GETR noun). This feature can be turned on or off with the function DOLLARMACRO. Evaluating (DOLLARMACRO NIL) will turn it off causing the dollar sign to lose its special significance. Similarly, (DOLLARMACRO T) will turn the feature back on.

3.4.3 Testing the Contents of a Register

NULLR tests the value of a register

Since the value of a register which has never been set is defined to be NIL, there is no way to tell whether or not a register has been set to the value NIL or has never been set. To provide the user with this capability, the function NULLR will return T if and only if a register has been explicitly set to NIL. The syntax is simply:

```
(NULLR '<register name>)
```

3.5 Flow of Control

Six function are provided which modify the flow of control. Five of them are analogous to arcs PUSH, POP, TO, JUMP and FAIL. The sixth provides a special facility by which one can save the state of a subcomputation just before returning via a POP and later resume it with the same register environment.

3.5.1 Effecting State Transitions

TO and JUMP advance the ATN to a new state

These are typically found as the last action at the end of an arc, as in:

```
(CAT v t (SETR v *)(TO s:verb))
(TST T (out-of-words)(JUMP s:end))
```

They can, however, be called at any point by the user and embedded in arbitrary LISP expressions. For example, the destination of the following arc is determined at run-time:

```
(CAT v t (setr v *) (cond ((transitive $v) (TO s:verb-trans))
                          ((TO s:verb-interans))))
```

```
(TO '<state name>)
 eg: (TO s:verb)
```

The TO function effects a transition to the named state and causes the current word to be advanced.

(JUMP '<state name>)
 eg: (JUMP s:end)

The JUMP function also causes the ATN interpreter to enter the specified state but does not advance the input word. The next state will find the current word unchanged.

3.5.2 Invoking and Returning from Sub-computations

PUSHATN simulates a PUSH

(PUSHATN <state>)
 eg: (PUSHATN np:start)

The effect of a call to the function PUSHATN is similar to that of a PUSH arc. It recursively applies the ATN network to the input string starting in state <state>. The value returned by this function is the value POPed by the sub-computation or the atom FAIL if the sub-computation does not reach a final state. Additional side-effects are:

- o The variable * is bound to the value POPped.
- o The variable LEX is bound to the new current word.
- o The variable STRING is bound to the rest of the input string.

POPATN simulates a POP

(POPATN <value>)
 eg: (POPATN (buildquestion))

The POPATN function is similar to the POP arc. It causes control to return to the most recent PUSH arc or PUSHATN function call. As with the POP arc, the POP will only succeed if nothing has been placed on the HOLD list at the current level.

3.5.3 Controlling Backtracking and Failure

FAIL controls backtracking

(FAIL <where>)

The FAIL function is almost identical to the FAIL arc. It causes the ATN interpreter to backtrack to a point specified by the argument <where>. The options for this argument are:

ARC : fail from the current arc
* : fail from the current state
PUSH : fail from current level (i.e. to most recent PUSH)
TOP : fail from entire network (i.e. give up completely)
<state> : fail back to the state named <state>

3.5.4 Suspending and Resuming Computations

Suppose that we wanted to parse a sentence like:

How many planes were there which required no repairs in April?

In this sentence, the relative clause "which required no repairs in April" has been extraposed to the right (or the head "how many planes" extraposed to the left, if you like). We would like to be able to parse these two sub-strings as if they occurred together, as is usually the case:

... planes which required no maintenance in April ...

Using RESUME and RESUMETAG, one can "remember" where one stopped parsing a noun phrase (e.g. in "How many planes") and, at some later point in the input string, resume parsing the same noun phrase in the same state and with the same register environment.

RESUMETAG saves a configuration

```
(RESUMETAG '<state name>)
(RESUMETAG)
```

The RESUMETAG function returns a "tag" which encodes a state name and a register environment. With this "tag", one can later resume computation in this state with the same register environment. If the <state name> argument is not given, the current state is used.

RESUME resumes a suspended computation

```
(RESUME <tag> . '<register names>)
 eg: (RESUME $nptag verb)
```

The RESUME function resumes the subcomputation which generated <tag> (i.e. with a call to RESUMETAG) in the current position in the input string. Optional arguments to RESUME specify registers whose contents should be "sent" to the lower computation. The value returned by RESUME is the value POPped by the lower level.

3.5.5 The HOLD/VIRTUAL Facility

The HOLD action together with the VIR arc provide a natural mechanism for handling constituents which are parsed "out of place" in a sentence. The HOLD action allows one to place a parsed constituent on the HOLD list. At some later time, the VIR arc can be used to retrieve the held object and process it as if it were found at that point in the input string.

An example will show the usefulness of this facility. English syntax allows for the fronting of the "unknown element" in a question (e.g. a wh-noun phrase such as "who" or "which man". One would probably like a grammar to locate the place from which the unknown element was fronted. One strategy, then, is to place the unknown constituent on the HOLD list (via the HOLD action) and then write the grammar such that it checks the HOLD list whenever it can't find a constituent in the input string.

For example, consider the following sentences:

> Who did John see?
> Who did John sit next to?
> Who did John believe that Harry saw?
> Who does John think the FBI wanted to arrest?

These sentences can be handled by placing the constituent generated by the "who" on the HOLD list indexed as a Noun Phrase. At each point in the grammar where a NP is sought as either the subject of a sentence, the object of a sentence, or the object of a preposition, we include a VIR arc which examines the HOLD list for a held NP.

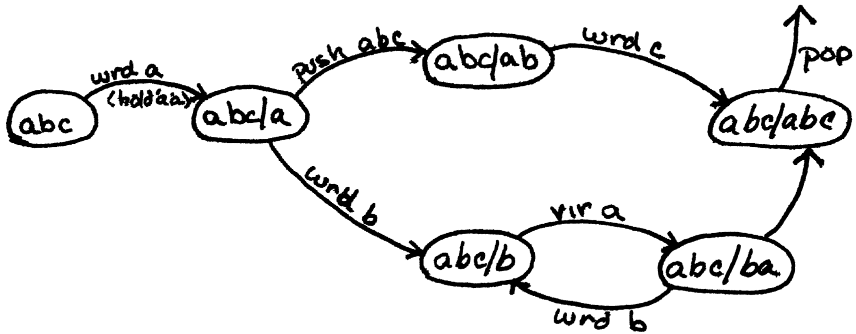
The addition of the HOLD/VIRTUAL concept to a RTN extends its power by providing, in essence, an additional stack. The ATN grammar in figure 4, for example, accepts the context-sensitive language

$$S \rightarrow a^n b^n c^n$$

<u>The HOLD action</u>

```
(HOLD <form> <cat> <features>)
 eg: (HOLD * 'np nil)
```

The HOLD action places the expression <form> on the HOLD list indexed under the category <cat>. An optional third argument, <features> is a




```
(defatn anbncn

        (abc (wrd a t (hold 'a 'a) (to abc/a)))

        (abc/a (wrd b t (to abc/b))
               (push abc t (to abc/ab)))

        (abc/ab (wrd c t (to abc/abc)))

        (abc/abc (pop t))

        (abc/b (vir a t (to abc/ba)))

        (abc/ba (wrd c t (to abc/abc))
                (wrd b t (to abc/b))))
```

Fig. 4: A Grammar for a^n b^n c^n

list of features to be associated with the entry. In addition, the level at which this action is being done is recorded with the entry. This allows a VIR arc to select only those constituents held at a particular level.

The VIRtual arc

The VIR arc allows the grammar to retrieve an element from the HOLD list. In retrieving the element, the user can specify:

- o The lexical category under which the element was held.
- o A limit on the number of levels above the current one at which the element was held.
- o An arbitrary test which the element or its features must pass.

The syntax frame for the VIR arc is:

```
(VIR <head> <test> ... <actions> ... <destination>)
```

where <head> can have either of the following two forms:

```
<cat>
(<cat> <level>)
```

The first form will match any item on the HOLD list indexed under the category <cat> regardless of the level at which it was held. In the second form, <level> specifies the level(s) at which a candidate item was held. This argument should be a number (0 or positive) which limits the number of levels above the current one at which the item was held. Thus:

```
(VIR (PP 0) ...)
```

will match items indexed as a PP which were held at the current level only. Similarly,

```
(VIR (ADV 2) ...)
```

would find ADV's held at the current level or the next two higher levels.

