1983

Parsing natural language

Leonard E. Wilcox

Parsing

Natural Language

by

Leonard E. Wilcox, Jr.

This thesis is submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science at Rochester Institute of Technology, January 27, 1983.

Approved by: _____
Professor Peter G. Anderson

_____
Professor John A. Biles

_____
Professor John L. Ellis

I hereby grant permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Leonard E. Wilcox, Jr.

February 11, 1983

Abstract

People have long been intrigued by the possibility of using a computer to "understand" natural language. Most researchers attempting to solve this problem have begun their efforts by trying to have the computer recognize the underlying syntactic form (the parse tree) of the sentence.

This thesis presents an overview of the history of syntactic parsing of natural language, and it compares the major methods that have been used. Linguistically, two recent grammars are described: transformational grammar and systemic grammar. Computationally, three parsing strategies are described and compared: top-down parsing, bottom-up parsing, and a combination of both of these methods. Several important natural language systems are described, including Woods' LUNAR program, Winograd's SHRDLU, and Marcus' PARSIFAL.

Keywords: natural language, computational linguistics, parsing, syntax, transformational grammar, systemic grammar, augmented transition network, ATN, Earley's algorithm, Cocke-Younger-Kasami algorithm.

Contents

CHAPTER 1

Introduction

Since the advent of computers, many people have been interested in using computers for the analysis of natural language. People have long believed that language is one of man's highest functions, and one which separates man from other animals. Researchers in artificial intelligence believe that if computers can be programmed to analyze sentences and stories in much the same way that man does, then that may well indicate that computers can perform some of man's other mental functions as well. Simulating language comprehension on a computer may also provide some other insights into how man functions.

The problem of natural language comprehension by computer did not succumb to the efforts of the early researchers in artificial intelligence. As a matter of fact, it is still a field that is being actively investigated. There are people now who do not believe that a computer will ever be able to "understand" natural language in a way that is at all related to human comprehension (Dreyfus, 1979; Weizenbaum, 1976).

One of the subordinate problems of natural language comprehension is that of syntactic parsing. Can a computer accept a sentence in English (or some other natural language) and recognize its grammatical form?

This particular natural language problem has received much study, and various solutions have been proposed. Some of the computer systems developed for natural language parsing have been performance oriented, and others have been presented as answers to linguistic and psychological questions.

The following chapters trace some of the main events in the history of syntactic parsing of natural language by computer. Two important aspects of the problem are examined in parallel.

First, what have been the motivating linguistic concerns? What grammars were used? Why were they chosen?

Second, what parsing strategies were employed? How did these strategies evolve and improve over the years? Did the parsers in any way model the way humans parse sentences?

This thesis examines and compares several of the important natural language parsing systems developed within the last twenty years. Two grammars (transformational and systemic) are described, as well as parsers that are top-down, bottom-up, and a combination of both of these methods.

CHAPTER 2

Chomsky grammar types

Chomsky established a hierarchy of formal grammars (Chomsky, 1959) that provides a useful framework in which to examine certain aspects of natural language. We will be most concerned with 'type 2', or context-free grammars.

Given that a formal grammar is a 4-tuple,

G = (N, <sigma>, P, S), where

N	= the set of non-terminal symbols,
<Sigma>	= the set of terminal symbols,
P	= the set of production rules, and
S	= the Sentence symbol (or Start symbol).

Chomsky grammars are of the following four types (Aho and Ullman, 1972; Griebach, 1981):

type 0: unrestricted grammars

Production rules are of the form

<alpha> --> <beta>, where <alpha> and <beta> may contain any combination of terminal or non-terminal symbols.

type 1: context-sensitive grammars

Production rules may be of the form

<alpha> --> <beta>, where <alpha> contains at least one non-

terminal, <beta> may be any combination of terminals and non-terminals, and <beta> must be at least as long as <alpha>. This prevents having a grammar that shrinks, expands, shrinks, expands, etc. Applying productions always will cause the right side of the derivation to stay the same size or expand.

An alternative way to write a production in a context-sensitive grammar is:

a X b --> a Y b, where

a and b are terminals,

X is a non-terminal, and Y is a

non-empty combination of terminals

and non-terminals.

type 2: context-free grammars

All productions are of the form A --> <beta>, where <beta> is composed of terminals and non-terminals.

Many programming languages are described by context-free grammars. It is generally agreed that natural language can not be represented adequately in a context-free setting.

type 3: right linear (or regular) grammars

All productions are of the form:

A --> a B , or

A --> a, where A, B

are non-terminals, and

'a' is a terminal.

Many of the early attempts at language translation on computers were based on context-free grammars. The next section describes one noteworthy example.

CHAPTER 3

The multiple path syntactic analyzer

This system was developed at Harvard (Kuno, 1962) and produced all possible parses of a sentence. Every word in the input string was looked up in a dictionary, and all possible syntactic interpretations of the word were recorded. As a word was parsed into the sentence, each syntactic use of the word was tried with every one of the currently active parse trees. Each one of these potentially acceptable parses was maintained as a new possible parse tree and stored in an area called the 'prediction pool'. The process continued until each parse tree either failed or accepted the entire sentence.

The rules of the grammar were stored in a 'grammar table' which tried to match rules with the possible syntactic categories of the curently active word.

Consider the sentence :

"They are driving to Texas."

The first word is unambiguously a pronoun. The initial prediction made is that the input string is a 'sentence'. The grammar rules are of the form:

G(c,s) = <form of the rest of the sentence>

where c = the syntactic part currently sought, and

s = the syntactic category of the current word.

As an example, there were eight rules that could be used to start a sentence with a pronoun. They included:

G_1(sentence, pronoun) = <predicate> <period>

e.g., "They moved."

G_2(sentence, pronoun) = <adjective clause> <predicate> <period>

e.g., "They who desired a promotion moved."

. .

. .

. .

G_8(sentence, pronoun) = <comma> <subject phrase> <comma> ... etc.

e.g., "They, the people of Canada, ... etc."

These eight rules were stored in the 'prediction pool', a push-down store of all the currently active alternatives. When the next word, 'are', was read, it could be interpreted as either an intransitive verb ("They are.") or an auxiliary verb ("They are eating dinner.").

In each of the eight rules in the prediction pool, the first item in the right side of the rule was the new syntactic category being searched for, and each of these paired up with all the possible syntactic uses of 'are'. This produced sixteen new combinations, namely:

G_1(predicate, intrans. verb) = ...

G_2(predicate, auxiliary verb) = ...

G_3(adj. clause, intrans. verb) = Null (no rule applies)

G_4)adj. clause, auxiliary verb) = Null

. . .

. .

. .

G_{15}(comma, intrans. verb) = Null

G_{16}(comma, auxiliary verb) = Null

Most of these possibilities would not work, and so they were discarded right away. This process was repeated until the last word in the sentence had been accepted.

The grammar that guided the parsing was equivalent to a context-free grammar. Linguistically, it was a traditional descriptive grammar that provided parse trees of the surface structure of the sentence; i.e., it only stated word categories and intermediate constituents.

One of the shortcomings of using a context-free grammar for natural language analysis becomes apparent in this system. The grammar was unwieldy. There were over 3400 rules in the grammar table, due at least in part to a failure to recognize and deal with some of the regularities of natural language, such as embedded clauses.

The parsing algorithm could become very time-consuming, particularly on sentences where the words (when taken out of context, as this system always took them) were syntactically ambiguous. The algorithm could no doubt have been speeded up if only a single (most likely) parse had been produced, instead of all possible parses. Generating all the possible parses of a sentence led to some peculiar interpretations of sentences, especially since there was no semantic component to rule out some of the

really bizarre parses. One famous example is reported by Bertram Raphael (Raphael, 1976). The sentence "Time flies like an arrow" was input to the syntactic analyzer, and four valid parses were generated:

(1) The word 'time' was treated as a verb. According to this parse, a person might 'time' (perhaps using a stop watch) flies in exactly the same way that he would clock an arrow.

(2) In this parse, the speed is measured of only those flies that happen to be similar to arrows.

(3) Treat 'time' as a noun modifying another noun - "time flies" - the same construction as "mosquito net". At any rate, this unusual type of fly (the time fly) is quite fond of arrows.

(4) Time moves along just the same way an arrow does. This is the correct parse.

This was no doubt a difficult sentence to parse. If the words are examined out of context (in this case, 'context' can be considered to be our own world knowledge), they are syntactically ambiguous. It cannot be determined if 'time' is to be treated as a noun, adjective, or verb just by having the word 'time'. Knowing the context is critical.

In that sense, this was realy a tour de force parsing performance. From a practical point of view, however, three of the parses make no sense. This type of example showed researchers in this field that there had to be some semantic input if translation systems ever were to be successful.

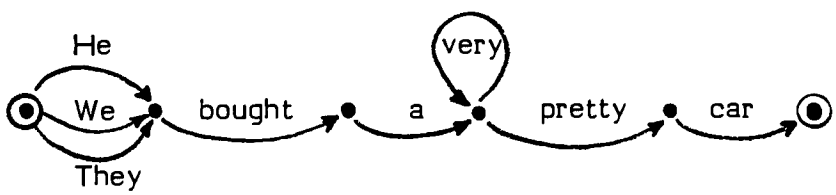
This is not to imply that syntactic parsing is unimportant. It is much less expensive than many of the complex semantic systems being developed today. A sensible goal is to do as much of the analysis as possible using syntax as a guide, and then use a semantic component to help out where appropriate.

CHAPTER 4

The inadequacy of context-free grammars

It is now generally believed that context-free grammars are not able to represent all the grammatical constructions of natural language.

This conclusion was stated by Noam Chomsky in his book Syntactic Structures, first published in 1957. He started by examining finite state automata, saying that they were the simplest model of natural language that was worth examining. For example:



This produces an infinite number of grammatical sentences, but as a model of language there are many types of sentences that it can not handle.

More abstractly, a finite state automaton can not handle these languages:

L_1 = { a^n b^n , where n >= 1 }

L_2 = { A reverse(A) } e.g., abcddcba.

L_3 = { A A } e.g., abcdabcd

To relate this to natural language, Chomsky then considered the sentential forms:

E_1 = If S_1 , then S_2 .

E_2 = Either S_3 , or S_4 .

E_3 = The man who said that S_5 , is arriving today.

Each of these sentences is broken into two interdependent elements by the comma, e.g.

If ... then

Either ... or

'man' ... 'is' (they must agree in number.)

Between any of these pairs of words we may insert a sentence. We need to temporarily halt the processing of the main sentence to undertake the processing of the embedded sentence. (We will find later that an extension of a finite state graph, namely a recursive transition network, does have the needed power for this.)

Consider the sentence:

"The man who said that [either a or b] is arriving today."

This sentence requires agreement between 'man' and 'is', and a matching 'either' and 'or'. An abstract view of this is:

man	either	or	is
a	b	b	a

This has the same 'mirror image' property that L_2 has, and therefore Chomsky concludes that natural language (specifically this construction in English) can not be represented by a finite state automaton.

The next construction that Chomsky examines as a model for natural language is phrase structure grammar. Phrase structure grammars usually are equated with context-free grammars. They are more abstract than finite state automata in that they use non-terminal symbols for intermediate representations. These non-terminals are frequently phrase markers of some sort (e.g., <sentence> --> <noun phrase> <verb phrase>).

Context-free grammars are more powerful than finite state automata. They can, for instance, generate L_1 and L_2.

$L_1 = \{ a^n b^n \}$

S --> ab
S --> a S b

$L_2 = \{ X \text{ reverse}(X) \}$

S --> a S a
S --> b S b
S --> a a
S --> b b

Whether context-free grammars are adequate to represent natural language is a question that has not been answered as resoundingly as some linguists would suggest. Papers frequently contain statements such as: 'But it is well known that context free grammars are not adequate for English' (Bates, 1978). The paper most often used to support this claim (Postal, 1964) uses a grammar for Mohawk, making it inaccessible to any but a handful of linguists. Sampson (Sampson, 1975) is a linguist who claims that Postal's data are incorrect, and gives a counter example in English.

