20(P)

Massachusetts Institute of Technology

Artificial Intelligence Laboratory

AI Memo 462

A Comparison of PARSIFAL with Augmented Transition Networks

William R. Swartout

March 1978

This paper compares Marcus' parser, PARSIFAL with Woods' Augmented Transition Network (ATN) parser. In particular, the paper examines the two parsers in light of Marcus' Determinism Hypothesis. An overview of each parser is presented. Following that, the Determinism Hypothesis is examined in detail. A method for transforming the PARSIFAL grammar rules into the ATN formalism is outlined. This transformation shows some of the fundamental differences between PARSIFAL and ATN parsers, and the nature of the hypotheses used in PARSIFAL. Finally, the principle of least commitment is proposed as an alternative to the Determinism Hypothesis.

This investigation was supported (in part) by National Institutes of Health Grant No. 1 P41 RR 01096-01 from the Division of Research Resources.

1. An Overview of ATN Parsers

At first glance, an Augmented Transition Network looks much like a finite state machine: the network has a number of states (or nodes) connected together by arcs. The arcs have tests and actions associated with them which indicate whether or not the arc may be traversed to a new state and what to do if the arc is traversed. Indeed, the ATN formalism *is* similar to a finite state machine, but one which has been augmented in three important ways:

1) Labels on the arcs between states may represent parts of the network as well as just individual input words. A recursive mechanism is employed so that networks may be self-referencing. For example, if the non-terminal symbol NP (noun phrase) appears on an arc, the arc will be traversed only if there is an accepting path through the NP network. Furthermore, traversing the NP network may cause the NP network to recursively re-invoke itself.

2) Arbitrary tests may be placed on an arc which must be satisfied for the arc to be traversed.

3) Actions which may be placed on arcs are more extensive than those permitted in finite state machines. These actions will be executed if the arc is traversed. These actions are usually used for building structures. They may also use registers for storing partially built structures, flags, word features, and so forth.

The resulting machine has the power of a Turing machine, although its power may be reduced by placing restrictions on the types of conditions and actions permitted on arcs [Woods 1972].

1.1 Arc Types

As mentioned above, arcs are directed links which connect the various states of an ATN. Table 1 below, reproduced from [Woods 1972], shows the various types of arcs permitted. A CAT arc may be taken if the current input word belongs to the specified category.[1] A WRD arc is similar to a CAT arc except that it checks for specific words. A MEM arc is really just a short-hand notation for a number of WRD arcs. It allows a transition to be made if the input word is one of a specified list of words. A TST arc is traversed only if the specified test is satisfied. All of the above arc types (CAT, WRD, MEM, and TST) consume a word of the input string when they are taken. The other arc types (JUMP, PUSH, POP,

1. Recall that any tests placed on the arc must also be satisfied before the arc may be traversed.

and VIR) do not. In fact, the only difference between a JUMP arc and a TST arc is that the TST arc consumes an input word.

The PUSH and POP arcs are part of the recursive mechanism of the parser. A PUSH arc may be traversed if the state it names consumes a substring of the input starting at the current position. When a PUSH arc is taken, the ATN interpreter is recursively invoked. A POP arc is the way of returning from a PUSH. It can also be thought of as indicating the acceptance of some substring. After the return, the consumed substring will have been removed from the input. The pre-actions associated with a PUSH arc indicate actions to be taken before the recursive call is made. Usually, the pre-actions are SENDRs (described below) used to pass arguments to the lower level. The VIR arc will be described below in conjunction with the HOLD action.

```
(WRD     <WORD>       <TEST>      <ACTION>*)
(MEM     <(WORD*)>  <TEST>  <ACTION>*)
(CAT     <CATEGORY><TEST>      <ACTION>*)
(TST     <LABEL>      <TEST>      <ACTION>*)
(PUSH    <STATE>      <TEST>      <PREACTIONS><ACTION>*)
(POP     <FORM>       <ACTION>*)
(VIR     <CATEGORY><TEST>      <ACTION>*)
(JUMP    <STATE>      <TEST>      <ACTION>*)
```

Table 1. Types of Arcs[2]

```
(SETR    <REGISTER><FORM>)
(SENDR   <REGISTER><FORM>)
(LIFTR   <REGISTER><FORM> <WHERE>)
(HOLD    <FORM>)
(TO      <STATE>)
(SETF    <FEATURE> <FORM>)
```

Table 2. Types of Actions

2. The symbol * is the Kleene star, indicating that that element may be repeated zero or more times.

1.2 Actions

Table 2 lists the different types of actions that are available. SETR sets a register to the value of a form. SENDR and LIFTR are used in conjunction with the recursive mechanism. SENDR is similar to SETR, but rather than setting the register at the current level, it sets the register one level down. LIFTR is the same except that it sets higher-level registers. The "where" part of the LIFTR action is used to specify at what level the register should be set. SENDR and LIFTR are used for passing arguments between levels. TO is the action which indicates the target state of the transition.