Once a candidate match has been found, the <test> argument is evaluated in an environment where * is bound to the candidate constituent and FEATURES is bound to the (optional) features list associated with it. If <test> is non-NIL, then the VIR arc is taken and the item is removed from the HOLD list.

3.6 Useful Actions

3.6.1 Actions which Test the Current Word

A number of functions are provided for testing the current word. These are typically used as further conditions on an arc, as is the CAT function used in the following arc:

```
(PUSH pp (CAT prep) ..... (TO s:end))
```

or as a condition in one of the actions in an arc:

```
(CAT det T (and (GETF question)(LIFTR type 'q)) ... )
```

WRD compares the current word

```
(WRD '<word(s)>)
 eg: (WRD who)
     (WRD (which that who))
```

The WRD function returns T if the current word (i.e. *) matches its single argument. If this argument is an atom then the current word must be identical to it. If it is a list of atoms, then the current word must be a member of this list.

CAT checks the lexical category of the current word

```
(CAT '<category>)
 eg: (CAT pro)
     (CAT (n pro npr))
```

The CAT function tests the current word to see if it can be a member of the lexical category <category>. As with the WRD function, the argument can be an atomic category name or a list of atomic category names.

ROOT checks the root form of the current word

```
(ROOT '<root(s)>)
(ROOT '<root(s)> '<category>)
 eg: (ROOT be)
     (ROOT (be have))
```

The ROOT function compares the root form of the current word to its argument. Again, the argument can be an atom or a list of atoms. It returns T if and only if the current word has one of the specified root forms. Thus the expression (ROOT be) would be true if the current word

were be, is, am, are, was, etc. Note that if the current word is homomorphic between two words with different root forms, both possibilities will be explored. Thus, if the current word were LIE, both (ROOT lay) and (ROOT lie) would be true.

The ROOT function takes an optional second argument which can be used to limit the lexical categories that are considered in determining the root form of the current word.

<u>GETF gets an inflectional feature</u>

(GETF '<feature name>)
eg: (GETF tense)

GETF returns the value associated with the feature <feature name> for the current word. If the current word has no such feature, then NIL is returned. These features come from three sources:

1. Features on the dictionary entry for the current word.

   For example, the tense and number features for the word IS are coded as:

   (IS (BE (tense present)(number 3sg))

2. Features generated by the Dictionary Manager.

   For regularly inflected words, certain features are automatically generated.

3. The FEATURES property of the word.

   Each word can have a FEATURES property associated with it. For example, the word GIVE might have the features property:

   (TRANSITIVE INDIRECTOBJECT PASSIVE)

   where these features might mean that GIVE is a transitive verb which can take an indirect object and can be used in a passive sentence.

In cases one and two above, a feature can be:

- A tuple whose first element is the name of the feature and whose second is the value.

- A list of just the feature name. The value of this feature is taken as T.

- An atomic feature name. The value is assumed to be T.

3.6.2 Actions which Test Arbitrary Words

<u>CHECKF checks an inflectional feature</u>

(CHECKF '<feature> <word> '<cat>)

The CHECKF function is analogous to GETF, differing only in that it applies to an arbitrary word given as its argument. CHECKF returns the value of the feature <feature> associated with the word <word>. An optional third argument, <cat>, determines the lexical category assumed for the word. Thus:

```
(CHECKF transitive (GETR verb) 'v)
```

returns the value of the TRANSITIVE feature for the word in the VERB register when it is interpreted as a verb. Similarly,

```
(CHECKF tns * 'v)
```

returns the TNS feature for the current input word when interpreted as a verb.

3.6.3 Miscellaneous Actions

The BUILDQ function provides a convenient method for constructing arbitrary trees which contain values held in registers. It is modeled after the function described in [Woods71], its syntax being:

```
(BUILDQ '<structure> . '<fillers>)
```

The first argument, <structure>, is an arbitrary list structure which can contain special symbols at its leaves. The function returns a similar structure by replacing these symbols by values computed from the remaining elements, <fillers>.

In filling the structure, BUILDQ traverses the structure in "preorder". If the symbol "+" is encountered, then the next filler argument is taken as the name of an ATN register and its contents are substituted for the "+". If the symbol "*" is encountered, it is replaced by the current value of the atom "*". On encountering the symbol "#", the next filler argument is evaluated and the resulting value used to replace the "#". If the symbol "@" is found as the first element of a list, then the remaining elements of that list are interpreted as above and the results are APPENDed together.

A few examples may make the operation of BUILDQ clear. Suppose we have the following register context:

```
DET              the
NOUN             man
PRE-MODIFIERS    (large heavy)
POST-MODIFIERS   ((PP (in the park)))
```

Then if we evaluate:

```
(BUILDQ (NP # (N +) (DET +)) (gensym) noun det)
```

we would get:

```
(NP G0027 (N man) (DET the))
```

Evaluating the expression:

```
(BUILDQ (NP # (N +) ( @ (MOD) + +))
        (gensym) noun pre-modifiers post-modifiers)
```

would yield the expression:

```
(NP (N man)
    (MOD large heavy (PP (in the park))))
```

(NEXTWRD) (NEXTWRDS)

The NEXTWRD function returns the next word in the input stream (i.e. the CAR of STRING). Note that this function does not invoke any of the Dictionary Manager Specialists such as the compound word recognizer or the word substituter (see section 5.3) but simply returns CAR of STRING. If there are no more words in the input stream, i.e. STRING is NIL, then NEXTWRD returns NIL.

The NEXTWRDS function returns a list of the possible next words in the input stream after all of the dictionary specialists have been applied.

3.7 Tracing Facilities

Every programming language should provide adequate debugging tools to enable the user to perfect his programs. Since the ATN should be viewed as a kind of programming language, we have provided powerful tracing facilities which we have found to be useful and necessary to build and debug large grammars.

Functions are provided which will print a trace of the developing parse and allow the user to specify arbitrary points at which to suspend computation and enter a "break point". A facility is also provided by which the user can monitor the values being assigned to particular registers.

TRACE-STATE traces ATN states

The function TRACE-STATE allows the user to trace the flow of control thru specified states and optionally break upon entering them. If a state is being traced, the message:

    in state <state name>

will be displayed on the terminal when the state is entered. If the state fails, i.e. no arcs from the state lead to a final state, the following message will be displayed:

    failing from state <state name>

If the user associates a BREAKPOINT with a state, then the message:

    ;BKPT <state name>

will be typed, computation suspended, and a LISP breakpoint entered. Typing an "$P" will resume the ATN computation.

The syntax of the TRACE-STATE function is:

    (TRACE-STATE '<arg1> '<arg2> ... '<argn>)

where each argument specifies one or more states to be traced and optionally associated with a breakpoint. The arguments are interpreted as follows:

<state name>	:	trace the named state
*	:	trace all states
(<state name> BREAK <test>)	:	trace the state and enter a breakpoint if <test> is true.
(* BREAK <test>)	:	trace all states and enter a breakpoint if <test> is true.

UNTRACE untraces ATN states

The function UNTRACE-STATE removes states from the set of ATN states to be traced. Its syntax is:

```
(UNTRACE-STATE '<arg1> ... '<argn>)
```

where each argument specifies one or more states to be "untraced". If an argument is the name of an ATN state being traced, then that state is removed from the list. The argument * causes all states to be removed from the list of states to be traced.

Registers can be traced as well

A similar feature allows one to monitor the values assigned to registers. The function TRACE-REG causes a specified ATN register to be "traced". Whenever a value is assigned to that register with a SETR, the message:

```
setting register <register name> to <value>
```

will be typed on the TTY. Similar messages are displayed when a register is set via the functions LIFTR or SENDR.

The syntax of the function TRACE-REG is similar to that of TRACE-STATE, being:

```
(TRACE-REG '<arg1> '<arg2> ... '<argn>)
```

An argument can be the name of a register, the special "wildcard" *, or a three-tuple (<register name> BREAK <test>). The last case allows one to trace the setting of a register and enter a BREAKPOINT if the arbitrary lisp expression <test> is true.

The function UNTRACE-REG is used to remove specified registers from the list of registers being traced. Its argument convention is similar to that of UNTRACE-STATE.