Postal's argument is based on the { X X } language (L_3) given earlier. L_3 cannot be generated by a context-free grammar (see Aho and Ullman, 1972, p. 198). Many linguists claim that the word 'respectively' in English calls this { X X } type of construction into use. An example is:

"My brother, my children, and my wife, respectively,
sails, ride bikes, and drives."

Examining the agreement, we have:

brother	children	wife	sails	ride	drives
a	b	c	a	b	c

Sampson claims that a more natural sentence results by making all the verbs plural ("My brother, my children, and my wife, respectively, sail, ride bikes, and drive"). This would eliminate the { X X } construc-

tion, weakening the argument that a context-free grammar is inadequate for representing natural language.

Regardless of whether or not English can be formally written with all its nuances as a context-free grammar, the question of naturalness of representation arises. Chomsky claims (Chomsky, 1957, p. 34) that a language theory can be considered inadequate if:

(1) it can not model all the acceptable sentences in a language, or if it produces incorrect ones, or

(2) it succeeds, but only in an awkward or artificial way.

Simple phrase structure grammars tend to be clear and easy to follow. Complicated structures tend to lose their 'naturalness', and require derivations that are not representative of how humans might produce them. So even if a construct can be modelled using a context-free grammar, there might be a much more 'natural' way to handle it.

Toward that end, Chomsky proposed a new type of grammar, one that contained a "natural algebra of transformations having the properties that we apparently require for grammatical description." (Chomsky, 1957, p. 44). Chomsky's transformational grammar is only one approach to extending a context-free grammar so that it will be a natural representation of human acceptance of natural language. We will examine others, also.

CHAPTER 5

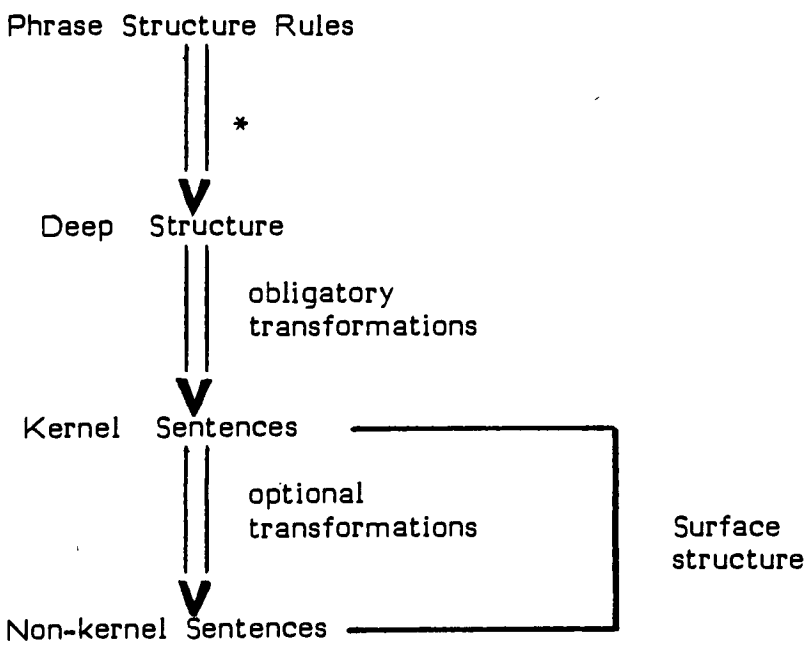
Transformational grammar

Chomsky's training in linguistics was under Harris and others in what is called the Descriptivist School (Sampson, 1980). They used constituency grammars (context-free grammars) to describe syntax and structure in English. Chomsky also had a strong background in mathematics, so perhaps it is not surprising that he combined these notions into his ideas on grammar and linguistics.

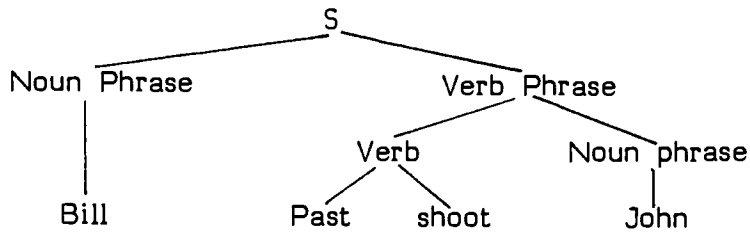
Chomsky was searching for "linguistic universals", or ideas and concepts that could be found in all languages. He believed that the core, or set of simple essential sentences, could be generated by a phrase structure grammar in any language. He called these sentences the deep structure representations. They were stripped of endings and passive forms, etc., and were the base component of the language. The deep structure of a sentence could contain ideas that were implicit (but unstated) in a final form of the sentence. For example, 'The man was shot' could have been generated by the deep structure 'Someone shoots the man'.

These deep structure versions of sentences, while representing the essential thought behind a sentence, were not necessarily grammatical. They also did not represent the full range of natural language. Chomsky augmented this base component with groups of transformations.

There are two basic types of transformations: obligatory transforms are ones that must be applied to every deep structure before it can be considered a grammatical sentence. For instance, an obligatory transform guarantees the agreement of subject and verb. Kernel sentences are sentences that have had only obligatory transformations applied to them. Optional transforms are also available, and can do such things as change a sentence into the passive voice. Sentences which have had both obligatory and optional transforms applied to them are called non-kernel sentences. Both kernel sentences and non-kernel sentences are grammatical, and are examples of surface structure as opposed to deep structure. Graphically, we have:



For example, phrase structure rules could produce the deep structure:



Applying the obligatory transformation for agreement of a verb and a third person singular subject, and also the transformation for past tense, would produce the kernel sentence "Bill shot John".

Applying the optional passive transform to this kernel sentence produces "John was shot by Bill".

To get a better feel for the idea of a grammatical transformation, it might be useful to look at a few of the simpler transformations. The following have been adapted from a 'new grammar' textbook (LaPalombara, 1976). The symbol 's' means to add an 's' (at least conceptually - 'man' + 's' becomes 'men', not 'mans'), and '0' means do not add an 's'. In the case of a noun, the 's' indicates plural and the '0' singular. Examples:

The	man + 0	Tense + buy s	the	house + 0.
The	man	buys	the	house.

The	man + s	Tense + buy 0	the	house + 0.
The	men	buy	the	house.

The present tense transformation rule (obligatory) used in these sentences is:

Present ====> [s, if the NP is third person sg.
0, otherwise

Notice that these are much like the context-sensitive rules that could be used to extend a context-free grammar. They require looking back in the sentence, for instance, to see if the subject is third person singular.

The following (optional) transformation rule changes a declarative sentence into a "Yes - No" question.

$NP+Tense+aux_1$ $(+Aux_2)$ $(+Aux_3)+V+$<rest of sentence>

==> $Tense+Aux_1+NP$ $(+Aux_2)$ $(+Aux_3)+V+$<rest of sent.>

"The man was going to buy a ticket"

==> "Was the man going to buy a ticket?"

Notice that this rule requires that there be at least one auxiliary verb if the rule is to apply. Another rule is used when no auxiliaries are present which introduces the word 'do' in place of 'Aux', as in "The man bought a ticket" ==> "Did the man buy a ticket?"

A transformation that is used frequently is the "Active - Passive" transform.

$NP_1 + T + V_{transitive} + NP_2$

==> $NP_2 + T + be + V_{transitive} + 'by' + NP_1$

"The man bought a ticket"

==> "A ticket was bought by the man".

Transformational grammars extended the basic context-free phrase structure rules by adding some context-sensitive rules. The added power of the formalism allowed a much neater statement of some complex grammatical constructions than had been available previously. Computational linguists found the ideas particularly appealing. As mentioned before, computer analysis of natural language often requires some semantics, and the idea of deep structure was a step in that direction. The emphasis on syntax also made it less expensive to implement on a computer than the more complex semantic approaches.

The biggest drawback to using transformational grammar to parse sentences on a computer is that transformational grammar is essentially a generative grammar. Starting from scratch, it provides a very powerful way to create new grammatical sentences. It is not, however, a recognizer of syntactic forms. The rules cannot be run in reverse to determine if an input sentence is grammatical, or to determine its constituent parts.

Transformational grammar is not without its critics. Some linguists (Sampson, 1980) feel that transformational grammar indeed may be a more compact way to represent certain complex grammatical constructions, but it is not the way that humans generate sentences in their minds. In other words, it is an inaccurate model of the human sentence generating mechanism. There are still many researchers studying transformational grammar and searching for linguistic universals, but the

voices of dissent are getting louder.

Computational linguists have never been able to use the full power of transformational grammar, due to the difficulties of using it as a recognizer. Even with that limitation, however, transformational grammar has proven itself a very useful tool for natural language analysis, as we will see when we get to William Woods' work on augmented transition networks.

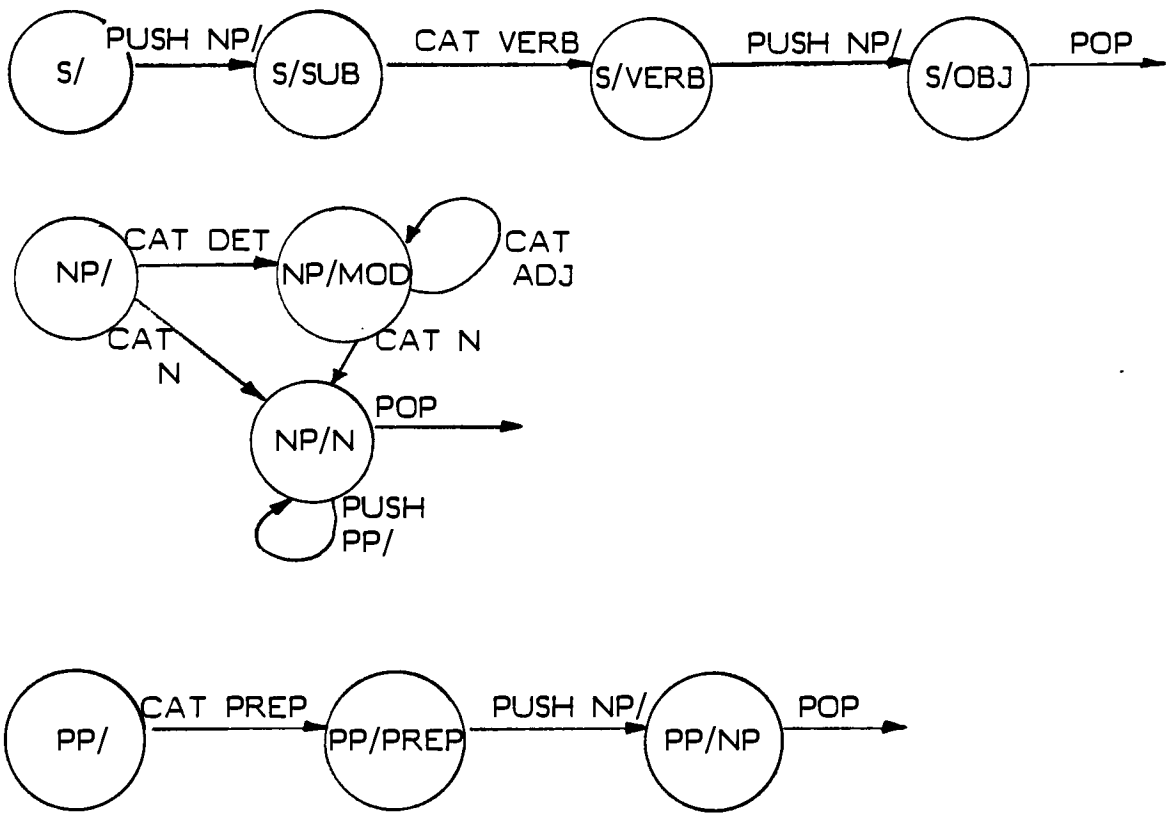
CHAPTER 6

Recursive transition networks

As was mentioned earlier, finite state transition networks were used in some early attempts to represent natural language on computers. They did not prove to be particularly successful. Extending this formalism by adding recursion, however, made a much better model of natural language grammar. The resulting network structure is called a recursive transition network, sometimes abbreviated to RTN. It is weakly equivalent to a context-free grammar in that it can produce the same set of languages, but the sentences in the languages may have different parse trees.