The HOLD action is used together with the VIR arc to deal with what is called left extraposition in generative grammars. This phenomenon occurs when some constituent (usually a noun phrase) of an embedded construction is moved up and to the left in surface structure. Woods [1972] gives as an example the sentence, "This is the hat that I told Mary to find somebody to get a shovel to bury." The problem in trying to analyze such constructions with a left-to-right parser is that when the parser encounters the constituent, it can't tell where it belongs (because it hasn't parsed that portion of the sentence yet) even though it may know that the constituent is out of place. This problem was solved by the development of a hold list. The HOLD action allows the parser to put a constituent which has been found out of place on the hold list. As the parser continues to parse the sentence, it will come across the place where the constituent would have been, had it not been extraposed from its position in deep structure. A VIR arc taken at this point will retrieve the constituent from the hold list and treat the constituent as if it had occurred at the current position.

Burton [1976] introduces an additional useful notion: the feature register. Many tests of grammaticality use the features of structured consituents: determining subject-verb agreement is one example. In the original ATN formalism, features were only associated with words; features associated with structures had to be built into the structures themselves. Feature registers allow the separation of structural information from features, allowing the grammar to be more perspicuous. The action SETF is used to set the feature register.

```
(GETR     <REGISTER>)
*
(GETF     <FORM>     <FEATURE>)
(BUILDQ   <FRAGMENT><REGISTER>*)
(LIST     <FORM>*)
(APPEND   <FORM>     <FORM>)
(QUOTE    <ARBITRARY STRUCTURE>)
```

Table 3. Types of Forms

1.3 Forms

Table 3 lists the types of forms to be described here. GETR is used to retrieve values from registers. The "where" component may optionally be used to indicate a register not at the current level. The meaning of the * form depends upon the context of its use. When used in a WRD or MEM arc or in a test condition of a JUMP, POP, or PUSH arc, the value of * is the current input word. During the actions of a PUSH arc, * refers to the result returned by the lower computation. During both the conditions and actions of a VIR arc, * refers to the thing being retrieved from the hold list. GETF is a function which returns features of the value of a form.

BUILDQ is a function used for building the tree structures which represent the parser's analysis of the sentence. The arguments to BUILDQ are a tree structure template with specially marked leaves and a list of registers. BUILDQ constructs structures by substituting the values of the registers for the specially marked leaves in the template. Woods gives as an example (BUILDQ (S + (VP + +)) SUBJ V OBJ) which would produce:

```
(S (NP (NPR (JOHN))
   (VP (V LIKES)
       (NP (NPR MARY))))
```

if the contents of the registers SUBJ, V, and OBJ were (NP (NPR JOHN)), (V LIKES), and (NP (NPR MARY)), respectively. Burton [1976] lists a few additional forms in his description of an ATN: LIST, APPEND, and QUOTE. These forms are similar to their counterparts in LISP.

1.3.1 The Backup Mechanism

Many implementations of ATNs use an automatic backup mechanism. In parsing a sentence, the parser may find itself in a state where several of the links leaving that state may be traversed because all of their tests succeed. If the links are ordered, the parser takes the one with highest priority; otherwise, it may choose arbitrarily. Of course, there is no guarantee that the choice it makes is the correct one. If the choice is incorrect, the parser will be blocked at some later time in the parsing process. When this situation occurs, the parser backs up into the states it has been in previously, trying alternative paths, until a successful path is found.

The process described above produces a depth-first search of the possible parses. By simulating parallel processing, it is possible to produce breadth-first search. Whenever there are multiple arcs leaving a node, each of the arcs is traversed by its own processor, and these processors continue to move through the network. Sooner or later, if there is a correct parse for the input sentence, one of them will find it. If we are only interested in that parse, we can stop the other processors, otherwise, we can continue to let them run to find other possible parses.

2. An Overview of PARSIFAL

In his introduction, Marcus [1978] claims that "all current natural language parsers that are adequate to cover a wide range of syntactic constructions operate by simulating non-deterministic machines, either by using backtracking or by pseudo-parallelism." PARSIFAL, a reaction against this, represents an effort to parse English "strictly deterministically." The concept of strict determinism will be dealt with later in the section on the Determinism Hypothesis; this section will describe the major mechanisms of the parser.