3.8 Global Variables

The following is a list of the most important global variables in the interpreter.

- * The current constituent
  The global variable * is always bound to the "current constituent. Precisely what this is depends on the context. Within a CAT arc, it is bound to the root form of the current word. Within a PUSH arc it is bound to the constituent POPped by the sub-computation (except in the "preactions" where * is the current word).

In a VIR arc, * is bound to the element being removed from the HOLD list. For the other arc types, it is bound to the current word (i.e. LEX).
In several other contexts, * is temporarily bound to other elements. For example, when evaluating the <test> in a VIR arc or in a (GETR <register> NEAREST <test>), * is bound to the candidate element.

- LEX The current word
  The atom LEX is always bound to the current word, exactly as it appears in the input string. If the input string is exhausted, then LEX is bound to NIL.

- STRING The input string
  The atom STRING is always bound to a list of the words remaining in the input string.

- STACK
  STACK is a list of the names of states PUSHed to which have not POPped. This can be used to determine how deeply embedded the current computation is. For example, the following POP arc will only be taken if we will be returning to the top level and the input is exhausted:

```
(POP (buildnp)
     (AND (NULL stack)(NULL *)(NULL string)))
```

The following additional global variables may be of some use to the user:

- ALIST is an association list which encodes the contents of the registers, both current and past. Elements of ALIST are tuples whose CAR is the name of a register and whose CDR is a value associated with that register.

- STATE is the current ATN state in interpreted grammars. The variable is undefined in compiled grammars.

- ARC is the current arc being taken in interpreted grammars. It is undefined in compiled grammars.

- USE-ALL-WORDS? is a variable which controls the ability to POP to the TOP-LEVEL when words remain in the input string. This is permitted only when USE-ALL-WORDS? is bound to NIL.

- LEXPRESCAN? is a variable which controls when the dictionary manager functions are invoked. If it is bound to T, then the dictionary manager is applied to the entire string before the grammar is applied and the results stored in a chart form. If NIL, then the dictionary manager functions are applied as input is needed.
- ALIST-STACK is stack of the values of ALIST for higher levels of computation.
- ARC-STACK is stack of pending actions on any active PUSH arcs (i.e. those which have not yet been popped to).
- POPSTACK is a stack of tags to the active PUSH arcs.

4.0 THE ATN COMPILER

4.1 Introduction

The purpose of the ATN Compiler is to translate ATN networks into pure LISP code which can be executed directly. This code can subsequently be translated into machine language code by a standard LISP compiler. In fact, the PLANES ATN compiler has been written in such a way as to facilitate the direct compilation of ATN networks into machine language by the MacLISP compiler [Moon74].

Burton describes another compiler for ATN grammars in [Burt76b]. The scope of this compiler is similar to the one described in this section, although the implementational details and some of the capabilities are quite different. Finin describes an ATN compiler written in the APL programming language which is very similar to the PLANES ATN compiler in [FINI78].

Advantages of compilation

The advantages of compiling an ATN network are the same as those of any compilation process: reduced execution time and storage space. We have applied our compiler on a modified version of the LUNAR English grammar [Wood72] and found that the execution time for the resulting LISP code was decreased by about a factor of 2. The resultant machine language code achieved a speed up factor of 5 to 10.

The compiler is able to preserve all of the features available under the interpreter. A grammar writer need take no special steps or consider-

ations if he intends to compile his network. In other words, if one has a system of ATN networks which behaves correctly in the interpreter, then compiling these networks without modifications will result in compiled code which also behaves correctly.

The PLANES ATN compiler has the additional feature of being able to incrementally compile ATN grammars. Each state is compiled independantly, making it easy to use the compiler on a grammar which is still evolving. Incremental compilation also allows one to replace the compiled version of a single state with an uncompiled version. Thus, if one discovers a bug in some state of a large compiled grammar, he can edit the source code and reload only the corrected state. The rest of the compiled grammar can still be used.

Special Features of the Compiler

A compiled ATN network is more efficient mainly because the interpretation stage is by-passed. This compiler also achieves greater efficiency thru four additional optimization processes:

1. The resulting LISP code is locally "optimized" by a simple pattern matching scheme.

2. Certain common actions and arc conditions are "open coded" into more primitive LISP code.

3. The compiler automatically detects local contexts in which certain ATN features are not being used (e.g. explicit failure via the FAIL function). In these contexts, code to handle these features need not be generated.

4. The user is able to globally "turn off" a number of general ATN features which he is not using. This causes the compiler not to produce the code necessary to implement these features.

4.1.1 Compilation of a State

The basic organization of the ATN compiler is shown in figure 5 (An Overview of the ATN Compiler). The function COMPILESTATE1 is applied to each state in the network. It produces a LISP function which will simulate the action of the state when called. In doing so, it applies the function COMPILEARC to each arc in the state. The resulting code for each arc is combined with additional code to produce the function.

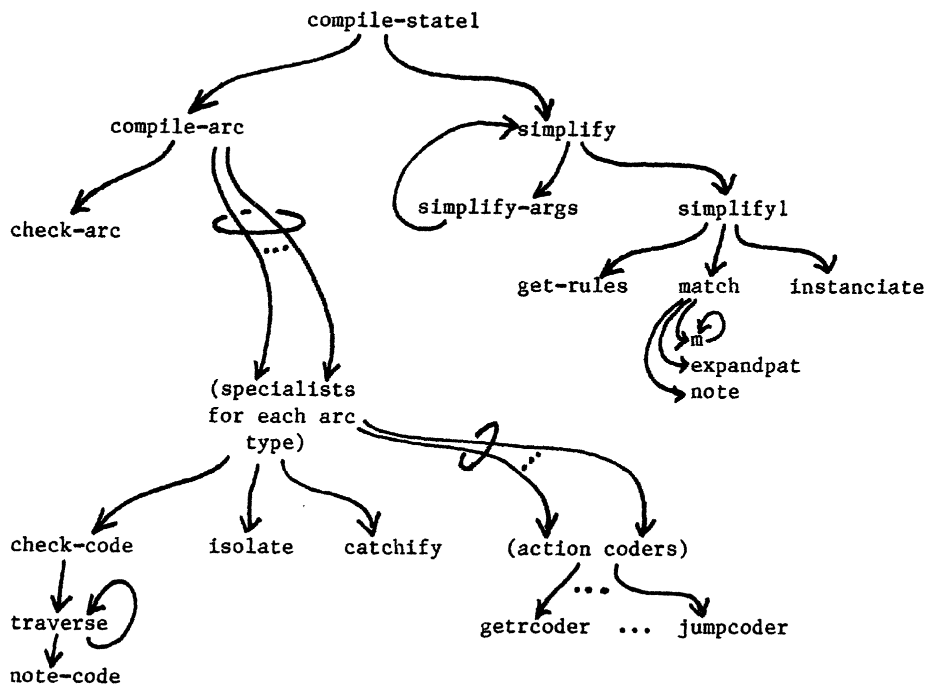


Fig. 5: An Overview of the ATN Compiler

The basic frame for an ATN state function is:

```
(DEFUN (<state name> ATNEXPR ATNSUBR)
       NIL
       <tracing code>
       (CATCH (PROGN <code for arc 1>
                     <code for arc 2>
                      ...)
              <state name> )
       <more tracing code>
       'FAIL)
```

DEFUN is, of course, the MacLisp function defining function. The first argument gives the name of the function being defined (<state name>) and the properties under which its uncompiled and compiled definitions are to be stored (ATNEXPR and ATNSUBR, respectively). The second argument to DEFUN is NIL, indicating that we are defining a function of no arguments.

The (CATCH ... <state name>) construct is included to implement the ability to fail to a named state from a later point. The inclusion of the tracing code and the (CATCH ... <state name>) expression is under the control of the user thru switches described in a later section.

Once this code has been produced, an optimization procedure is applied to the function. The result is then evaluated if we are in the LISP interpreter or passed to the MacLisp compiler if it is resident.

4.1.2 Compilation of an Individual Arc

The function COMPILEARC is responsible for writing an expression for an individual arc. Its job is to:

1. verify that the arc is in a legal format.

2. dispatch the arc to the appropriate specialist for compilation.