Recursive transition networks are capable of suspending the processing of one type of constituent in order to process another constituent in the network. It PUSHes from the current location in the network to a sub-network. The processing continues there until this inner constituent is processed. The result is then POPped back to the higher-level network, where the original processing continues. Consider a sentence such as: "The man who was soon to be elected president was flying to Washington". The main part of the sentence is: "The man was flying to Washington". That would be handled on the main sentence network. When the modifying clause "who was soon to be elected president" is parsed, transfer will shift from the main network to a subordinate one to process this part of the sentence.

Let's now look more closely at a sample recursive transition network, and see how it might be used to parse a sentence.



In this RTN, the nodes are labelled using the pattern (subgraph / part just parsed). 'CAT' means 'syntactic category', so 'CAT N' on an arc would require that the current word has to be a noun if that arc is to be traversed. 'PUSH NP/' calls for a suspension of current parsing, and relocating to the subgraph 'NP/' to try to build sentence constituent at that point. 'POP' signals that a sentence constituent has been built successfully during the 'PUSH', so control returns to the end of the PUSHed-from arc.

This RTN can be used to accept simple subject-verb-object declarative sentences such as:

"Birds in the wild eat fruit."

"The children watched an old lion at the zoo."

Let's parse the second of these two sentences using this RTN.

Start at (S/). Immediately 'PUSH' down a level to subgraph (NP/). The current word is 'the', a determiner, so the arc to (NP/MOD) is traversed. The word pointer is advanced to 'children'. The 'CAT ADJ' test is unsuccessful, but 'CAT N' can be followed to (NP/N). 'PUSH PP/' is attempted, and the arc test ('CAT PREP') out of node (PP/) fails, so the attempt to build a prepositional phrase is halted. Control returns to (NP/N). From (NP/N), a 'POP' can and does occur, passing control back to the end of the 'PUSH-NP' arc. At this point, the noun phrase

(NP (DET 'the') (N 'children'))

has been accepted as subject of the sentence.

The next word is 'watched', a verb. Since (S/SUB) can advance to (S/VERB) if a verb is present, it does so, building (V 'watched') as the verb of the sentence.

At (S/VERB), a 'PUSH' to (NP/) is again performed, this time to try to find the object of the sentence. The determiner 'an' permits traversal to node (NP/MOD). The 'CAT ADJ' arc is followed back to (NP/MOD) again, accepting the word 'old'. The next word, 'lion', permits an advance to node (NP/N). 'PUSH PP/' is attempted, and since the next word ,'at', is a preposition, state (PP/PREP) is reached. Notice that the nesting is now <u>two</u> levels deep in the network.

The 'PUSN NP/' found at (PP/PREP) causes a push to a third level. The noun phrase 'the zoo' is accepted in sub-network (NP/). At (NP/N), control 'POP's back to node (PP/NP). The prepositional phrase is now complete:

(PP(PREP 'at')(NP(DET 'the')(N 'zoo')))

At (PP/NP), another 'POP' returns control to (NP/N). The object (a noun phrase with modifying prepositional phrase) is complete:

(NP(DET 'an')(ADJ 'old')(N 'lion')
(PP(PREP 'at')(NP(DET 'the')(N 'zoo'))))

A 'POP' returns control to the top level graph, (S/), at node (S/OBJ). The 'object' of the sentence is complete. At this point, one last 'POP' occurs, signifying the successful completion of the sentence, below:

(NP(DET 'the')(N 'children')) (V 'watched')
(NP(DET 'an')(ADJ 'old')(N 'lion)
(PP(PREP 'at')(NP(DET 'the')(N 'zoo'))))

The recursive PUSHes in recursive transition networks capture some of the generalities of natural language that a successful, perspicuous grammar should. The temporary suspension of processing at one point in the network to allow processing of embedded constituents, increases the capabilities of the network significantly, while avoiding the complexity and expense of new subgraphs.

Recursive transition networks have the same power as context-free grammars. This means that they are unable to handle all the grammatical

constructions in natural language in a satisfactory way. However, RTNs are able to handle large pieces of natural language, and are, therefore, of some real value. They also are the foundation upon which a very successful syntactic parser has been constructed, namely the augmented transition networks to be discussed in a later chapter.

CHAPTER 7

Earley's algorithm

Before proceding to augmented transition networks, an important parsing algorithm that was developed in 1968 by Jay Earley at Carnegie-Mellon University will be examined (Earley, 1968, 1970).

Earley's algorithm is an algorithm for parsing context-free grammars. It is mentioned at this point because the natural language grammars discussed so far have been context-free grammars or their equivalent. The syntactic analyzer's grammar table was equivalent to a context-free grammar; the phrase structure core grammar that Chomsky uses in transformational grammar is a context-free grammar; and the recursive transition network just discussed is weakly equivalent to a context-free grammar. It has also been shown that Earley's algorithm can be modified fairly easily to parse recursive transition networks (Woods, 1969).

Earley's algorithm (in a somewhat modified form) is being used for a current natural language research project at M.I.T. (Martin, Church, and Patil, 1981). These researchers are creating a syntactic parser before going on to examine some semantic issues. They have concluded that, rather than setting up an unnecessarily complex formalism to handle all possible sentences, there should be different parsers for different types of sentences. They have broken up the set of possible sentences into three cases. The largest group of sentences are those that are amenable to

representation in a context-free grammar. A modified form of Earley's algorithm is used to parse these sentences. The second group of sentences are those that contain conjunctions, and those with movement of constituents within a sentence, so called 'wh-movement'. Special purpose algorithms are used to handle these situations. The third group consists of a number of minor special cases, such as idioms that are handled by special-case procedures as well. This notion of splitting the parsing into cases may be a very practical way of handling some of the syntactic complexities of natural language. It also shows that Earley's algorithm still has practical applications in natural language analysis.

Earley's algorithm is an important method of parsing a context-free grammar for a number of reasons. It is a top-down, breadth-first parser. It begins with the 'start' symbol, and follows through all the possible parses, one step at a time, using the input string to guide the process. The time bound on Earley's algorithm is proportional to n^3, where 'n' is the length of the input string. This compares quite favorably with other context-free parsing algorithms. The Cocke-Younger-Kasami algorithm, for example, parses in time proportional to n^3 for <u>any</u> context-free grammar it is given. Earley's algorithm is a <u>worst case</u> $O(n^3)$. It is $O(n^2)$ for unambiguous grammars, and $O(n)$ for many context-free grammars, including most programming language grammars. It has been modified for many special purposes, but the basic idea behind it is an important one for efficient parsing of context-free grammars and recursive transition networks.

Earley's algorithm is a tabular parsing method. As each element of an input string is accepted, a table of the possible applicable grammar rules is maintained. A pointer (which is called 'dot') is positioned in each rule to show how far along the parsing may have progressed in each rule.[1] Items are also followed by a number stating the rule set in which the particular item originated. It is necessary to record this information because a new item may be added to the current set of rules as parsing progresses, and we need to be able to get back to an item's starting point if it appears that the underlying rule was indeed used in the parse.

I_0 is the initial set of items derived from the production rules. It contains all the rules that can be used to accept any possible initial symbol, either by accepting it directly or through other rules. Start by including an item [<phi> --> .S , 0] , where 'S' is the start symbol. If S --> A <alpha> is a production, then add item [S --> .A <alpha> , 0] to I_0. By transitive closure, if A --> B <beta> is a production, then add item [A --> .B <beta> , 0] to I_0 also. Continue in this fashion until no more new items can be added to I_0.

I_0 now contains all the items capable of accepting the first symbol, a_1, in the input string. 'Dot' is placed before the possible accepting symbol (in this case, it will be located before the first symbol on the right side of each production).

---

[1] The word rule will be used to describe a production rule in the grammar, and item will be used to describe a production rule that has been "dotted".

As each input symbol is read, a new table of items is created. If the current input symbol is the first symbol after 'dot' in any item in the most recent table, then 'dot' is moved over that symbol and the item is placed in the new table of rules. Earley calls this process the scanner.

The predictor examines all these newly created items and uses transitive closure on the symbol immediately after 'dot' in any of these items that the 'scanner' has created.

The completer looks for any items that the 'scanner' has completed - i.e., items in which 'dot' follows the last symbol. This item is then traced back to the item table in which it originated, and all the items in that table are examined to see if 'dot' can be advanced by this completed rule. If any of these items can be advanced, they are brought forward to the newly constructed table of items.

This process continues until all the input symbols have been consumed. The last table of items (i.e., I_n for an input string of length 'n') will contain the item [<phi> --> S . , 0] if the input string is accepted.

A more formal description follows. The notation used is a combination of that used by Earley (Earley, 1968, 1970) and by Aho and Ullman (Aho and Ullman, 1972).

Start with a context-free grammar, G, where

G = (N , <Sigma> , P , S) , and

N = the set of non-terminal symbols,

<Sigma> = the set of terminal symbols,

P = the set of production rules, and

S = the start symbol.

Also given is an input string = $a_1\ a_2 \ldots a_m$.

Construct I_0:

Step 1.) Add the item [<phi> --> .S , 0].

Step 2.) For every rule of the form S --> <alpha> in P, add an item to I_0 of the form:

[S --> .<alpha> , 0]

Step 3.) Transitive closure:

If item[A -->.B <gamma> , 0] is in I_0,

and B --> C <delta> is in P ,

then add [B --> .C <delta> , 0] to I_0.

For example, given the productions:

S --> A <alpha>

A --> B <beta>

B --> c

To create I_0 for this grammar, add the items:

<phi> --> .S , 0 by rule 1

S --> .A <alpha> , 0 by rule 2

A --> .B <beta> , 0 by rule 3

B --> .c , 0 by rule 3

Construct I_n:

Subscript 'n' can be any number from 1 to m, where 'm' is the length of the input string. In order for I_n to be constructed, I_1, I_2, ... , I_{n-1}, must have been constructed already.

Step 4.) The scanner.

If [A --> <alpha> . a_n <beta> , i] is in I_{n-1} then add the item

[A --> <alpha> a_n . <beta> , i] to I_n.

Step 5.) The predictor.

If [A --> <alpha> . A <beta> , i] is in I_n and A --> <gamma> is in P, then add the new item

[A --> . <gamma> , n] to I_n.

Step 6.) The completer.

If [A --> <alpha> . , i] is in I_n (i.e., a completed rule has been found in I_n), then examine I_i for items of the form

[B --> <beta> . A <gamma> , j] .

If any are located, then add to I_n item(s) of the form

[B --> <beta> A . <gamma> , j] .

Repeat steps 5 and 6 until no new items can be added.

When I_m has been completed (where 'm' is the length of the input string), examine it for items of the form [<phi> --> S . , 0]. If found,

then the input string is in L(G).

To see how Earley's algorithm works in practice, a context-free grammar for simple declarative sentences is provided below. This grammar is almost equivalent to the RTN for declarative sentences given earlier. The only difference is that this grammar does not accept more than one adjective as a noun modifier, whereas the RTN did.