There are three major components to the PARSIFAL parser:

-An *active node stack* used for holding structures while they are being completed

-A set of *pattern/action rules* organized into packets and used for controlling the analysis process.

-A *buffer* of finite length. The patterns of the rules are matched against the contents of the buffer and the words of the input sentence enter through it. The buffer gives the parser a look-ahead capability.

These elements will be described in detail below.

2.1 The Active Node Stack

The parser uses a stack to hold incomplete constituents while they are being completed. The stack is the parser mechanism for dealing with the recursive properties of natural language. The parser pushes incomplete constituents (or nodes) onto the stack while it is working on that node's daughters. After a node is completed, it is popped from the stack, and the parser continues to work on the superior node. At all times, the parser has access to only two nodes on the stack: the bottom node (thinking of the stack as growing downward), also called the current active node, and the S or NP node closest to the bottom of the stack which is called the *dominating cyclic node*, or sometimes just the *current cyclic node.* When a node is popped from the stack, it is inserted into the buffer (see below) if it is not attached to a node higher in the stack.

The completed structure of nodes represents the analysis produced by PARSIFAL, just as the structures created by BUILDQ are the output of the ATN parser. As with an ATN, the grammatical features of a node represent the primary grammatical properties of the constituent it represents. A sample structure (from Marcus [1978]) for the sentence "I will schedule a meeting." is shown below:

```
S20
    NP47    I
    AUX20
        WORD112     will
    VP22
        WORD113     schedule
        NP50    a meeting
    WORD116   .
```

2.2 The Buffer

Marcus [1978] states that "the constituent buffer is really the heart of the grammar interpreter; it is the central feature that distinguishes this parser from all others." The buffer may be thought of as a finite linear list of cells. Each cell may contain one node. In most cases, a buffer length of three is adequate, but for parsing noun phrases the buffer is extended to five cells. The parser matches the patterns of rules against the buffer and uses the buffer to perform look-ahead. Since the length of the buffer is limited to five cells, the parser may only look ahead at most five cells before taking some action.

There are three actions that the buffer may perform. *Read (i)* returns the value stored in the ith cell of the buffer. *Delete (i)* which deletes the item in the ith cell and then shifts all the cells numbered higher than i down by one to fill in the gap. *Insert (w,i)* inserts the item w into the ith cell after shifting the contents of the ith and higher-numbered cells up by one to create a gap.

While the simple buffer presented above is adequate for most constructions, it is convenient to introduce an offset mechanism for dealing with noun pharses. This mechanism will cause the index given to the Read, Insert, and Delete commands to be offset from the beginning of the buffer by some amount *m*. A stack called the *buffer pointer stack* is used in conjunction with these offsets. The stack may be pushed or popped, and the offset at the bottom of the stack (thinking of the stack as growing down) is the one currently in effect.

2.3 The Grammar Rules

PARSIFAL uses a set of pattern/action rules to decide what to do next. The rules are organized into *packets*. If a packet is active, then the patterns of the rules in that packet are examined to see whether the action part of the rule should be taken. More than one packet may be active at any time. Usually, the names of the packets suggest their function. Typical packet names are: PARSE-AUX, PARSE-SUBJ, BUILD-AUX, and PARSE-VP. Rules also have a numerical priority associated with them. At each point in the parsing process, the highest priority rule with a matching pattern is the one executed[3]. If two or more rules with equal priority match, the parser is free to choose among them.

3. Marcus suggests that it would be possible to represent the priority of a rule by a set of rule pairs that indicate which rules have priority over others. It might be necessary to use some sort of hierarchy if some rules are discovered which cannot fit into the global partial ordering imposed by the numerical priorities of the rules. An example would be a rule whose priority changes depending on the set of active rules. Apparently, this situation has not yet arisen.

The pattern of a grammar rule consists of a list of partial descriptions of parse nodes. Each partial description must match for the entire pattern match to succeed. There may be as many as five partial descriptions in each pattern: one for each of the three buffer positions, and one each for the current active node and the current cyclic node, the two accessible stack nodes. The descriptions are Boolean combinations of tests of grammatical features. It is also possible to check the features of the daughters of a node using tree walking notation provided by the grammar language.

If there are not enough symbols currently in the buffer to match against a pattern, the grammar interpreter will automatically place items from the input string into consecutive buffer cells until the buffer is full enough so that a match may proceed. Interestingly, this is the *only* mechanism the parser has for reading the input string.

Marcus [1978] lists a number of different types of actions that may be explicitly specified by a grammar rule:4

-Create a new parse node.