The verification includes a simple check on the last action of an arc. For arc types POP, TO, JUMP and DO this action should not specify a destination (i.e. call TO or JUMP). For the rest of the arc types, the terminal arc should be a destination. If this condition is not met a warning message is displayed.

Each arc compiler specialist compiles one type of arc. It examines the condition and actions and produces a LISP expression which simulates that arc. The general form of the code for an arc is given below:

```
(CATCH ((LAMBDA (ALIST) (AND <arc dependent code>
                             <test>
                             (PROGN . <actions>)))
         ALIST)
        ARCFAILURE)
```

The (CATCH ... ARCFAILURE) construction is necessary to handle a possible (FAIL 'ARC) in the <actions> or the <test>. The ((LAMBDA (ALIST) ...) ALIST) is required to isolate the possible side effects caused by the arc.

Before adding these two fragments, the arc compiler checks to see if they are really necessary. The <test> and <actions> are examined to see if they might contain a call to a function with a side effect (e.g. a call to SETR) or a call to the FAIL function. In examining the expressions, the compiler must be aware of which expressions are "executable" and which are potentially "data".

The following conservative algorithm is used to determine whether evaluating an s-expression will generate a side effect.

1. Trivial case
   Atoms, of course, can't cause side effects. Return NIL.

2. Known offenders
   Some functions are known to cause side effects (e.g. SETR). Return T.

3. Guilty until proven innocent
   If the function is unknown (i.e. one which the user has written) then assume it might generate a side effect. Return T.

4. Fexpers, etc.
   If the function has no arguments or is known not to evaluate any of its arguments, then return NIL.

5. Recurse
   Recursively apply this procedure to those arguments of the s-expression which are evaluated. If any generates a side effect, then return T, else return NIL. Here the compiler must know how

a function evaluates its arguments. For the standard built-in LISP functions and the built-in ATN functions and actions this information is stored for each function in an "evaluation template".

An analogous procedure is used to determine if evaluating an expression might generate a failure. If the arc contains no expressions which might have a side effect then the LAMBDA expression is omitted. If no FAIL action is found, then the CATCH is omitted.

Another of the compiler's optimizing technique is to delay the CATCH (if any) and LAMBDA binding (if any) until the last possible moment. Thus, the following WRD arc:

```
(WRD because (getr obj)(setr conj *)(to s:conjoined))
```

is compiled as:

```
(AND (EQ * 'because)
     (GETR obj)
     ((LAMBDA (ALIST) (SETR conj *)(TO s:conjoined))
      ALIST))
```

Note that the LAMBDA binding is done only if the arc is actually taken and that no CATCH was generated since no FAIL action was contained in the arc.

4.1.3 Optimization

The optimization phase of the compiler was originally added to make the compiler easier to write and debug. The idea was to simplify the code generation process for each of the arcs. In compiling an arc, we needn't produce efficient code, but merely straightforward code which is guaranteed to work properly. Once the code has been generated, a standard "optimizing" routine is applied to it to simplify the expression.

The General Procedure

The basic procedure to optimize a LISP s-expression it to traverse the expression in "post order", applying a simplification algorithm at each node. Thus the expression is optimized "from the bottom up", i.e. the arguments to a function are optimized before the function call itself is optimized. We must take care, however, that the optimization proce-

dure is only applied to "executable" nodes and not to "data nodes". An "executable" node is one which is known to be evaluated as a LISP expression. All other nodes are assumed to be "data" nodes.

Thus, the optimizer does not attempt to optimize a function call unless that function is:

- o known to be a EXPR, SUBR, LEXPR or LSUBR
- o known to be a FEXPR or FSUBR which evaluates all of its arguments (e.g. AND, OR, etc.)
- o a special FEXPR or FSUBR that the optimizer "understands".

The functions covered in the third case are those which are known to evaluate some of these arguments in a special manner. For example, the optimizer knows that the function PROG evaluates all but its first argument and that the arguments to the COND function are lists of expressions to be optimized.

Simplification

Once the arguments to a function have been optimized, the function call is simplified. This simplification is driven by a set of production-like rules which specify code transformations. These transformations simplify the code but do not change its behavior. These rules are three-tuples which have the form:

(<pattern> <result> <restrictions>)

Each rule states that if a LISP expression matches <pattern> and <restrictions> evaluates to non-NIL, then it can always be replaced by the simpler s-expression <result>.

The <pattern> and <result> elements of a rule are general s-expressions in which the characters * and ? have special significance. The atom "?" is allowed to match any single s-expression. The atom "*" may match zero or more sister s-expressions. Any atom whose first character is a "?" or a "*" behaves in a similar manner except that that atom is bound to the s-expression(s) matched. As an example, consider the pattern:

(PROG ?A *B (RETURN *))

when matched against the following expression:

```
(PROG (A B C)
      (SETQ A (TIMES X X))
      (SETQ B (TIMES X A))
      (RETURN (LIST A B)))
```

The match succeeds and has the side effects of setting the variables ?A and *B to (A B C) and ((SETQ A (TIMES X X))(SETQ B (TIMES X A))), respectively. Figure 6 (Some Simplification Rules) gives a list of some of the simplification rules used in the ATN compiler.

Application of the simplification rules

The simplification rules are applied according to the following algorithm:

1. Collect the rules which may apply.
   The rules are indexed by function name (i.e. the CAR of the expression).

2. Find a rule which matches the expression.
   The matcher compares the <pattern> and <restriction> of each rule against the data until a match is found.

3. If no matching rule was found, then stop else continue.

4. Replace the expression.
   Replace the expression with the instantiation of the <result> element of the rule found in step 2.

5. Continue with step 1.
   Since the function being called may have changed, we must start from the top.

Figure 7 (An Example of Simplification) shows an example of the simplification process.

4.2 Using the ATN Compiler

The ATN compiler can easily be used to produce executable LISP code in the LISP interpreter or to produce machine language code in conjunction with the standard MacLisp compiler. In either case, the user can selectively disable unused features to achieve faster execution time. This is discussed in section 4.3.3.

(AND)	NIL	
(AND *A T *B)	(AND *A *B)	(OR *A *B)
(AND ?A)	?A	
(AND *A (AND *B) *C)	(AND *A *B *C)	
(APPEND *A (LIST *B)(LIST *C) *D)	(APPEND *A (LIST *B *C) *D)	
(CAR (CAR ?A))	(CAAR ?A)	
...etc...		
(COND)	NIL	
(COND *A (NIL *) *B)	(COND *A *B))	
(COND *A (T *B) *C)	(COND *A (T *B))	*C
(COND (?A ?B))	(AND ?A ?B)	
(COND (T *A))	(PROGN *A)	
(COND (?A))	?A	
(COND *E (?A *B (PROGN *C) *D) *F)	(COND *E (?A *B *C *D) *F)))	
(CONS ?A NIL)	(NCONS ?A)	
(DEFUN ?A ?B *C (PROGN *D) *E)	(DEFUN ?A ?B *C *D *E)	
(DEFUN ?A ?B *C NIL *D)	(DEFUN ?A ?B *C *D)	
(DO ?A ?B *C (PROGN *D) *E)	(DO ?A ?B *C *D *E)	
(EQ ?A NIL)	(NULL ?A)	
(EQUAL '?A ?B)	(EQ '?A ?B)	(ATOM ?A)
(EQUAL ?A '?B)	(EQ ?A '?B)	(ATOM ?B)
(LAMBDA ?A *B (PROGN *C) *D)	(LAMBDA ?A *B *C *D)	
(LIST ?A)	(NCONS ?A)	
(PROG ?A *B (PROGN *C) *D)	(PROG ?A *B *C *D)	
(PROGN ?A)	?A	
(PROGN *A (PROGN *B) *C)	(PROGN *A *B *C)	
(OR)	NIL	
(OR ?A)	?A	
(OR *A T *)	(OR *A)	
(OR *A NIL *B)	(OR *A *B)	

Fig. 6: Some Simplification Rules

```
(PROGN (AND T (COND ((> X 10.)(SETQ X 10.))))

               |_(cond (?a ?b))  ==>  (and ?a ?b)

(PROGN (AND T (AND (> X 10.)(SETQ X 10.))))

        |_(and *a (and *b) *c)  ==>  (and *a *b *c)


(PROGN (AND T (> X 10.)(SETQ X 10.)))

        |_(and *a t *b)  ==>  (and *a *b)

(PROGN (AND (> X 10.)(SETQ X 10.)))
|_(progn ?a)  ==>  ?a

(AND (> X 10.)(SETQ X 10.))
```

Fig. 7: An Example of Simplification

4.2.1 Compiling Networks into LISP Code

This section describes how one can compile networks into LISP code given he has a LISP into which the ATN interpreter has been loaded and the networks defined. Once the ATN compiler has been loaded, the user can then set whatever switches he wants (using SETQ). A single ATN state can be compiled with the function COMPILESTATE. This function takes one (unevaluated) argument, the name of the ATN state to be compiled. The function defines an equivalent function which will be called whenever control is transferred to that ATN state.