The grammar:

```
S      -->  NP  VP
VP     -->  V  NP  |  V  NP  PP
PP     -->  P  NP
NP     -->  DET  N  |  DET  ADJ  N  |  N
V      -->  AUX  V
N      -->  N  PP
DET    -->  the  |  a
V      -->  bought
N      -->  man  |  store  |  lamp
ADJ    -->  new
P      -->  in
```

I_0: Initialization

```
<phi>   -->  .  S  ,  0                  step 1
S       -->  .  NP  VP  ,  0             step 2, using item 1
NP      -->  .  DET  N  ,  0             step 3, using item 2
NP      -->  .  DET  ADJ  N  ,  0        step 3, using item 2
NP      -->  .  N  ,  0                  step 3, using item 2
N       -->  .  N  PP  ,  0              step 3, using item 5
N       -->  .  man  ,  0                step 3, using item 5
N       -->  .  store  ,  0              step 3, using item 5
N       -->  .  lamp  ,  0               step 3, using item 5
DET     -->  .  the  ,  0                step 3, using item 3
DET     -->  .  a  ,  0                  step 3, using item 3
```

I_1: a_1 = 'the'

DET	-->	the . , 0	step 4 on item 10 of I_0
NP	-->	DET . N , 0	step 6 using item 1
NP	-->	DET . ADJ N , 0	step 6 using item 1
N	-->	. N PP , 1	step 5 from item 2
N	-->	. man , 1	step 5 from item 2
N	-->	. store , 1	step 5 from item 2
N	-->	. lamp , 1	step 5 from item 2

I_2: a_2 = 'man'

N	-->	man . , 1	step 4 on item 5 in I_1
NP	-->	DET N . , 0	step 6 using item 1
N	-->	N . PP , 1	step 6 using item 1
S	-->	NP . VP , 0	step 6 using item 2
PP	-->	. P NP , 2	step 5 from item 3
VP	-->	. V NP , 2	step 5 from item 4
VP	-->	. V NP PP , 2	step 5 from item 4
V	-->	. AUX V , 2	step 5 from item 6
V	-->	. bought , 2	step 5 from item 6
P	-->	. in , 2	step 5 from item 5

I_3: a_3 = 'in'

P	-->	in . , 2	step 4 on item 10 in I_2
PP	-->	P . NP , 2	step 6 using item 1
NP	-->	. DET N , 3	step 5 from item 2
NP	-->	. DET ADJ N , 3	step 5 from item 2
NP	-->	. N , 3	step 5 from item 2
DET	-->	. a , 3	step 5 from item 3
DET	-->	. the , 3	step 5 from item 3
N	-->	. N PP , 3	step 5 from item 5
N	-->	. man , 3	step 5 from item 5
N	-->	. store , 3	step 5 from item 5
N	-->	. lamp , 3	step 5 from item 5

I_4: a_4 = 'the'

DET	-->	the . , 3	step 4 on item 7 in I_3
NP	-->	DET . N , 3	step 6 using item 1
NP	-->	DET . ADJ N 3	step 6 using item 1
N	-->	. N PP , 4	step 5 from item 2
N	-->	. man , 4	step 5 from item 2
N	-->	. store , 4	step 5 from item 2
N	-->	. lamp , 4	step 5 from item 2
ADJ	-->	. new , 4	step 5 from item 3

I_5: a_5 = 'store'

N	-->	store . , 4	step 4 on item 6 in I_4
NP	-->	DET N . , 3	step 6 using item 1
N	-->	N . PP , 4	step 6 using item 1
PP	-->	. P NP , 5	step 5 from item 3
PP	-->	P NP . , 2	step 6 using item 2 (This accepts 'in the store'.)
N	-->	N PP . , 1	step 6 using item 5 (This accepts 'man in the store'.)
NP	-->	DET N . , 0	step 6 using item 6 (This accepts 'the man in the store' and allows us to keep building from I_0.)
N	-->	N . PP , 1	step 6 using item 6
S	-->	NP . VP , 0	step 6 using item 7
PP	-->	. P NP , 5	step 5 from item 8
VP	-->	. V NP , 5	step 5 from item 9
VP	-->	. V NP PP , 5	step 5 from item 9
P	-->	. in , 5	step 5 from item 4
V	-->	. bought , 5	step 5 from item 11

I_6: a_6 = 'bought'

V	-->	bought . , 5		step 4 on item 14 in I_5
VP	-->	V . NP , 5		step 6 using item 1
VP	-->	V . NP PP , 5		step 6 using item 1
NP	-->	. DET N , 6		step 5 from item 2
NP	-->	. DET ADJ N , 6		step 5 from item 2
NP	-->	. N , 6		step 5 from item 2
N	-->	. N PP , 6		step 5 from item 6
DET	-->	. the , 6		step 5 from item 4
DET	-->	. a , 6		step 5 from item 4
N	-->	. man , 6		step 5 from item 6
N	-->	. store , 6		step 5 from item 6
N	-->	. lamp , 6		step 5 from item 6

I_7: a_7 = 'a'

DET	-->	a . , 6		step 4 on item 9 in I_6
NP	-->	DET . N , 6		step 6 using item 1
NP	-->	DET . ADJ N , 6		step 6 using item 1
N	-->	. N PP , 7		step 5 from item 2
N	-->	. man , 7		step 5 from item 2
N	-->	. store , 7		step 5 from item 2
N	-->	. lamp , 7		step 5 from item 2
ADJ	-->	. new , 7		step 5 from item 3

I_8: a_8 = 'new'

ADJ	-->	new . , 7		step 4 on item 8 in I_7
NP	-->	DET ADJ . N , 6		step 6 using item 1
N	-->	. N PP , 8		step 5 from item 2
N	-->	. man , 8		step 5 from item 2
N	-->	. store , 8		step 5 from item 2
N	-->	. lamp , 8		step 5 from item 2

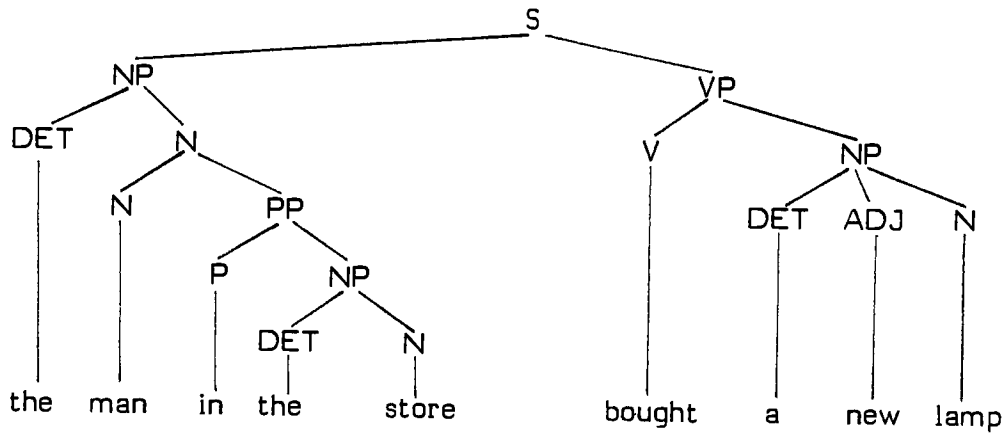
I_9: a_9 = 'lamp'

N	-->	lamp . , 8	step 4 on item 6 in I_8
NP	-->	DET ADJ N . , 6	step 6 using item 1 (This accepts 'a new lamp'.)
N	-->	N . PP , 8	step 6 using item 1
VP	-->	V NP . , 5	step 6 using item 2 (This accepts 'bought a new lamp'.)
VP	-->	V NP . PP , 5	step 6 using item 2
S	-->	NP VP . , 0	step 6 using item 4 (This accepts the entire sentence.)
<phi>	-->	S . , 0	step 6 using item 6 (This formally signifies the acceptance of the sentence.)
PP	-->	. P NP , 9	step 5 from item 3
P	-->	. in , 9	step 5 from item 8

Take the accepting items in each set, and write them in the opposite order in which they were generated.

<phi>	-->	S	from I_9
S	-->	NP VP	from I_9
VP	-->	V NP	from I_9
NP	-->	DET ADJ N	from I_9
N	-->	lamp	from I_9
ADJ	-->	new	from I_8
DET	-->	a	from I_7
V	-->	bought	from I_6
NP	-->	DET N	from I_5
N	-->	N PP	from I_5
PP	-->	P NP	from I_5
NP	-->	DET N	from I_5
N	-->	store	from I_5
DET	-->	the	from I_4
P	-->	in	from I_3
NP	-->	DET N	from I_2
N	-->	man	from I_2
DET	-->	the	from I_1

These productions yield this parse tree:



CHAPTER 8

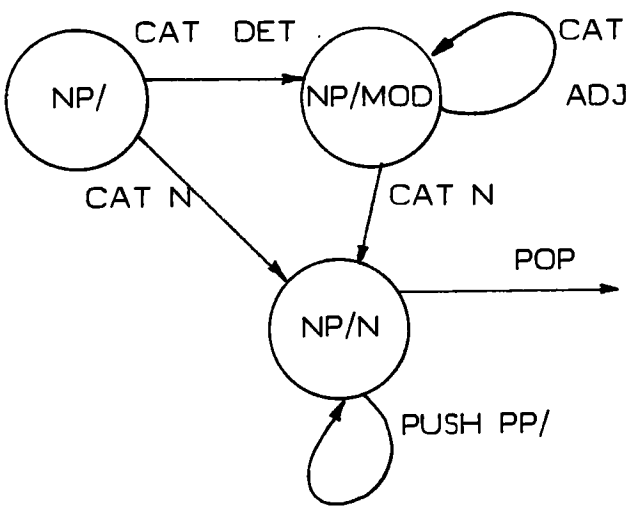
Augmented transition networks

As mentioned before, adding recursion to a finite state network creates a formalism that is equivalent in power to a context-free grammar. However, it was also pointed out that context-free grammars are not adequate to represent the full richness of natural language. It is necessary to add some context-sensitive rules (or their equivalent) if a reasonably compact and understandable grammar is to be achieved. Recursive transition networks were extended to augmented transition networks (or ATNs) to do just that (Thorne et al, 1968; Bobrow and Fraser, 1969; Woods, 1969). Registers were established that allowed the saving of information for future reference (e.g., save the 'number' of the subject so that later the agreement of subject and verb can be guaranteed). Arbitrary tests and conditions were allowed on the arcs (not just the 'category' tests allowed in RTNs). Structure building actions were also added to the arcs, so that traversing an arc would cause a phrase or clause to be created and saved in a register.

With the addition of registers, complex tests on arcs, and structure-building actions on arcs, this formalism now has the power of a Turing machine (Woods, 1970, p. 597).

Augmented transition networks have emerged as a powerful tool with which to syntactically parse natural language. Since their introduction, over a decade ago, ATNs have become the standard against which other syntactic parsers are measured.