-Insert a specific lexical item into a specific buffer cell. For example, when the the parser recognizes an imperitive it inserts the word "you" into the buffer to serve as the subject. After doing that, the sentence may be parsed as if it were a normal declarative.

-Attach a newly created node or a node in the buffer to the current active node or the current cyclic node, causing the grammar interpreter to remove the node from the buffer.

-Pop the current active node from the stack. If it is not yet attached anywhere, it will be inserted into the first position of the buffer.

-Assign features to a node in the buffer or to one of the accessible nodes in the stack.

-Activate and deactivate packets of rules.

The actions possible within PARSIFAL's rules are considerably more constrained than those of an ATN. In particular, the following specific restrictions (paraphrased here) are mentioned by Marcus:

-The value of the buffer cells or the accessible stack nodes may not be set by rule actions. (Note that the values of the accessible stack nodes do change during rule execution, but their values cannot be set arbitrarily by a rule action.)

4. Some actions are specified implicitly and will be described later.

-No user-defined funtions are allowed.

-There is no recursion or iteration within actions.

-The only structure building operations in rule actions are a) attaching one node to another, and b) adding features to a node's feature set.

There are a few subtle actions that should be mentioned. These actions are implicit, that is, they are not explicitly ordered by the actions of the rules, but they occur indirectly when certain explicit actions are taken. As mentioned earlier, the reading of the input string symbols into the buffer is an implicit process. Symbols are read into the buffer as needed to match the patterns of rules. Another point involves the activation and de-activation of rule packets. While packets may be explicitly activated or de-activated by rule actions, packets are thought of as being active with respect to the current active node. Thus, when a new node is created, the packets associated with the previous node are no longer active. Those packets will be re-activated when the new node is popped off the stack, so that the old node becomes the currently active node once again.

3. The Determinism Hypothesis

The central theme of this paper involves the Determinism Hypothesis presented in Marcus' thesis. Marcus claims that it is the underlying motivation for the structure of his parser. Unfortunately, it is a little difficult to pin down exactly what the Determinism Hypothesis is because the definition of determinism is non-standard[5] and somewhat vague. Futhermore, the Determinism Hypothesis itself changes in Chapter 9. I will attempt to characterize what the Determinism Hypothesis entails (by quoting Marcus), and then outline a number of interesting questions the hypothesis raises.

Marcus formulates the Determinism Hypothesis as follows:

5. A more standard informal definition of determinism (paraphrased from Kain [1972]) is: A machine is deterministic if at every step in the machine's operation the input symbol and current state uniquely determine the next state. He also defines a non-deterministic Turing machine as one in which there may be several possible actions for any state-symbol combination. Note that if we adopt this definition, then it is possible for Marcus's parser to be non-deterministic since he does not specifically disallow the situation where two rules having equal priorities both have patterns that match successfully [Marcus 1978, page 90]

> Natural language can be parsed by a mechanism that operates "strictly deterministically" in that it does not simulate a non-deterministic machine...

He goes on to state:

> Rather than attempting to formulate any rigorous, general explanation of what it means to "not simulate a non-deterministic machine", I will focus instead on several specific properties of the grammar interpreter which will be the focus of this paper. These properties are special in that they will block this interpreter from simulating non-determinism by blocking the implementation of either backtracking or pseudo-parallelism.

Next he lists three properties of the interpreter to give the reader a feeling for the necessary conditions for strict non-determinism. First, the structures created by the parser are permanent; they may not be erased. Second, all syntactic substructures created by the machine for a given input must be output as part of the structure assigned to that input. That is, once something has been created it can't be ignored or discarded. Third, temporary syntactic structures may not be hidden in the internal state of the machine.

The Determinism Hypothesis raises a number of questions. Could an ATN interpreter be used to parse deterministically like PARSIFAL? Do people seem to parse deterministically? What seem to be the trade-offs between non-deterministic and deterministic parsing? Is the Determinism Hypothesis really the best way to view Marcus' parser? I will attempt to shed some light on these questions below.

3.1 Can an ATN Interpreter Parse Like PARSIFAL?

In this section, I will show how it is possible to use a mechanistic (and quite simple) transformation to turn the rules of Marcus' parser into an ATN. Of course, once we know that an ATN parser has the power of a Turing machine it should come as no great surprise that it is possible to use it to simulate PARSIFAL. The point of the simulation is not to show that it *can* be done, but rather to show *how* it's done. Some parts of PARSIFAL may be naturally transformed into the ATN formalism, while the simulation of others is somewhat awkward. The differences between things which are easy and difficult to simulate may be reflected in the differences between the grammars produced for the two formalisms. Those features of PARSIFAL which are easy to simulate will tend to be found in both grammars, while features which are difficult will tend to be employed in PARSIFAL alone. By comparing what's easy and what's hard to simulate we can perhaps gain some insight into the real differences between PARSIFAL and ATN parsers.