The function COMPILEATN can be used to compile an entire ATN network. Its single (unevaluated) argument should be the name of the ATN network to be compiled. It simply applies COMPILESTATE to each of the states found in the network.

4.2.2 Compilation in the Lisp Compiler

To compile a file of ATN networks with the standard MacLisp compiler one need only ensure that the ATN compiler is loaded into the MacLisp compiler. The easiest way to do this is to include the following expression at the top of the file containing an ATN grammar.

```
(DECLARE (FASLOAD ATNCOMPILER ... ))
```

This will cause the ATN compiler to be automatically loaded when the file is compiled. In the environment of the compiler, the functions DEFARC, DEFSTATE, DEFINTERUPT, etc. are defined to be macros compile their arguments into Lisp code. Thus, the Lisp compiler sees only the Lisp code and not the grammar in the ATN formalism.

Switches may be set in a similar manner. For example, one might include the expression:

```
(DECLARE (FASLOAD ATNCOMPILER ...)
         (SETQ ATNTRACE? NIL)        ;no tracing
         (SETQ FAILARC? NIL))        ;no (FAIL 'ARC) actions
```

in the file containing the networks.

<u>Interactions with NCOMPLR</u>

If the ATN networks call any user defined functions they may have to be declared to NCOMPLR. Auxiliary functions must be declared if they are

FEXPRs, FSUBRs, LEXPRs, LSUBRs, or MACROs. For an explanation of how to do this, consult [Moon74].

4.2.3 Switches

The following switches control the generation of code by the ATN compiler. For each switch the default value is given after its name.

[1] ATNTRACE? T
If ATNTRACE? is T then code which allows states to be traced (i.e. via TRACE-STATE) will be generated. A NIL value supresses the generation of this code. Thus, this switch might be set to NIL if the network is fully debugged or a production module is desired.

[2] ATNFAILARC? T
If ATNFAILARC? is T then code will be generated which allows one to use the (FAIL 'ARC) action. If NIL, then this code is not generated.

[3] ATNFAILSTATE? T
Code which allows one to fail to a named state is generated if this switch is T. If NIL, then the code is not generated. Recall that one can fail to a named state with the FAIL arc or the FAIL action.

[4] ATNFAILPUSH? T
If T, then code will be generated which allows one to fail from a sub-computation (i.e. to the last PUSH arc or PUSHATN action). If NIL, then the code will not be generated.

[5] ATNFAIL? T
This switch has no effect if it has the value T, but a NIL value disables all explicit failure. Thus, if ATNFAIL? is NIL, this implies that ATNFAILARC = ATNFAILSTATE = ATNFAILPUSH = NIL.

[6] ATNSIMPLECAT? NIL
This switch controls the code generated for a CAT arc and the CAT action. If it is T, then it is assumed that each word will have at most <u>one</u> entry for a particular lexical category in the dictionary. If this is so, the compiler need not set up a decision point for each CAT arc or action.

5.0 THE LEXICON

5.1 Introduction

The lexicon consists of two parts: the DICTIONARY which contains words and certain features associated with them and the DICTIONARY MANAGER which is a package of procedures for accessing, maintaining, and updating the dictionary.

A dictionary entry for a word typically consists of the syntactic category that the word belongs to and a list of syntactic features for the word when interpreted under that category. In English, as in many languages, a word can belong to several lexical categories. In such cases, multiple entries can be made in the Dictionary. For example the word CRASH might have the following entries:

```
CRASH      N -es
CRASH      V -es-ed (intransitive)
```

This says that crash can be interpreted as a noun (N) whose plural is formed be adding the suffix "es" or as an intransitive verb (V) whose inflectional forms can be generated by adding the suffixes "es" or "ed".

Initially, no entries need exist for the regularly inflected forms of words. Such forms are discovered and entries for them generated as they are needed. For example, if the input contains the word CRASHED, a set of programs (Morphology Specialists) are invoked which discover that CRASHED is the past tense form of the root word CRASH. These programs insert the lexical entry:

```
CRASHED    V (CRASH) (tense past)(intransitive)
```

The words CRASHES and CRASHING are similarly recognized and their lexical entries generated.

5.2 The Dictionary Format

The dictionary entry for a word is stored on the LISP property list for that atom (i.e. the word). The standard property names which might be used are atoms representing lexical categories (e.g. N for noun, DET for determiner, etc.), and three special atoms: COMPOUNDS, SUBSTITUTE, and FEATURES.

For each lexical category, <C>, a word might have a property indexed under the indicator <C>. The value of this property depends on the par-

ticular category and the relationship between the word and its root form. These are discussed in the next section.

The value of the COMPOUNDS property for a word is a tree which represents pre-defined compound phrases which that word may start. Associated with each phrase in the tree is a word which that phrase should be mapped into. The user need not concern himself with the exact representation of this tree as the functions DEFPHRASE and DEFP (described later) can be used to create it.

The value of the SUBSTITUTE property for a word is a list of phrases which should replace that word. Thus, it forms a kind of inverse to the COMPOUNDS mechanism. This property is discussed further under the functions DEFSUBSTITUTE and DEFSUB.

The value of the FEATURES property is a list of atoms or sublists which name "features" to be associated with the word. These features may be accessed in several ways within the ATN interpreter (see, for example, the functions GETF and CHECKF).

The value of a lexical category property for a word specifies the root form(s) of the word and a set of inflectional features to be associated with the word. The exact encoding of this information is as follows:

1. If the value is a list whose first element is atomic, then the first element is the root form of the word and the remaining elements are the inflectional features. For example:

   ```
   BLEW  V (blow (tns past))
   ```

   says that the word BLEW is the past tense of the verb BLOW.

2. If the value is a list whose first element is a sub-list, then the word has more than one possible root form when interpreted under this lexical category. Each sub-list should be of the form discussed in point one above. AN example is:

   ```
   SAW  V ((see (tns past))
          ((saw (tns present)(untensed))))
   ```

   This property states that the word SAW is <u>either</u> the past tense form of the verb SEE <u>or</u> the present/untensed form of the verb SAW (what we do to a board).

3. If the value is atomic, then the word itself is taken as the root form and the inflectional features are supplied by the dictionary manager. Exactly what features are supplied is described in the next section. Some examples are:

```
COOK V s-ed
     N -s
FAST ADJ er-est
     V   s-ed
     N   -s
```

<u>The meaning of atomic category values</u>

The lexical categories N (noun), V (verb), ADJ (adjective), and ADV (adverb) are treated in a special manner by the dictionary manager. An atomic value for one of these properties can specify that the word fits one of a number of regularly inflected paradigms. For nouns, the following possibilities exist:

*	The noun is irregular.
-s	The plural is formed by adding an "s".
-es	The plural is formed by adding an "es" or, if the noun ends in a "y", by deleting the "y" and adding an "ies".

For verbs, we have the following:

*	The verb is irregular.
s-ed	The third person singular is formed by adding an "s" and the past/past-participle is formed by adding an "ed".
es-ed	The third person singular is formed by adding an "es" and the past/past-participle is formed by adding an "ed".

For adjectives (ADJ) and adverbs (ADV) we have the following possibilities:

*	The word has no regularly inflected comparative or superlative (e.g. UNSUCCESSFUL).
er-est	The comparative and superlative are formed by adding an ER and EST, respectively (e.g. LONG). If the word ends in a Y, then it is first changed to an I (e.g. HAPPY).
r-st	Add an R or ST to form the comparative and superlative (e.g. WIDE).