To begin to get some idea of how an ATN operates, consider the (NP/) subgraph from the earlier recursive transition network. It is coded below as an augmented transition network in LISP. The (NP/) subgraph is:



In LISP, we have:

```
(NP/    (CAT DET T                  ; enter (NP/) - is the word
                                    ; a DET?
           (SETR DET *)             ; yes it is - set the DET
                                    ; register.
           (TO NP/MOD))             ; go to node NP/MOD, but
                                    ; first advance one word.
        (CAT N T                    ; not a DET - is it a
                                    ; noun?
           (JUMP NP/N))             ; noun found - go to NP/N,
                                    ; but do not advance a word.
        (T 'fail'))                 ; no DET, no Noun - announce
                                    ; a failure.
(NP/MOD (CAT ADJ T                  ; enter (NP/MOD) - is the
                                    ; word an ADJ?
           (SETR ADJS               ; yes - add it into the
                                    ; ADJS register.
              (APPEND ADJS *))
                                    ;
           (TO NP/MOD))             ; proceed to (NP/MOD) -
                                    ; advance to next word.
        (CAT N T                    ; is the word a noun?
                                    ;
           (JUMP NP/N))             ; yes - go to (NP/N),
                                    ; no word advancement.
        (T 'fail'))                 ; did not find any
                                    ; acceptable words -'fail'.
(NP/N   (SETR NU                    ; found a noun phrase.
                                    ; set the 'number'.
           (GETF NUMBER))           ; retrieve 'number' from
                                    ; word's feature list.
        (POP (BUILDQ                ; POP the completed noun phrase
                                    ; up one level.
              (NP + + (N *)(NU +) DET ADJ NU ))))
                                    ;
```

The following LISP functions are used:

*	= current word
SETR	= set a register
TO	= goto this node, advance one word
JUMP	= goto this node, don't advance the word pointer
GETF	= get this feature
GETR	= get the contents of the register
BUILDQ	= build a structure, in this case a noun phrase. The '+'s are filled in with the values of DET, ADJ, and NU, respectively.
POP	= return the structure that follows to the next highest level in the processing, i.e. back to where the PUSH was initiated. The newly created structure will return to that higher level in '*'.

Notice the use of registers. In this ATN, the 'number' of the noun is saved in register 'NU'. 'DET' and 'ADJS' are also registers capable of accepting words. If this noun phrase is the subject of a sentence, the value of 'NU' will be checked with the number of the verb to see that they match.

The only tests on arcs that are listed in this example are the 'category' tests. More complex tests are available. For example, reaching node (S/SUB) and beginning to accept a verb, the following tests might be found:

```
(S/SUB  (CAT V (AGREE (GETR SUBJ) (GETF *))
               (SETR NU (GETF NUMBER))
               (SETR TNS (GETF TENSE))
               (SETR V (BUILDQ (V (TNS +) (NU +) (V *) TNS NU )))
               (TO S/VERB)))
```

Notice that two tests need to be satisfied before this arc can be traversed: the word must be a verb, and it must AGREE with the subject. Although this is a simple example, it shows all the basic types of augmentations that are used in an ATN.

The first ATN is usually considered to be the one developed at Edinburgh (Thorne et al , 1968, 1969). Almost simultaneously, Bobrow and Fraser developed an ATN in the U.S. (Bobrow and Fraser, 1969). The early ATN that stands out, however, was the one developed by William Woods at Harvard and at Bolt, Beranek, and Newman (Woods, 1969, 1972). The Woods ATN was used in a program called LUNAR (Woods, 1972). It parsed queries to a data base of information on the lunar rock samples brought back by the astronauts. The idea was to allow geologists to query the data base using English rather than a formal computer language. LUNAR contained both a syntactic and semantic component and was very successful. The syntactic parsing used an ATN.

The LUNAR ATN stands out from from the others for a number of reasons. It was one of the first ATNs, and it was the first one in which the underlying theory was examined. Woods did a thorough job, using a

large enough subset of English to permit the parsing of a wide range of sentence types. A semantic analysis of each parsed sentence was also carried out. Following are some of the actual sentences that geologists asked of LUNAR, and that it was able to answer (Woods, 1972).

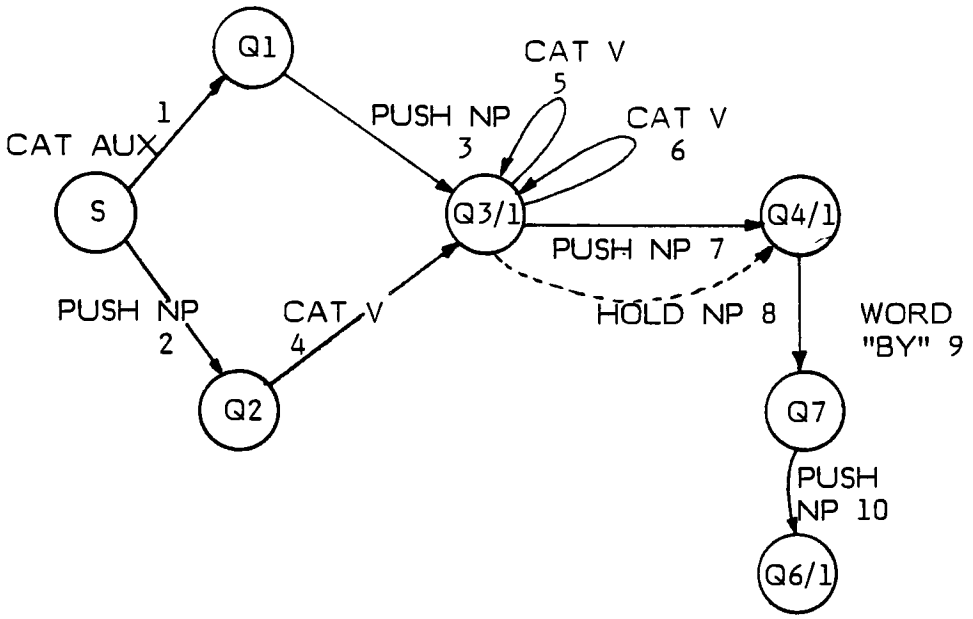
'What is the average plagioclase content in crystalline rocks?'

'List modal plag analyses for lunar samples.'

'In how many breccias is the average concentration of aluminum greater than 13 percent?'

LUNAR also performed some transformations on the sentences it parsed. The transformation rules were incorporated directly into the structure building rules; no separate transformational component was used. The parser produced a deep structure representation.

Following is an ATN fragment that appeared in one of Woods' papers (Woods, 1969). The network is abbreviated somewhat in order to emphasize one feature of Woods' system, that being the parser's ability to derive the deep structure of a sentence without recourse to a separate transformational component. The network follows:



ATN network adapted from (Woods, 1969).

Conditions	Actions
1.) T	1.) (SETR V *) (SETR TNS (GETF TENSE)) (SETR TYPE (QUOTE Q))
2.) T	2.) (SETR SUBJ *) (SETR TYPE (QUOTE DCL))
3.) T	3.) (SETR SUBJ *)
4.) T	4.) (SETR V *) (SETR TNS (GETF TENSE))
5.) (AND (GETF PPRT) (EQ (GETR V) (QUOTE BE)))	5.) (HOLD (GETR SUBJ)) (SETR SUBJ (BUILDQ (NP (PRO 'Someone')))) (SETR AGFLAG T) (SETR V *)
6.) (AND (GETF PPRT) (EQ (GETR V) (QUOTE HAVE)))	6.) (SETR TNS (APPEND (GETR TNS) (QUOTE PERFECT))) (SETR V *)
7.) (TRANS (GETR V))	7.) (SETR OBJ *)
8.) (TRANS (GETR V))	8.) (SETR OBJ *)
9.) (GETR AGFLAG)	9.) (SETR AGFLAG NIL)
10.) T	10.) (SETR SUBJ *)

Conditions and forms of final states:

(Q3):
Condition: (INTRANS (GETR V))
Form:
(BUILDQ (S + + (TNS +)(VP (V +))) TYPE SUBJ TNS V)

(Q4) and (Q6):
Condition: T
Form:
(BUILDQ (S + + (TNS +)(VP (V +) +)) TYPE SUBJ TNS V OBJ)

A partial ATN adapted from (Woods, 1969)

Consider the sentence:

"The river was crossed by the troops."

(S) Begin at (S). Arc 1 can not be traversed, so PUSH for a 'NP'. This will be successful, returning (NP (DET 'the) (N 'river')). Before going to node (Q2), it is necessary to do the actions associated with arc 2, namely the two SETR operations. Set the SUBJ register to (NP (DET 'the') (N 'river')) and TYPE = DCL. (The quote before DCL inhibits evaluation of the expression, i.e. treat DCL as a constant, not a variable that should be evaluated).

(Q2) The only way to leave (Q2) is if the next word is a verb, which 'was' is. En route to node (Q3/1), set the verb register to the untensed form of the word.

(Q3/1) The '/1' at the end of the node name indicates that it is an accepting state. If the sentence ended here, it would be grammatical ('The river was.'). There are more words, however, so arc 5 is attempted. The current word, 'crossed', is a verb. The conditions on arc 5 require that it be a past participle (which it is), and the current contents of the verb register must be 'be'. Both these tests are satisfied, so the actions are undertaken. The current contents of the subject register are put in the HOLD register. The HOLD register is used when a sentence component

is found out of sequence. The component is put aside in the HOLD register until it is needed later. Since this a passive sentence, there is a good chance that the first noun phrase will be used as the object of the sentence. It will sit in the HOLD register until then.

The next action builds a new subject register, SUBJ = (NP (PRO 'Someone')). The AGFLAG (agent flag) is set, indicating that we want to be on the lookout for an agent (or doer of the action) later in the sentence.

The final action on this arc updates the verb register, V = 'cross'.

(Q3/1) Traverse arc 5 to node (Q3/1). The current word is 'by'. Arcs 5 and 6 will fail, and the PUSH for a NP on arc 7 will fail also. Arc 8 is a dotted line which is a special arc that Woods called a virtual arc. It can be traversed only if the HOLD register contains a noun phrase. The other test on arc 8, that the verb be transitive ('cross' requires an object), also is satisfied, so the virtual arc is traversed to node (Q4/1). Because it is a virtual arc, '*' contains the contents of the HOLD register. The word pointer in the sentence stays at 'by'. The action on arc 8 sets the object register, OBJ = (NP (DET 'the') (N 'river')).

(Q4/1) Notice that (Q4/1) is an accepting state, and all the necessary sentence constituents have been filled. If there were no more words in the sentence (i.e., "The river was crossed."), the BUILDQ rule at Q4 could be used to construct the sentence:

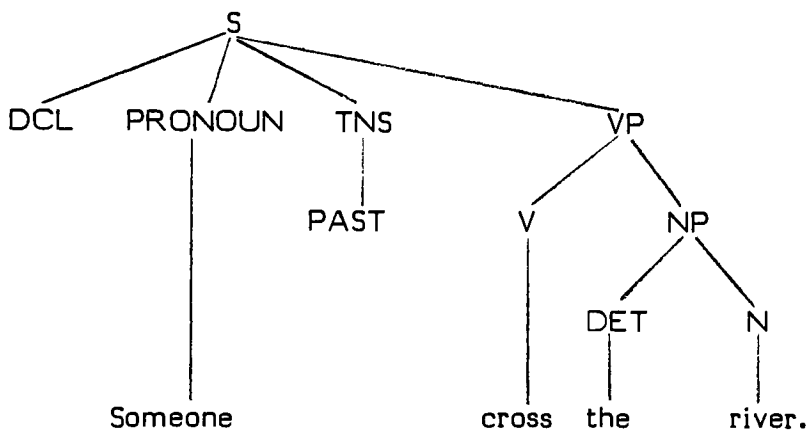
BUILDQ (S + + (TNS +) (VP (V +) +)) TYPE SUBJ TNS V OBJ)

This would yield the deep structure:

(S DCL (NP (PRO 'Someone')) (TNS PAST)

(VP (V 'cross') (NP (DET 'the') (N 'river'))))

The tree for the sentence would be:



However, there is still more sentence to parse. The word 'by' is pointed to, allowing the transition on arc 9. This says that the agent of the action has been found, so the agent flag is cleared (AGFLAG = NIL).