The simulation will be done in several stages. To start, only the 3 cell version of Marcus' parser will be dealt with (the version that assumes noun pharses enter pre-parsed). We will simulate this version of his parser with a one-state ATN. This simulation is presented first since it is fairly easy to explain. However, it doesn't look much like a conventional ATN (we've taken the N out of ATN). Next, we will turn this one-state ATN into a more conventional multi-state ATN. This transformation should provide some insight into the hypotheses used by Marcus' parser. Finally, we will extend the discussion to include the full form of his parser. We will proceed by showing how the various data structures of Marcus' parser may be simulated within the ATN formalism.

3.1.1 A One-State ATN to Simulate PARSIFAL

Our first simulation of PARSIFAL will use a one-state ATN. To simulate the buffer, we will use three special registers. The rules of the grammar will be looping arcs which perform actions and return to the single state. As suggested by Marcus, the rule packet mechanism can be simulated by associating a register with each rule packet. If a register has a non-zero value then the packet associated with that register is considered active. Finally, the active node stack may be simulated by the ATN recursive mechanism. These ideas will be made more precise below.

3.1.2 Simulating the Buffer

The buffer will be simulated by three special registers: CELL1, CELL2, and CELL3. These correspond to the three cells of the buffer. The buffer operation of Read (i) can be directly simulated by the ATN GETR form. The operations Insert (i) and Delete (i) can also be easily simulated by the appropriate use of GETRs and SETRs. How things can be attached to the active node stack and dropped into the buffer will be dealt with when the simulation of the stack is described. Simulating the implicit PARSIFAL operation of reading the input string into the buffer is a little more complex.

To simulate the way the input symbols are placed in the buffer, a TST arc is placed before each rule arc which checks to see that all the buffer cells examined by the following rule arc are filled[6]. If not, then the TST arc is taken, and it fills the cell registers with symbols from the input string. Thus, if a rule arc examines only the first buffer cell, it is only necessary to have one TST rule preceeding it to fill the (potentially empty) first cell. If a rule examines all three cells, it would be necessary to have three TST arcs, one for each cell.[7]

3.1.3 Arcs as Rules

The rules of the PARSIFAL grammar may be represented as arcs. Those rules that do not involve pushing or popping the active node stack may be simulated as JUMP arcs, while those that do may be PUSH or POP arcs (they will be dealt with later). The priority of the rules may be reflected in the ordering of the arcs[8]. As suggested above, the packet mechanism may be simulated by associating registers with packets and then placing a conditional clause in the test portion of the arcs repesenting a packet of rules to assure that an arc is traversed only if the register associated with the packet is non-zero (that is, only if the packet is active). Simulating the pattern portion of a rule is not difficult within the ATN formalism, because it allows arbitrary tests to be placed on an arc.

There are various rule actions that we must also be able to simulate:

-Inserting a specific lexical item into a specific buffer cell may easily be done by setting the appropriate buffer cell after moving its old contents to the appropriate buffer position.

-Activating and deactivating packets of rules may be done by setting the register associated with the rule packet.

-Assigning features to a node may be done by using the feature register associated with that node.

The simulation of the remaining operations (such as creating a new parse node, attaching a node to the stack, and popping the stack) are discussed in the next section.

6. The ATN formalism permits arcs to be ordered.

7. Actually, it wouldn't be necessary to have a TST arc before each rule. For example, a rule arc need not have a TST arc preceeding it if some higher-priority rule in the same packet has already checked all the cell registers used by the rule in question.

8. Note that the ordering of the rules will be independent of packet membership.

3.1.4 Simulating the Active Node Stack

To simulate the active node stack, we will use the recursive mechanism of the ATN. Using the hold list might also seem like a likely way to implement the stack, using HOLD actions to place things on the list and VIR arcs to pop them off. However, the hold list was designed to solve a very specific problem--the problem of left extraposition. Thus, it does not seem that the hold list would be a natural way to implement the active node stack.