5.3 The Dictionary Manager

The Dictionary Manager is a collection of programs which access, maintain, and update the Dictionary. They also act as a filter between the

Parser and the user's input suggesting alternative words as the 'next word'. As each word of the input is needed by the parser, the Dictionary Manager checks to see if there is an entry in the Dictionary for that word or if the word belongs to certain categories which are recognized procedurally, such as numbers. If an entry exists, then the word is passed on to the parser with a list of features associated with the word. These features are those found in the dictionary together with any others suggested by the Morphology Specialists.

If the word does not have a lexical entry, then a series of Morphology Specialists are invoked to see if the word is a regularly inflected form of a known word. These specialists use their knowledge of typical English affixes to propose candidate 'roots' for the word. Each candidate is then looked up in the Dictionary and, if found, checked to see if it accepts the affix removed.

If this process fails then the Punctuation Checker is called which examines the word to see if it might contain any embedded punctuation. For example, if the input is TAIL-NUMBER, the Punctuation Checker would suggest that the two words TAIL and NUMBER be substituted.

Finally, if the word is still unknown, control is passed to the New Word Learner. This module interacts with the user and attempts to create a lexical entry for the word.

Two other subsystems generate alternative suggestions for the 'next word': a Compound Word Recognizer and a Word Substituter. The Compound Word Recognizer is used to map short phrases into single "words". For example, we might map the phrases UNITED STATES and UNITED STATES OF AMERICA into the single "word" USA. Just before a new word is passed to the parser, the Compound Word Recognizer checks to see if that word can begin a phrase that it knows. If so, then it checks the rest of the sentence, and if it matches, suggests the alternative 'next word'.

The Word Substituter provides a similar mechanism - one which can expand a single word in the input into a sequence words. For example, the word DIDN'T could be expanded into the sequence DID NOT. The sequence of words substituted can be of zero length, which provides a facility for ignoring words. This facility might be used, for example, to ignore extraneous punctuation.

5.4 Auxiliary Functions

Several functions are provided which simplify creating dictionary entries and loading and manipulating dictionary files.

DEFINEWORD and DEFW create dictionary entries for a word

```
(DEFINEWORD <word> <definition>)
(DEFW '<word> . '<definition>)
```

These functions are used to create a dictionary entry for the word <word> or to add information to the dictionary entry for <word> if an entry is already in the dictionary.

The function DEFINEWORD takes two arguments: a word and a "property list fragment". It adds the property list fragment to the property list of the word.

The function DEFW takes an indefinite number of arguments, none of which are evaluated. The first is the word and the remaining ones are alternating property names and values. The name-value pairs are added to the word's property list.

Examples of these functions in use are:

```
(DEFINEWORD 'fly '(v * n -es features (intrans trans)))
(DEFINEWORD 'i '(pro *))
(DEFW ship n -s v -s-ed features (trans passive))
(DEFW ? punct *)
```

DEFINEPHRASE and DEFP Create Compounds properties

```
(DEFINEPHRASE <phrase> <word>)
(DEFP '<phrase> '<word>)
```

These functions create COMPOUNDS entries for a word. A COMPOUNDS entry will cause the ATN interpreter to map a sequence of words in the input string into a single word. These functions will cause the input sequence <phrase> (which should be a list of atoms) to be replaced by the single word <word>. For example:

```
(DEFP (how much) howmuch)
```

would create or modify the COMPOUNDS entry for the word HOW such that if the ATN interpreter encounters the two word sequence "HOW MUCH" in

the input string, it will replace these two words with the single word HOWMUCH. Note that if subsequent parsing fails, the interpreter will also try parsing the string as containing the two individual words HOW and MUCH.

Several COMPOUNDS entries can be created at once by including sub-lists in the first argument. For example,

```
(DEFP (how (much many)) howmuch)
```

is equivalent to:

```
(DEFP (how much) howmuch)
(DEFP (how many) howmuch)
```

Thus, a sublist in the <phrase> argument specifies that one of its elements be in the compound phrase.

<u>DEFSUBSTITUTE and DEFSUB create SUBSTITUTE entries</u>

```
(DEFSUBSTITUTE <word> <phrase>)
(DEFSUB '<word> '<phrase>)
```

These functions create or modify SUBSTITUTE properties for a word. The first argument is an atom and the second argument is an atom or list of atoms which should be substituted for the first whenever it occurs in the input string. The following example will explain the various possibilities:

- (DEFSUB aircraft plane)
  If the second argument is an atom (but not the atom NIL), then it will replace the first argument in the input string. Thus, PLANE will replace instances of the word AIRCRAFT in the input string.

- (DEFSUB don't (do not))
  If the second argument is a list of atoms, then that list will replace the first argument in the input string. Thus, occurances of the word DON'T will be replaced by the two words DO and NOT.

- (DEFSUB please NIL)
  If the second argument is NIL, then the first argument will be deleted from the input string. Thus instances of the word PLEASE will be deleted from the input string. This provides a simple method for ignoring certain words (such as "extraneous" punctuation).

As with the COMPOUNDS mechanism, if a substitution is made and failure later backs up to that point, the ATN interpreter will attempt to continue the parse without making the substitution.

Dictionary Files

If one is dealing with a large number of words it is convenient to store them in a file in a special simplified format. Two functions are provided to manipulate such dictionary files.

The format of a dictionary file is a sequence of lists. The first element of each list is an atom or a list. If it is an atom, then the list is taken as the argument list to the function DEFW. In other words, the first element is a word and the remaining elements are alternating property-names and associated values. If the first element is a sub-list, then the entire list is used to supply arguments to the function DEFP. In this case, the first element is a phrase and the remaining element (there should be only one) is the word that the phrase gets mapped into. The following might be the beginning of such a dictionary file:

```
(a det *)
(about prep *)
(an det *)
((and / or) and-or)
(and-or conj *)
(ask v s-ed features (trans indobj thatcomp tocomp))
   .
   .
   .
```

LOADDIC loads a Dictionary File

(LOADDIC . '<file specifications>)
eg: (LOADDIC navy dic ...)

The function LOADDIC is used to read a dictionary file and create the specified entries. Its syntax is similar to that for the MacLISP I/O functions ("old I/O" functions, that is). It takes from one to four arguments where the first is the file name, the second the file extension, the third the device and the fourth is the PPN. Unsupplied arguments are defaulted as follows:

arg 1	:	<file name>	:	must be supplied
arg 2	:	<file extension>	:	defaults to DIC
arg 3	:	<device>	:	defaults to current device
arg 4	:	<ppn>	:	defaults to the current ppn

GRINDDIC reformats a dictionary file

(GRINDDIC . '<file specifications>)
eg: (GRINDDIC navy dic)

This function sorts and grinds (i.e. pretty-prints) a dictionary file. The sorting function sorts each top level list in the file by the "left-most" atom in the list only. The argument syntax and meaning is identical to the function LOADDIC. The function GRINDDIC is useful in maintaining growing dictionaries in an easily readable format.

Miscellaneous dictionary managing functions are provided

Currently, one other auxiliary function is supplied for managing a dictionary. The function SHOWWORD "pretty-prints" the dictionary entry for a word. It's syntax is:

(SHOWWORD '<word>)

6.0 AUXILIARY FUNCTIONS

6.1 Interfacing with the LISP Editor

The functions EDITATN and EDITSTATE allow one to apply a general purpose List structure editor [Gabr75] to ATN networks and individual states. These functions do not normally reside in the ATN system but are automatically loaded if they are called. In addition, this subsystem modifies appropriate Editor variables so that the editor can recognize ATN network definitions in files. Thus, one can say (EDIT <network name> <file specifications>) and expect the editor to find the network definition in the specified file. This also allows one to use the REFILE editor command to update the file once the networks have been edited. A special purpose ATN Grammar editor called NETEDI is also available. This editor is described in [Fini77b].

EDITATN applies the Editor to an ATN network

This function sets up an ATN network for editing and then invokes the LISP Editor. It takes one or no arguments, the syntax being:

```
(EDITATN <network name>)
(EDITATN <state name>)
(EDITATN *)
(EDITATN)
```

If EDITATN is called with the name of a network, then that network is set up for editing. If called with the name of an individual state, then the editor is invoked on the network which includes that state. If called with the argument *, then the network which includes the current state is edited. If no argument is given, then the network which was last edited is set up for re-editing.