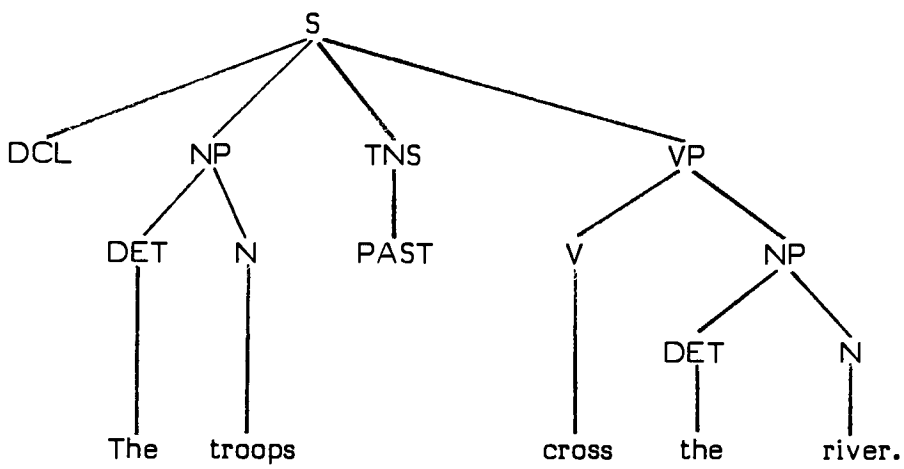
(Q7) The arc from (Q7) is a PUSH for a noun phrase. It is successful, returning (NP (DET 'the') (N 'troops)). The actions associated with arc 10 say to reset the subject register,
SUBJ = (NP (DET 'the') (N 'troops')).
Proceed to (Q6/1).

(Q6/1) This is an accepting state, and since there are no more words to accept, the parse is done. Use the Q6 BUILDQ rule (the same

one as Q4) to produce the structure:

(S DCL (NP (DET 'the') (N 'troops')) (TNS PAST)
(VP (V 'cross') (NP (DET 'the') (N 'river'))))

The associated tree structure is:



Notice that the question "Was the river crossed by the troops?" would produce the same deep structure, except that the TYPE would be Q (a question).

CHAPTER 9

Parsing strategies for ATNs

Most of the parsing strategies for ATNs are top-down. This means that the parsers start with the 'S' or 'Sentence' symbol and try to break that down into its constituent parts, finally arriving at the leaves of the tree (the actual words). This is sometimes called 'predictive' parsing, since the parser 'predicts', for example, that a sentence will break down into a 'noun phrase' followed by a 'verb phrase'. These 'predictions' are modified as parsing progresses. Top-down algorithms frequently employ a depth-first approach. As recognition of the sentence proceeds, if there is a choice between two or more arcs at a particular point, one of the arcs is taken and the rest are ignored, at least for the time being. The parser attempts to construct one successful parse as it moves through the network.

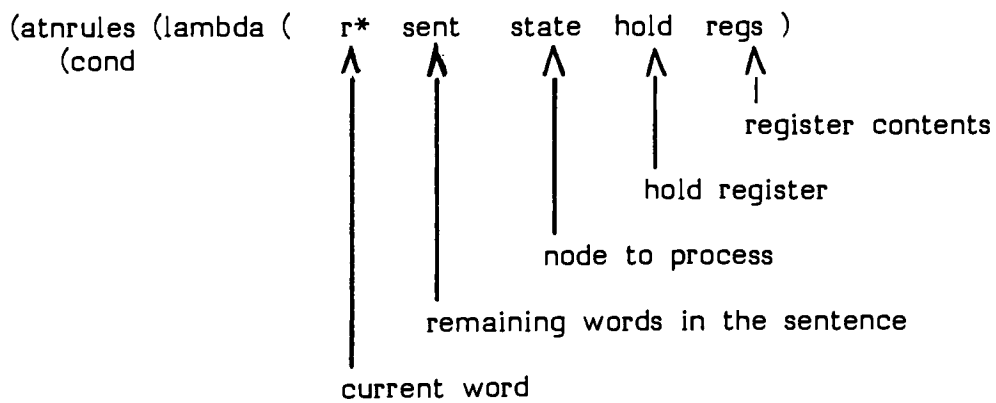
Compare this to Earley's algorithm, which is a top-down, breadth-first algorithm. All the alternatives are maintained as the words are accepted, which means that Earley's algorithm can provide all the possible parses when it terminates. A top-down depth-first algorithm will produce only one parse. If all the parses are desired, the parser will have to backtrack.

Backtracking is the biggest problem with top-down depth-first parsing. Although the alternatives at a branch point can be ordered so that

the most likely one is taken, inevitably the wrong alternative is taken at some point in some sentence (not all sentences are the 'most likely' ones).

A straight top-down depth-first algorithm without recursion requires that the programmer keep track of alternatives that have not yet been tried. It is necessary to maintain registers that store the state at the nodes, and record the paths taken or yet to be explored.

Recursive descent parsing may make this 'system accounting work' much simpler. In one particular recursive descent ATN parser (DiBacco and Wilcox, 1981), the entire grammar was coded as a single function in LISP. The function was of the form:



The function 'atnrules' had many 'cond', or conditional, statements. Each one of them corresponded to a node in the grammar network. For each call of 'atnrules', the first condition to be satisfied was the matching up of the 'state' parameter in the function call with the correct node (or 'cond') in the function. When the node had been found and any entry

tests had been satisfied, the registers and pointers were updated and another call to 'atnrules' was made with these new parameters.

This is a significant aid to backtracking because when an arc failed, the recursive function automatically tried the next possibility. The information needed at that level was present in the parameters to the function. If none of the possibilities worked at a given level, the recursion would step back to the previous level and try another alternative there. This method can be used to enumerate every possibility, which while making it slow, does provide a way to produce every parse of a sentence.

One of the difficulties with this type of single-step backtracking is that it tends to re-parse constituents over and over again that do not need to be re-parsed. Single-step backtracking, while easy to program, ignores completed intermediate constituents.

Consider these examples from (Marcus, 1980):

"Is the block sitting in the box ?"

"Is the block sitting in the box red ?"

In the first sentence, a top-down depth-first parse operating on the 'most likely alternative' method would conclude that 'the block' is subject, 'is sitting' is the verb, and 'in the box' is an adverb phrase, modifying the verb.

The second of these sentences would be parsed in the same way until the word 'red' is encountered. Realizing at this point that the parse is incorrect, the parser would begin to backtrack- one word at a time, trying all the possibilities. It will get all the way back to 'sitting' before

it can find the correct parse: 'the block' is subject, 'is' is the verb, 'sitting in the box' is a gerund phrase modifying the noun phrase 'the block' (this is also sometimes referred to as a 'relative clause'), and 'red' is a predicate adjective modifying 'the block'.

Backtracking algorithms, particularly those that blindly backtrack one step at a time, tend to have this type of inefficiency.

The LUNAR parser was similar to a recursive descent parser, but it contained some optimizations. It did keep a list of alternatives, and the 'state' of the system for each of these possibilities. These 'snapshots' of the system contained the registers, words still to be parsed, remaining arcs, and the stack. These alternatives were created whenever more than one path could be used to exit from a node, which might happen when a word could belong to more than one syntactic category. This is a modification of the list of alternatives that is created by Earley's algorithm.

The LUNAR parser was designed so that it could proceed either depth-first or breadth-first. It could take the single most likely alternative and follow that through to the end (depth-first), or it could advance each alternative one step at a time (breadth-first). This ability to parse sentences either depth-first or breadth-first was provided in the parser as an aid to experimentation.

One helpful optimization that Woods' LUNAR parser contained was a well-formed-substring-table, or WFST. Whenever a POP was encountered in the network, the newly formed constituent was saved in the WFST. This made the backtracking more efficient, allowing it to back over entire

phrases rather than just single words. In the first 'block' sentence given earlier, both 'the block' and 'in the box' would have been saved in the WFST. Recognizing these as completed constituents helps avoid the problem of unnecessary duplication of parses, and so speeds up the process.

CHAPTER 10

Systemic grammar

In the late sixties, scholars at London University developed an approach to linguistic structure that was an alternative to transformational grammar. Professors Halliday and Hudson believed that sentence organization was determined by the function of the sentence and the information that it was intended to convey. They developed a new grammar, systemic grammar, to describe this.

Halliday (Halliday, 1967, 1970) described three aspects of the function of language: the ideational, the interpersonal, and the textual. The ideational aspect is the one with which computational linguistics has been most concerned.

The ideational function of language communicates the ideas in a sentence. There are certain categories of information that may be present within a sentence: is there an action represented or a belief expressed, is there an actor or recipient of the action, etc.? These concerns are similar to those expressed by case grammar.

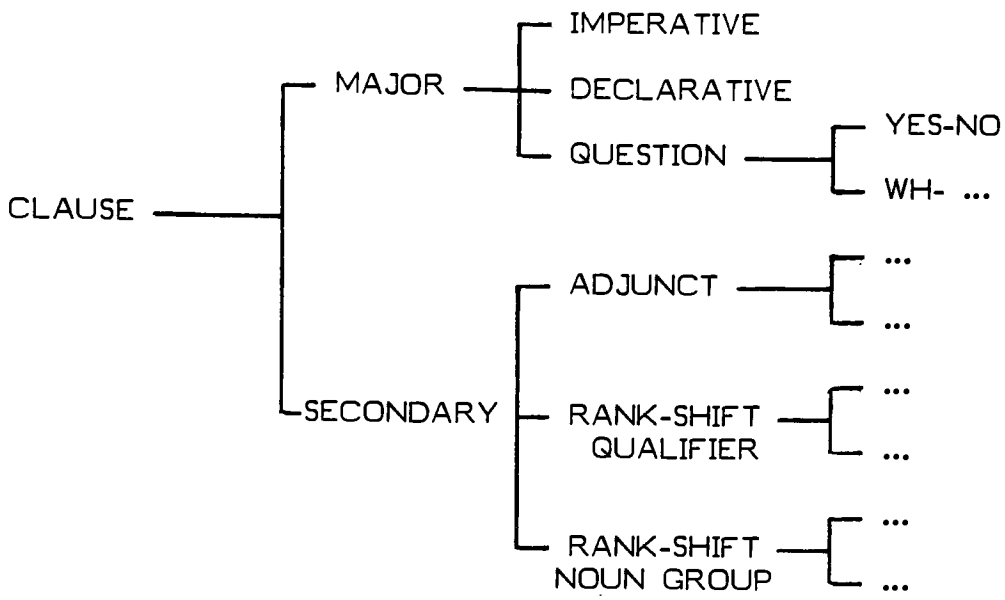
How does the form of a sentence convey meaning? Fillmore's development of case grammar (Fillmore, 1968) was motivated by certain languages (such as Latin and Greek) where the case endings (e.g., nominative, genitive, accusative, dative) convey semantic information. The dative case, for example, is used to convey the sense of 'to' or 'for', the reci-

pient of an action. English does not have case endings, but these same functions are achieved by word order and the use of certain prepositions. Fillmore argues that the underlying meaning of the sentence is carried by the main verb and accompanying noun phrases. The noun phrases will, depending on their 'case', serve certain specific purposes such as 'agent', 'instrument', or 'recipient'.

Systemic grammar tries to capture some of these same generalizations, and as such it integrates some semantic concerns into the syntactic analysis. For example, some verbs may have a second object that relates to the notion 'place', such as the verb 'send' (e.g., one object: 'Send the book', two objects: 'Send the book here'). Making this type of relationship explicit in a grammar provides more information about about how a sentence conveys meaning.

Feature lists are used by systemic grammar to record this information. Sentences are broken up into three ranks: clauses, groups (or phrases), and words. Each of these levels has lists of features attached. For example, at the 'clause' level, features state whether the clause is 'major' or 'secondary'; if 'major', whether it is 'imperative', 'declarative', or a 'question'. These sets of mutually exclusive options are the 'systems' for which the grammar is named.