Creating a new parse node is simulated by traversing a PUSH arc. At each level, the register NODE is used to store the contents of that parse node. When the node is complete (i.e. when PARSIFAL would pop the node off the active node stack) a POP arc is traversed. In PARSIFAL, when a node is popped off the stack, it is merely removed if it is attached to a higher level node, however, if it is not, it is dropped into the buffer. To simulate this an additional flag register is needed: NODE-ATTACHED. If the current active node becomes attached either to the node above it or to the current cyclic node, then the NODE-ATTACHED register is set using LIFTR at the level above the current active node. Upon return from a lower level, if NODE-ATTACHED is not set, the interpreter inserts the structure being returned into the buffer. Nodes may be attached by using GETR to retrieve the value of the higher level nodes; APPEND, LIST, and QUOTE to perform the structural operations; and LIFTR to replace the new structures back at higher levels.

During the pre-actions of the PUSH arc, it will be necessary to pass down certain structures using SENDR. Among these are the buffer and the set of rule packets that are to be active while parsing at the lower level. Note that by using the recursive mechanism of the ATN, packets of rules are active or inactive with respect to the currently active node, the same as they are in PARSIFAL. No additional mechanism is required to simulate PARSIFAL's implicit action of switching rule packets as nodes are popped.

3.1.5 A Multi-state ATN

We can convert the one-state ATN outlined above into a multi-state ATN as follows: First, we intoduce a new state for each of the possible combinations of active packets. In general, we will have 2^N distinct states, (where N is the number of packets), although in actual practice the number of states will be considerably smaller since the number of rule packets active at any time is usually quite small. Since the states represent all possible combinations of active rule packets, we can eliminate the rule packet registers of the one-state simulation. To switch active packets, we merely jump to the appropriate state. To create a new active node, we push to the state representing the set of rules that are to be active. To handle the buffer, each state must have a set of TST links associated with it similar to the ones described above.

The network that results from the sample grammar presented in appendix C of Marcus' thesis is shown in Figure 1. A number of details have been omitted to keep the diagram as simple as possible. The actions and tests associated with arcs are not shown, and the TST arcs used for buffer filling have been left out. All unlabelled arcs are JUMP arcs. Multiple arcs between nodes represent rules with different tests which perform different actions but which activate and deactivate the same rule packets. The SUBJ-VERB node is connected to over 20 nodes. To save space, only a few of these have been shown.