EDITSTATE invokes the editor on an individual ATN state

The function EDITSTATE is similar to EDITATN expect that it applies the editor to an individual ATN state. If an argument is given, it can be the name of the state to be edited or the special atom * which refers to the state that the interpreter is currently in. If no argument is given, the editor is primed with the last edited ATN state. The syntax is:

```
(EDITSTATE '<state name>)
(EDITSTATE *)
(EDITSTATE)
```

6.2 Displaying ATN Networks and States

SHOWATN displays an ATN network

The function SHOWATN is used to display an entire ATN network in "pretty print" format. The optional argument is interpreted in a manner identical to that for the function EDITATN. The options are:

```
(SHOWATN '<network name>)
(SHOWATN '<state name>)
(SHOWATN *)
(SHOWATN)
```

SHOWSTATE displays an individual ATN state

The SHOWSTATE function "pretty prints" an individual ATN state. Its optional argument is interpreted like that of EDITSTATE. The syntax is:

```
(SHOWSTATE '<state name>)
(SHOWSTATE  )
(SHOWSTATE)
```

7.0 EXAMPLES

7.1 A Sample ATN Network

The following three ATN networks comprise a simple grammar to a small subset of English. Note that most features of English syntax are not handled. Two auxiliary functions, BUILDSENTENCE and BUILDNP, are used to construct the parse trees.

```
(defatn sentence
    ((s: (cat aux t (setr v *) (to s:aux))
         (push np: t (setr subj *) (to s:subj))
         (vir np t (setr subj *) (to s:subj)))
     (s:aux (push np: t (setr subj *) (to s:v)))
     (s:subj (cat v t (setr v *) (to s:v)))
     (s:v (cat v
               (getf pastpart)
               (cond ((eq $v 'be)
                       (hold $subj 'np nil)
                       (setr subj 'someone))
                      ((eq $v 'have)
                       (setr aspect 'perfect))
                      (t (fail 'arc)))
               (setr v *)
               (to s:v))
          (push np: (getf transative $v) (setr obj *) (to s:obj))
          (vir np (getf transative $v) (setr obj *) (to s:obj))
          (and (wrd to (getf strans $v))
               (push s:subj
                     t
                     (sendr subj $obj)
                     (setr obj *)
                     (to s:end)))
          (and (wrd that (getf strans $v))
               (push s: t (setr obj *) (to s:end)))
```

```
        (jump s:end (getf intrans $v)))
   (s:obj (push np:
                (getf indobj $v)
                (setr indobj $obj)
                (setr obj *)
                (to s:end))
          (vir np
               (getf indobj $v)
               (setr indobj $obj)
               (setr obj *)
               (to s:end))
          (and (wrd to)
               (push s:subj
                     t
                     (sendr subj $obj)
                     (setr obj *)
                     (to s:end)))
          (jump s:end))
   (s:end (push pp: (cat prep) (addr pps *))
          (pop (buildsentence)))))

(defatn noun-phrase
   ((np: (cat det t (setr det *) (to np:det)) (jump np:det))
    (np:det (cat adj t (addr adj *) (to np:det))
            (cat n t (addr adj *) (to np:det))
            (cat n t (setr n *) (to np:n)))
    (np:n (push pp: (cat prep) (addr modifiers *) (to np:n))
          (and (wrd (which who that whom)
                    t
                    (hold (buildnp) 'np nil))
               (push s: t (addr modifiers *) (to np:end)))
          (jump np:end))
    (np:end (pop (buildnp)))))

(defatn prep-phrase
   ((pp: (cat prep t (setr prep *) (to pp:prep)))
    (pp:prep (push np: t (setr np *) (to pp:end))
             (vir np t (setr np *) (to pp:end)))
    (pp:end (pop (buildq (pp + +) prep np)))))
```

```
(defun buildsentence nil
    (buildq (s (verb +) (subj +) (obj +) (indobj +) (pps +))
            v
            subj
            obj
            indobj
            pps))

(defun buildnp nil
    (buildq (np (n +) (det +) (@ (adj) +) (@ (modifiers) +))
            n
            det
            adj
            modifiers))
```

7.2 A Sample Dictionary

This example shows a small dictionary which was used with the example network to parse the sentences in this appendix. Only the lexical features used by the simple grammar are included for these words.

```
(a det *)
(aircraft substitute ((plane)))
(by prep *)
(electrical adj *)
(engine n -s)
(extensive adj *)
(for prep *)
(good adj *)
(has v (have (tns present)))
(have v irr features (transative))
(I n *)
(in prep *)
(list n -s v s-ed features (transative))
(maintenance n -s)
(me n (i))
(of prep *)
(plane n -s)
(poor adj er-est)
(record n -s)
(repair n -s v s-ed features (transative))
(require v s-d features (transative))
(see v irr features (transative intransative))
```

```
(show v s-ed features (transative indobj))
(that det *)
(the det *)
(to prep *)
(very adj *)
(want v s-ed n -s features (transative indobj strans))
(were v (be (tns past)))
(which det *)
(which det *)
(you n *)
(? substitute (nil))
(/. substitute (nil))
```

7.3 Examples of Operation

Given below are the results of applying the example ATN network to six sentences. For each, the sentence is given exactly as it was typed, preceded by a ">>". Following that, the list POPed by the ATN network is given.

```
>> which planes required engine maintenance?

(S (VERB REQUIRE)
   (SUBJ (NP (N PLANE) (DET WHICH) (ADJ) (MODIFIERS)))
   (OBJ (NP (N MAINTENANCE) (DET NIL) (ADJ ENGINE) (MODIFIERS)))
   (INDOBJ NIL)
   (PPS NIL))

>> electrical repairs were required.

(S (VERB REQUIRE)
   (SUBJ SOMEONE)
   (OBJ (NP (N REPAIR) (DET NIL) (ADJ ELECTRICAL) (MODIFIERS)))
   (INDOBJ NIL)
   (PPS NIL))

>> I want a list of those planes

(S (VERB WANT)
   (SUBJ (NP (N  I) (DET NIL) (ADJ) (MODIFIERS)))
   (OBJ (NP (N PLANE) (DET THOSE) (ADJ) (MODIFIERS)))
   (INDOBJ (NP (N YOU)
               (DET NIL)
               (ADJ)
```

```
               (MODIFIERS (PP TO
                               (NP (N LIST)
                                   (DET NIL)
                                   (ADJ)
                                   (MODIFIERS))))))
     (PPS NIL))
```

>> I want you to list those planes.

```
(S (VERB WANT)
   (SUBJ (NP (N I) (DET NIL) (ADJ) (MODIFIERS)))
   (OBJ (S (VERB LIST)
           (SUBJ (NP (N YOU) (DET NIL) (ADJ) (MODIFIERS)))
           (OBJ (NP (N PLANE) (DET THOSE) (ADJ) (MODIFIERS)))
           (INDOBJ NIL)
           (PPS NIL)))
   (INDOBJ NIL)
   (PPS NIL))
```

>> I want to see a list of the planes that required extensive engine repairs.

```
(S
 (VERB WANT)
 (SUBJ (NP (N I) (DET NIL) (ADJ) (MODIFIERS)))
 (OBJ
  (S (VERB SEE)
     (SUBJ (NP (N I) (DET NIL) (ADJ) (MODIFIERS)))
     (OBJ (NP (N LIST)
              (DET A)
              (ADJ)
              (MODIFIERS (PP OF
                              (NP (N PLANE)
                                  (DET THE)
                                  (ADJ)
                                  (MODIFIERS (S (VERB REQUIRE)
                                                (SUBJ (NP (N PLANE)
                                                          (DET THE)
                                                          (ADJ)
                                                      (MODIFIERS)))
                                               (OBJ (NP (N REPAIR)
                                                        (DET NIL)
```

```
                                                          (ADJ
                                                           EXTENSIVE
                                                           ENGINE)
                                                         (MODIFIERS)))
                                                (INDOBJ NIL)
                                                (PPS NIL)))))))
    (INDOBJ NIL)
    (PPS NIL)))
 (INDOBJ NIL)
 (PPS NIL))
```

```
>> I want you to show me the planes which have poor
   maintenance records.
```

```
(S (VERB WANT)
   (SUBJ (NP (N I) (DET NIL) (ADJ) (MODIFIERS)))
   (OBJ (S (VERB SHOW)
           (SUBJ (NP (N YOU) (DET NIL) (ADJ) (MODIFIERS)))
           (OBJ (NP (N PLANE)
                    (DET THE)
                    (ADJ)
                    (MODIFIERS (S (VERB HAVE)
                                  (SUBJ (NP (N PLANE)
                                            (DET THE)
                                            (ADJ)
                                            (MODIFIERS)))
                                  (OBJ (NP (N RECORD)
                                           (DET NIL)
                                           (ADJ POOR MAINTENANCE)
                                           (MODIFIERS)))
                                  (INDOBJ NIL)
                                  (PPS NIL)))))
           (INDOBJ (NP (N I) (DET NIL) (ADJ) (MODIFIERS)))
           (PPS NIL)))
   (INDOBJ NIL)
   (PPS NIL))
```