To get a better idea of what sort of information is maintained by feature lists, consider this fragment of a clause network (Winograd, 1972).



Vertical lines delimit 'systems' from which you choose one of the available options. For example, a secondary (or dependent) clause must be one of the following:

(1) Adjunct - An adjunct clause modifies another clause, providing 'time' or 'causal' links. For example, 'While John slept, Sarah read a book.'

(2) Rank-shifted qualifier - This clause modifies a word, and hence has had its rank (clause) shifted down to the word rank. An example is 'She is the person to ask for advice.'

(3) Rank-shifted noun group - A clause can also be shifted to a noun group, as in 'Bicycling to work is fun.'

Each of these options subdivides further, providing even more information about the structure and meaning of the sentence. The presence of a feature deep in the feature network assumes the presence of all its preceding features. For example, feature 'WH-' means that the sentence

is a wh-question ('Who', 'What', 'Where', etc.). The feature QUESTION must be present, as must MAJOR and CLAUSE.

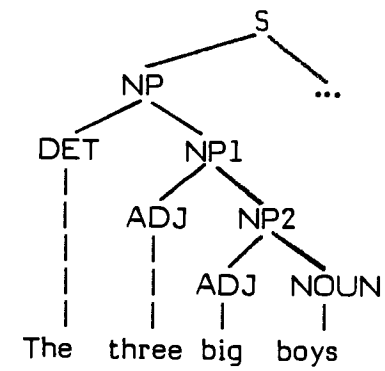
Systems may also interact. The structure of the clause is outlined in the previously listed system network. There is another clause network for transitivity, which is listed in the next chapter. The transitivity network states whether or not the verb takes an object, whether it may take a special kind of object, etc. As a clause is parsed, its feature list will contain items from both the structure and transitivity system networks. This combination of information from different networks and different ranks can provide a very complete description of the form of the sentence. The variety and richness of the information may also be of use to a semantic theory.

At most spots in a system network, there are only a few choices available as to what the next feature may be. This makes the notion of feature networks particularly easy to follow and use. The thorough use of system networks provides a lot of information about the sentence being examined.

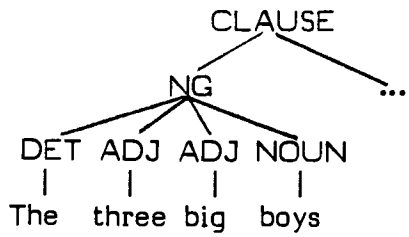
Systemic grammar is concerned with word groupings, and not details of word order. Since there are only three ranks, closely related words tend to be grouped together. This approach is less hierarchical (or at least it has fewer levels to it) than many phrase structure grammars. Compare the binary tree diagram for the phrase

'the three big boys'

with the systemic diagram next to it.



Binary tree representation



Systemic representation

Attached to the category clause in the systemic representation above will be a feature list of all the important features that apply (e.g. declarative, active, etc.). The noun group will have a feature list attached, indicating that it is a plural noun group, and in this case it will have a specific number associated with it. Each word will also have a feature list that was created from its dictionary entry. The following chapter shows an example of how these feature lists are constructed and maintained, and how they can be used to direct parsing.

Advocates of systemic grammar claim that 'features' are a very useful way to guide the parsing of a sentence. Having all the features explicitly available in a list is an aid to the building of syntactic structures, since the presence or absence of certain features can provide necessary information to the parser. Feature lists are also helpful for semantic analysis. For example, assume that the phrase 'by the old man' has just been parsed, and control has passed to a semantic analyzer. If the sentence has feature 'passive', then 'the old man' can be set up as

the logical subject of the sentence (e.g., 'She was driven home by the old man'). This would not have happened if the parser had found an 'active' feature (e.g., 'Fred drove home by the old highway'). Note that this particular example would have been handled in Woods' transformational grammar based ATN using an AGENT flag.

CHAPTER 11

SHRDLU

Terry Winograd was one of the first people to use systemic grammar in a computer based natural language system. His involvement in linguistics began as a graduate student in mathematics at M.I.T., during which time he went to the University of London to study under Halliday and Hudson.

Winograd first used systemic grammar in an analysis of tonal harmony (Winograd, 1968). He claimed the use of 'systems' - sets of mutually exclusive features that are followed like a decision tree - helped show clearly the underlying structure of the grammar, whether it was a grammar for English or for tonal harmony. He believed the power of systemic grammar was in its ability to describe a sentence or string of symbols as a number of these interacting systems.

Winograd's program SHRDLU was his most thorough exposition of these ideas (Winograd, 1972). The program was very successful, breaking much new ground, particularly in the realm of integration of syntax and semantics. The use of systemic grammar facilitated this interaction.

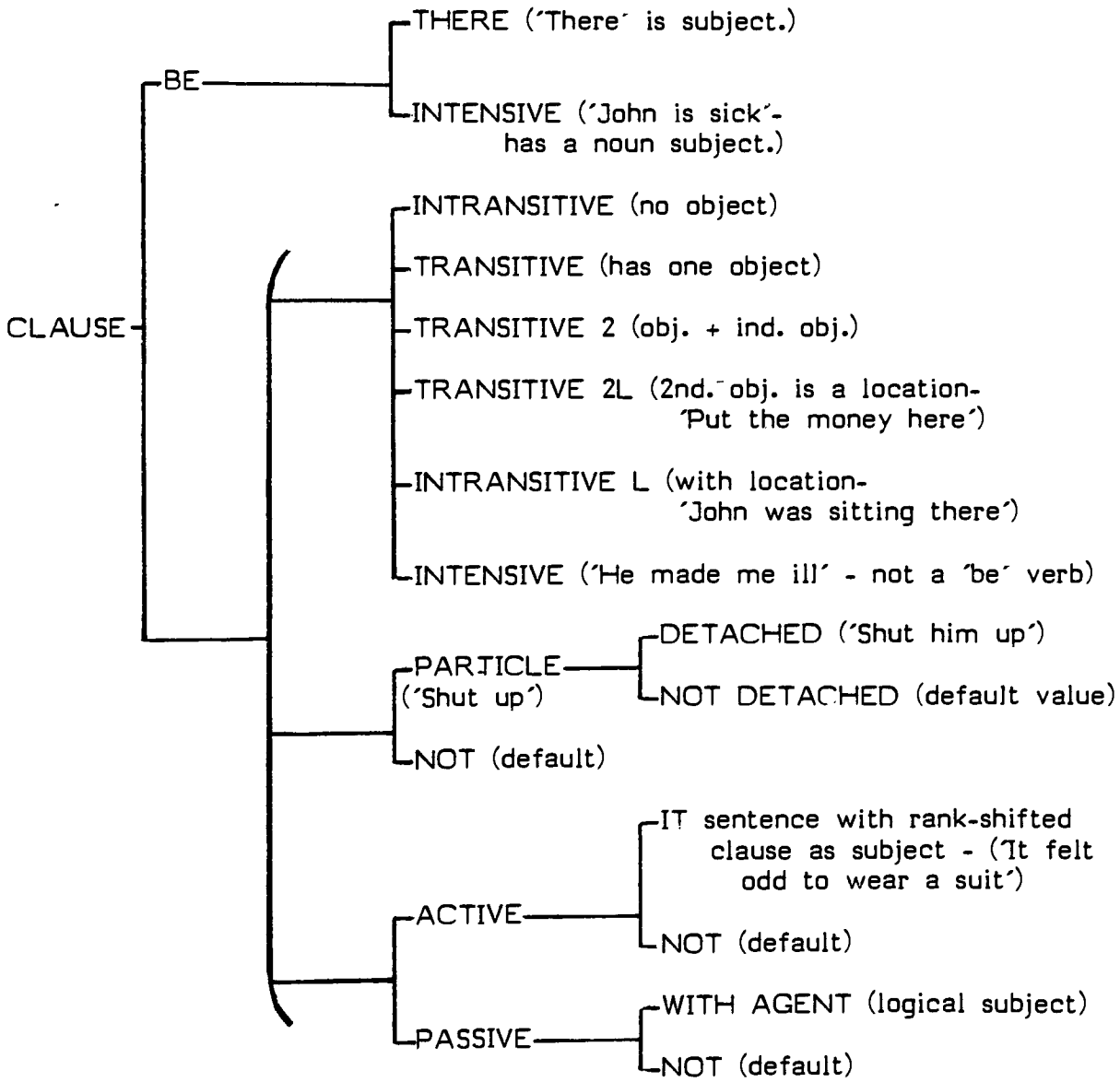
SHRDLU describes a 'blocks world' where colored blocks and other geometric solids are manipulated on a graphics screen by a 'robot' arm. The system has a very thorough knowledge of its 'micro-world', and can answer questions such as 'How many green blocks are on the red block'

and 'Can you put the large block on the green pyramid ?'. The semantic component is 'smart' enough to know that a block can not be balanced on a pyramid. Sample dialogue is described in detail in several places (for example, Winograd, 1972, pp. 8-15; Winston, 1977, pp. 157-178).

In the late 1960's, people were becoming aware of the necessity for interaction between the semantic and syntactic components of a natural language understanding system. Woods' LUNAR (Woods, 1972) utilized this idea. LUNAR parsed a sentence syntactically, and then examined the semantics to see if the parse made sense. Winograd's 'block-world' program, however, sustained a higher level of this semantic/syntactic interaction. SHRDLU did not wait until an entire sentence was parsed before checking semantics, but instead checked for semantic consistency after each phrase was built. When a syntactic ambiguity arose, it took the most likely alternative. When a semantic ambiguity arose, however, it carried along all possible interpretations in parallel, ruling out most of these parses as processing proceeded.

This type of syntactic/semantic interaction fit nicely with the ideas in systemic grammar. Systemic grammar, with its feature networks, made explicit those things that might have been left unstated by another grammar. A good example of this is the transitivity system. As a clause was parsed, a feature list was built that stated whether or not the clause was a 'be' verb (and hence took a predicate nominative or predicate adjective). If not, it determined whether the verb was transitive or intransitive; if transitive, whether it took one or two objects; etc.

The system network for clause transitivity used by Winograd (Winograd, 1972, p. 54) follows. Note that square brackets delimit a system from which only one of the options can be chosen. Rounded brackets allow a choice of one item from each of the systems bracketed.



Clause Transitivity System Network

The advantage of a system like this is that all the relevant information about the current status of the sentence is stated explicitly. The semantic component has all the information that it needs in a feature list and so does not have to do more work to get these facts.

As an example, consider the sentence: 'Put the money in the bag.' The dictionary entry for 'put' will indicate that it has the transitivity feature that allows a location as second object. That feature will rise from the 'word' rank to the 'clause' rank, and appear on the clause feature list. When 'in the bag' is being parsed, the semantic component will check to see if that can be construed as a location. Since it can be, the phrase will be accepted. The ready availability of this type of information (in feature lists) makes the interaction between the syntactic and semantic components easier to accomplish.

The following sample sentence parse shows some of the features that Winograd used. The sentence is a simple one, but it shows how the accumulated list of features is used to guide the parsing.

'Put some pennies in the red bag.'

The parser is top-down, or predictive, and it starts by assuming that a major clause will be the first clause-level component. The parser calls the CLAUSE program with features initialized to (CLAUSE MAJOR).

The first word is a verb, 'put'. A verb can be used to start a question or an imperative sentence. A verb that is used to start a question is usually a 'be', 'have', or 'do' verb. The parser will choose 'imperative' over 'question' in this case, so the features (VG IMPER) will be added. The parser now tries to interpret the sentence as an imperative. An

imperative sentence usually begins with the untensed (infinitive) form of the verb. 'Put' satisfies this test, so the features of the verb 'put' are added to the feature list. At this point the list contains:

```
(CLAUSE MAJOR)
    (VG IMPER)
        (VB MVB INF TRANSL)
```

The feature 'MVB' means that it is a main verb, and 'TRANSL' indicates that this verb can take two objects, one of which is a location.