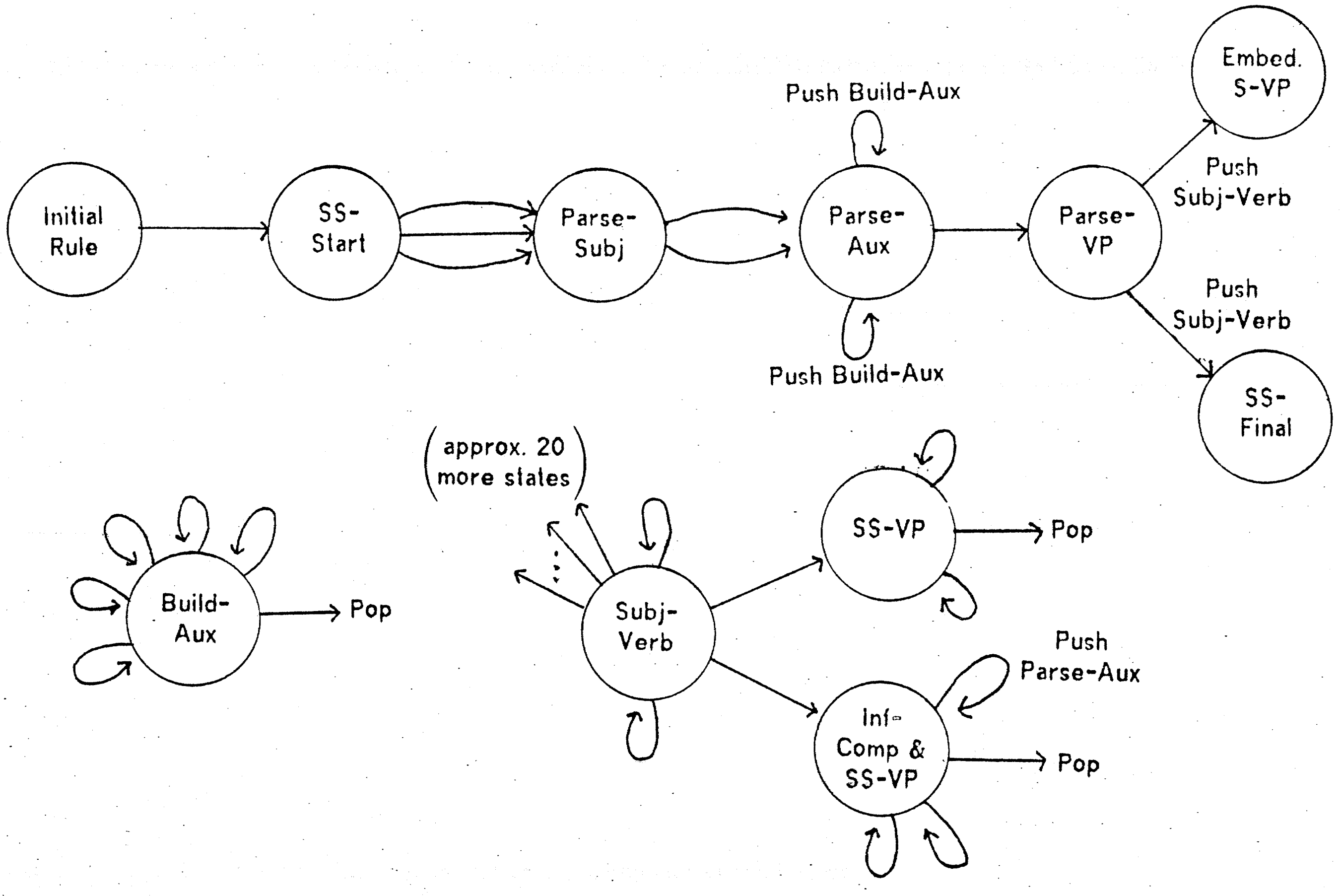

Figure 1. PARSIFAL Example Grammar as an ATN

I think the interesting thing about the multi-state ATN simulation of PARSIFAL is that it provides some insight into the nature of the hypotheses PARSIFAL uses. In Chapter 2 of his thesis, Marcus discusses hypothesis-driven parsing, and notes some of the shortcomings of that approach. A natural question to ask is whether or not PARSIFAL uses hypotheses, and if it does, what their nature is, and how they are represented. In conventional ATN parsers, the state the parser is in represents its "best guess" about what it is parsing. In the multi-state ATN simulation of PARSIFAL, the states again represent the current hypothesis, but the fact that the states of the ATN are the power set of the rule packets suggests an important difference: the hypotheses that PARSIFAL has are disjunctive[9] and they are represented by the set of rule packets that are active.

9. That is, hypotheses like: "It's either *x*, *y*, or *z*." are permitted.

The hypothetical nature of the rule packets can also be seen by examining the times when packets are active. For example, the packet PARSE-AUX is activated when parsing auxilliaries, PARSE-VP is active when a verb phrase is expected, and PARSE-SUBJ is used to remove the noun phrase subjects of various types of clauses. If several things are possible, but the parser can't determine with certainty which one will occur, it can activate packets which will handle all the possible cases. Of course, the grammar writer must take care that undesirable rule interactions do not occur as a result of having several packets active at once. PARSIFAL can also easily broaden or narrow a hypothesis by activating or de-activating rule packets. Although it's possible to implement this type of hypothesis within the ATN framework, I think it is fair to say that it's inconvenient, and that PARSIFAL can represent disjunctive hypotheses elegantly and easily.

3.1.6 Offsets

The sections above outline a method for simulating the version of PARSIFAL presented in the first four chapters of Marcus' thesis. To implement noun phrase parsing, we need to be able to offset the buffer. This can be easily done if we allow the registers in the ATN to be indexed and if we allow the addition of a stack to hold the offset pointers[10]. These do not seem to be unreasonable extensions to the ATN formalism. By making simple modifications to buffer handling (to take the indexing into account) and by implementing simple routines to push and pop the index stack, the Attention Shifting rules may be implemented.

3.1.7 Discussion

The above simulation of PARSIFAL may be viewed in two lights. On the one hand, it suggests that ATNs allow the grammar designer much more freedom of expression than PARSIFAL rules. After all, it was possible to simulate PARSIFAL with only two minor extensions to the ATN formalism, while simulating ATNs with PARSIFAL would be much more difficult, requiring several major additions to Marcus' parser

10. Without these extensions, the implementation is much more painful, although it can be done, by adding more tests.

(such as backup and recursion, to name a few). On the other hand, great expressive freedom is not always a good thing. By constraining the grammar design, one may gain insights into the characteristics of natural languages. Marcus already has some preliminary evidence for a theory of garden path sentences. His model of a three cell buffer seems to have some power to predict those sentences that people will have trouble parsing. It is difficult to see how ATNs (as conventionally implemented) could make such a prediction.

The simulation lends support to Marcus' claim that the buffer is the most unique thing about his parser. The buffer does not fit neatly into the ATN formalism: special arcs must be added to fill it, the three special cell registers must be passed laboriously up and down between levels, and special hacks must be placed on the actions of arcs to handle the insert and delete operations. The ATN formalism does not seem to encourage one to structure his grammar around a finite-length buffer.