7.4 Examples of Tracing States

The following example shows a trace of the action of the ATN network in parsing a sentence. This trace was generated by evaluating (TRACE-STATES *). Again, the input string is preceded by ">>" and the final value

POPed by the network is displayed at the end. Values POPed by final states have been abbreviated.

```
>> Which aircraft have very poor maintenance records?

in state S:
pushing to state NP:
   in state NP:
   in state NP:DET
   in state NP:DET
   failing from state NP:DET
   in state NP:N
   jump to state NP:END
   in state NP:END
   pop from state NP: with value (NP (N PLANE) ... )
jump to state S:SUBJ
in state S:SUBJ
in state S:V
pushing to state NP:
   in state NP:
   jump to state NP:DET
   in state NP:DET
   in state NP:DET
   in state NP:DET
   in state NP:DET
   in state NP:DET
   failing from state NP:DET
   in state NP:DET
   failing from state NP:DET
   in state NP:N
   jump to state NP:END
   in state NP:END
   pop from state NP: with value (NP (N RECORD) ... )
jump to state S:OBJ
in state S:OBJ
jump to state S:END
in state S:END
pop from state TOPLEVEL with value (S (VERB HAVE) ... )
```

```
(S (VERB HAVE)
   (SUBJ (NP (N PLANE) (DET WHICH) (ADJ) (MODIFIERS)))
   (OBJ (NP (N RECORD)
            (DET NIL)
            (ADJ VERY POOR MAINTENANCE)
            (MODIFIERS)))
   (INDOBJ NIL)
   (PPS NIL))
```

7.5 Examples of Tracing Registers

This example shows a trace of the values assigned to ATN registers as well. This was achieved by evaluating (TRACE-STATES *) and (TRACE-REG *).

```
>> I want you to list those planes

in state S:
pushing to state NP:
   in state NP:
   jump to state NP:DET
   in state NP:DET
   setting register ADJ to  (I)
   in state NP:DET
   failing from state NP:DET
   setting register N to  I
   in state NP:N
   jump to state NP:END
   in state NP:END
   pop from state NP: with value (NP (N I) ... )
setting register SUBJ to  (NP (N I) ... )
jump to state S:SUBJ
in state S:SUBJ
setting register V to WANT
in state S:V
pushing to state NP:
   in state NP:
   jump to state NP:DET
   in state NP:DET
   setting register ADJ to  (YOU)
   in state NP:DET
   failing from state NP:DET
   setting register N to  YOU
   in state NP:N
```

```
   jump to state NP:END
   in state NP:END
   pop from state NP: with value (NP (N YOU) ... )
setting register OBJ to (NP (N YOU) ... )
jump to state S:OBJ
in state S:OBJ
pushing to state NP:
   in state NP:
   jump to state NP:DET
   in state NP:DET
   failing from state NP:DET
   failing from state NP:
sending register SUBJ to (NP (N YOU) ... )
pushing to state S:SUBJ
   in state S:SUBJ
   setting register V to LIST
   in state S:V
   pushing to state NP:
      in state NP:
      setting register DET to  THOSE
      in state NP:DET
      setting register ADJ to  (PLANE)
      in state NP:DET
      failing from state NP:DET
      in state NP:DET
      failing from state NP:DET
      in state NP:DET
      failing from state NP:DET
      setting register N to  PLANE
      in state NP:N
      jump to state NP:END
      in state NP:END
      pop from state NP: with value (NP (N PLANE) ... )
   setting register OBJ to (NP (N PLANE) (DET THOSE) ... )
   jump to state S:OBJ
   in state S:OBJ
   jump to state S:END
   in state S:END
   pop from state S:SUBJ with value (S (VERB LIST) ... )
```

```
setting register OBJ to  (S (VERB LIST) ... )
jump to state S:END
in state S:END
pop from state TOPLEVEL with value (S (VERB WANT) ... )

(S (VERB WANT)
   (SUBJ (NP (N I) (DET NIL) (ADJ) (MODIFIERS)))
   (OBJ (S (VERB LIST)
           (SUBJ (NP (N YOU) (DET NIL) (ADJ) (MODIFIERS)))
           (OBJ (NP (N PLANE) (DET THOSE) (ADJ) (MODIFIERS)))
           (INDOBJ NIL)
           (PPS NIL)))
   (INDOBJ NIL)
   (PPS NIL))
```

8.0 REFERENCES

[Bate78] Bates, M., "The Theory and Practice of Augmented Transition Network Grammars", in Natural Language Communication with Computers, Leonard Bolc (ed.), Springer Verlag, 1978.

[Bobr69] Bobrow, D., "An Augmented State Transition Network Analysis Procedure", in Proceedings of the First International Joint Conference on Artificial Intelligence, 1969.

[Burt76a] Burton, R. and Brown, J., "Multiple Representation of Knowledge for World Reasoning" in 'Representation and Understanding', Bobrow and Collins (eds.), Academic Press, 1975.

[Burt76b] Burton, R., "Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems", BBN Report No. 3453, Bolt Beranek and Newman, 1976.

[Fini77a] Finin, T.W., "An Interpreter and Compiler for Augmented Transition Networks", Report T-48, Coordinated Science Laboratory, University of Illinois, 1977.

[Fini77b] Finin, T.W. and Hadden, G., "Augmenting ATNs", Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977.

[Fini78] Finin, T.W., "Casting the RENDEZVOUS Analyzer Rules into Augmented Transition Network Form", IBM Research Report RJ 2146(29409), IBM RESEARCH LAB., San Jose, CA, 1978.

[Gabr75] Gabriel, R.P and Finin, T.W., "The LISP Editor", Working Paper 1, Advanced Automation Group, Coordinated Science Lab, University of Illinois, 1975

[Kay 75] Kay, M., "Syntactic Processing and Functional Sentence Perspective", Proceedings of the Conference on 'Theoretical Issues in Natural Language Processing', June 1975.

[Loza76] Lozano-Perez, T., "Parsing Intensity Profiles", M.I.T. AI Memo 329, May 1975.

[Moon74] Moon, D.A., "MacLISP Reference Manual", Project MAC, M.I.T., 1974.

[Simm73] Simmons, R.F., "Semantic Networks: Their Computation and Use for Understanding English Sentences" in 'Computer Models of Thought and Language', R.C. Shank and K.M. Colby (Eds.), Freeman and Co, 1973

[Thor68] Thorne, J.P., Bratley, P. and Dewar, H., "The Syntactic Analysis of English by Machine" in Machine Intelligence 3, Donald Michie (ed.), 1968.

[Walt75] Waltz, D.L., "Natural Language Access to a Large Data Base: an Engineering Approach", Advanced Papers of the Fourth International Joint Conference on Artificial Intelligence,1975.

[Walt76] Waltz, D.L., et. al., "The PLANES System: Natural Language Access to a Large Data Base", Technical Report T-34, Coordinated Science Laboratory, University of Illinois, 1976.

[Walt78] "An English Language QuestionAnswering System for a Large Relational Data Base", CACM vol. 21, July 1978.

[Wood70] Woods, W.A., "Transition Network Grammars for Natural Language Analysis", Communications of the ACM, October 1970.

[Wood72] Woods, W.A., Kaplan, R.M., Nash-Webber, The Lunar Sciences Natural Language Information System: Final report, Bolt Beranek and Newman report No. 2378, June 1972.