To change the topic slightly, suppose that we did not want to simulate PARSIFAL, but instead wished to implement a more conventional ATN using less backup--how could it be done? One way suggested by Burton [1976] is to structure the grammar so that deciding between similar structures is postponed as long as possible. Another possible way to decrease backup would be to adopt the notion of a differential diagnosis link from medicine [Rubin 1975][11]. When a physician finds that the hypothesis he originally chose is incorrect, he will often be able to go to the correct hypothesis directly. This is because he knows about diseases whose symptoms closely mimic those of his original hypothesis. Thus, if he finds he cannot confirm his original hypothesis, he will begin to think about similar diseases. Usually there is a specific test to determine which similar disease he should think about. One could imagine employing a similar sort of mechanism for parsing natural language. If the parser found that it could not proceed, and there was a differential link leaving the state it could traverse the link (assuming tests on the link were satisfied) and enter a new (and hopefully correct) state. As the link was traversed, it would probably be necessary to adjust some of the registers to reflect the fact that the basic hypothesis was changing, but this would have to be done on a case by case basis.

11. Marcus has also adopted this idea to deal with garden path sentences. Patil has referred to this use of differential diagnosis as "side-stepping".

3.2 The Determinism Hypothesis: The Correct View of PARSIFAL?

Marcus proposes the Determinism Hypothesis as the underlying motivation for the structure of PARSIFAL. Yet its definition is vague and somewhat non-standard. Additionally, the constraints on control structure and structure-building operations are confounded to some degree in the thesis and should be separated.

I would propose that Marcus parser is motivated by adherence to the "principle of least commitment" with respect to both control and structure-building operations. The principle of least commitment [Marr 1975, Miller and Goldstein 1976] states that a system should never do something that may later have to be undone. The structure building operations of PARSIFAL reflect this principle since structures may not be destroyed once created, although the existence of garden path sentences suggests that it may not be possible to adhere to it entirely. However, adherence to the principle with respect to structure building operations alone is not sufficient to constrain the design of the parser as tightly as Marcus wishes. As an example, the semantic grammar parser of Burton [1976] never creates structures it will have to destroy, but it does this by scouting ahead in the ATN to see what will happen before it commits itself to a structure. To avoid this problem, we can require adherence to the principle of least commitment with respect to control. That is, a process should not be committed to a particular path if it may become necessary to back out, or if the path may lead to a dead end. I think this viewpoint gives a somewhat clearer view of the motivation for PARSIFAL.

In viewing PARSIFAL, it should also be realized that a trade-off exists between the complexity of the algorithm and the degree to which the principle of least commitment may be followed. Marcus' experiments with garden path sentences suggest this[12]. While a parser with a buffer of 3 cells will make some commitments too early, and be taken down the garden path, a parser with 4 cells will handle more sentences correctly, and five will handle still more correctly. Thus, while a parser with a buffer of 5 cells will parse more sentences deterministically, the increased buffer length is the cost.)

12. A longer buffer implies a more complex algorithm.

4. Conclusions

In this paper, I have compared Marcus' PARSIFAL and Woods' ATN. I have outlined a method for representing PARSIFAL as an ATN. I have suggested that the underlying motive for Marcus' parser is perhaps better viewed as the principle of least commitment rather than the Determinism Hypothesis.

This paper may seem somewhat critical of PARSIFAL. That is not my intent. I feel that PARSIFAL represents a significant advance over previous parsers, because it actually uses the principle of least commitment and disjunctive hypotheses to a far greater extent than previous parsers, even though it would have been possible (in principle) to do the same things in earlier formalisms.

5. References

Burton, Richard R. "Semantic Grammar: an Engineering Technique for Constructing Natural Language Understanding Systems", BBN Report 3453, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, 1976

Kain, Richard Y. *Automata Theory: Machines and Languages* McGraw-Hill Book Company, 1972

Marcus, Mitchell Philip "A Theory of Syntactic Recognition for Natural Language" MIT, 1978

Marr, D. "Early Processing of Visual Information" MIT-AIM 340, 1975

Miller, M. L. and Goldstein, I. P. "PAZATN: A Linguistic Approach to Automatic Analysis of Elementary Programming Protocols" MIT-AIM 388, 1976

Rubin, Ann D. "Hypothesis Formation and Evaluation in Medical Diagnosis", AI-TR-316, 1975

Woods, William A. "An Experimental Parsing System for Transition Network Grammars", BBN Report 2362, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, 1972