The CLAUSE program now adds the feature IMPER to the clause's feature list, and sets off to find the first (direct) object.

The NG (noun group) program builds the object:

```
(NG OBJ OBJ1 DET INDEF NS)
    (DET INDEF)                    'some'
    (NOUN NPL)                     'pennies'
```

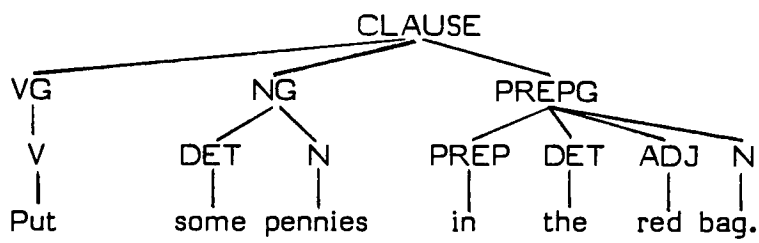
CLAUSE then looks for the location (LOBJ) that the feature TRANSL allows. It finds the prepositional phrase 'in the red bag' filling that role. It creates the feature lists:

```
(PREPG LOBJ)                       'in the red bag'
    (PREP)                         'in'
    (NG OBJ PREPOBJ DET DEF NS)    'the red bag'
        (DET DEF NPL NS)           'the'
        (ADJ)                      'red'
        (NOUN NS)                  'bag'
```

This phrase is accepted in the LOCATION role, providing a final structure of:

```
(CLAUSE MAJOR IMPER TRANSL)                    'Put ... bag.'
    (VG IMPER)                                 'Put'
        (VB MVB INF TRANSL)                    'Put'
    (NG OBJ OBJ1 DET INDEF NS)                 'some pennies'
        (DET INDEF)                            'some'
        (NOUN NPL)                             'pennies'
    (PREPG LOBJ)                               'in the red bag'
        (PREP)                                 'in'
        (NG OBJ PREPOBJ DET DEF NS)            'the red bag'
            (DET DEF NPL NS)                   'the'
            (ADJ)                              'red'
            (NOUN NS)                          'bag'
```

One interesting consequence of systemic grammar that was mentioned before is that parse trees tend to be flat. A parse tree for the sentence above is:



Each of the nodes (at each of the levels) has a feature list attached to it. The three levels (clause, group, word) are clear, and the words are grouped into meaningful units.

How does systemic grammar compare to the other formalisms for natural language recognition that have been described? Phrase structure grammars could handle the basic elements of systemic grammar, although phrase structure grammars would tend to do implicitly the things that systemic grammar makes explicit. An extended phrase structure grammar, or regular expression grammar, would be a clearer representation of

systemic grammar's levels. For instance, the prepositional phrase grammar could be stated as:

PREPG --> PREP (DET) (ADJ)* N

This representation, where a phrase can be thought of as a series of 'slots' with certain slots being optional (such as adjectives) and others being mandatory (such as the noun slot), is utilized in the systemic grammar based SLOT grammar described in the next chapter.

Phrase structure grammars are inadequate to handle all of systemic grammar's complexities because context-sensitive rules are needed. Transformational grammar adds context-sensitive rules to a base grammar by means of transformation rules; augmented transition networks use registers and tests on arcs; systemic grammar uses feature lists in the way that ATNs use registers. Systemic grammar's parsing algorithm can be driven by the contents of the feature lists. Systemic grammar, then, is as powerful as transformational grammar or augmented transition networks; it just chooses to emphasize different aspects of natural language.

CHAPTER 12

SLOT grammar

A recent effort in syntactic parsing by computer is the SLOT grammar developed by Michael McCord at the University of Kentucky (McCord, 1980). It uses systemic grammar for the recognition of syntactically acceptable sentences. The systemic grammar used is a form developed by Hudson called 'daughter dependency' grammar (Hudson, 1976). This grammar defines two types of dependencies: daughter dependencies and sister dependencies.

Daughter dependencies relate word categories that can be used together to create a phrase. For example, 'determiner', 'adjective', and 'noun' are all daughters of 'noun phrase'. 'Noun phrase' is considered to be the 'mother' of all those words that constitute the phrase.

Sister dependencies state information about completing verb phrases. It tends to be very 'word specific', since certain verbs have very specific requirements about what types of phrases or clauses are acceptable as object fillers. The verb 'believe', for example, has a sister dependency 'complement' because it can take a verb complement, as in 'I believe John will be here'. A verb like 'destroy' on the other hand will only have 'object' as its sister dependency.

There are two other important concepts used in McCord's SLOT grammar. First, 'slots' (as representations of grammatical relations) are

used to organize and build phrases. Slots list the possible fillers for phrases, and they include both optional and required word categories. A list of available slots is maintained for each word group or phrase in the sentence, and phrases grow by filling these slots with other adjoining words or phrases. Second, although the parser proceeds from left to right, the phrases are constructed in a 'middle-out' fashion. For example, the phrase 'the brown dog in the yard' is built beginning with the head word 'dog'. After accepting 'dog' as the main word in the phrase, 'the brown' is then appended to it on the left side, and the prepositional phrase 'in the yard' is added on the right side. McCord believes that 'slots' and 'middle-out' phrase construction are a more 'natural' representation of how humans build phrases.

When parsing, the main device used to control the building of phrases is the available slots list (ASLOTS). The head word in each phrase has attached to it a list of possible slots to be filled. These are the 'daughter dependencies' from Hudson's version of systemic grammar. For example, a noun phrase (NP) can be constructed using a DETerminer, any number of ADJectives, and any number of prepositional phrases (called REL by McCord). The list of which slots may be associated with which types of phrases is stated in the grammar program SYNTAX, of which an example will follow. When a noun is parsed, a phrase frame is created with the noun as its head word. Phrase frames are the central data structures used by the parser. These frames are lists of registers (e.g., ASLOTS) and their contents or values. Frames will be discussed in more

detail later on.

Consider the initial ASLOTS register of a noun phrase. It will look like this:

(DET ADJ* REL*),

where these are slots that are available to be filled as parsing progresses. Most slots are optionally filled, and they are removed from the ASLOTS list as they are used. Certain slots (those followed by a *) are multiple slots, and they are not removed from the ASLOTS list when filled, since you need to be able to account for situations like noun phrases that have more than one adjective.

The ASLOTS list is custom built for verb phrases, using the sister dependencies listed in the system dictionary. Simple transitive verbs will have an OBJ slot in their ASLOTS list, whereas a verb like 'give' will list both OBJ and IOBJ (indirect object). SYNTAX will list only the default slots, i.e. those slots that all verbs have. As an example, the verb 'fly' would have the sister dependency 'object' in its dictionary entry. The ASLOTS for 'fly' would be initialized as:

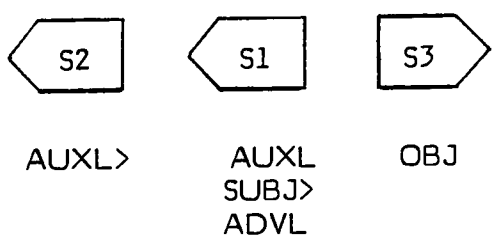
(BINDER SUBJ AUXL* ADVL* OBJ)

This is a very direct way to deal with the 'data-driven' aspects of natural language processing. Rather than setting a 'transitive' flag that needs to be carried along through the parsing, ASLOTS is a direct representation of what words and phrases should be sought (e.g., OBJ), and what things do not need to be searched for (e.g., IOBJ).

ASLOTS also provides a method for handling 'raising' constructions, such as 'which goal' in the sentence 'which goal did the referee claim

John scored ?'. Sentence constituents that are located 'out of place' in the sentence can be built and put into a slot later, at the time when filler is sought for that slot.

As mentioned before, McCord's SLOT grammar builds phrases using a 'middle-out' construction method. Consequently, it is necessary to keep track of what stage has been reached in the building of a phrase. It is also necessary to know at each stage what types of words or phrases are acceptable for continuing the building of the phrase, and whether or not they should be looked for on the left or right side of the phrase being built. This grammar uses a register STATE, attached to the phrase frame, to control this process. Directions for how to proceed from any state in a given phrase are contained in SYNTAX, a LISP program that contains the grammar. SYNTAX also can be described diagrammatically. An abbreviated SYNTAX diagram for verb phrases follows:



This grammar has three states and builds phrases in a 'middle-out' fashion starting at state 1 (S1). Each state has one or more categories attached to it; these categories indicate what words or phrases may be accepted in each state. 'State' also guides the acceptance of new words

by indicating from which side of the head word these new words may be received. For example, if a head verb (V) has been accepted, the phrase frame for the verb phrase will begin to be constructed starting at state S1. The parser will have to look to the left of the verb (note that S1 points to the left <) for an AUXL verb, a SUBJ noun phrase, or an ADVL adverb. In state S1, only SUBJ has a '>' following it. The '>' indicates an advancing state. Parsing can not advance beyond state S1 (while accepting words to the left of the main verb) until a SUBJ has been found. It is possible to add many ADVLs and AUXLs (always continuing to take words from the left side of the currently active phrase), but these do not advance the state from S1 to S2. Only after SUBJ has been found will the state advance to S2.

It may seem confusing at this point to be referring to a left-to-right parser that builds phrases in a middle-out fashion. The parsing ostensibly moves from left to right, but the phrases are built by starting at a head word and adding words and phrases from the left and from the right. The analysis presented here assumes some pre-processing of phrases, at least to the extent of identifying head words (which are usually nouns or verbs) and building the initial ASLOTS lists (after consulting the dictionary for sister dependencies). These examples are designed to show how the parser uses states and slots in accepting sentences. Consider another example, this time a question:

'Is Jim going to drive the car ?'

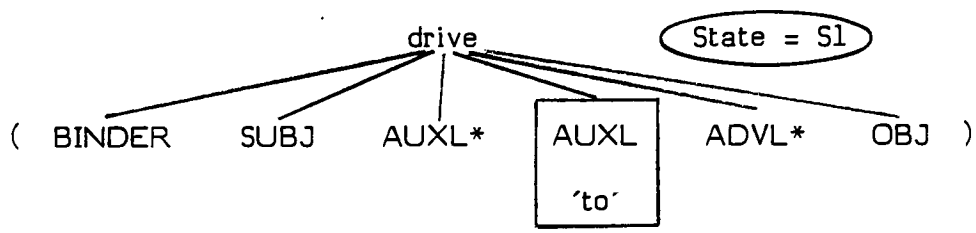
The words in this sentence can be labelled:

Is	Jim	going	to	drive	the car
AUXL	NP	AUXL	AUXL	HEAD VERB	NP

The phrase frame for the verb phrase with head verb 'drive' will contain the sister dependency OBJ and the default available slots list:

(BINDER SUBJ AUXL* ADVL*)

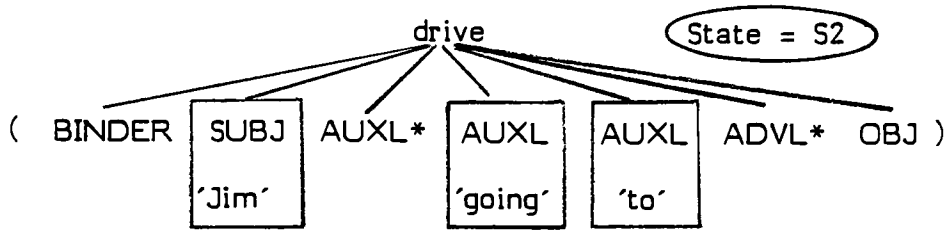
The main verb phrase parsing begins at state S1 accepting 'drive' as the head verb. S1 says 'look left for either an AUXL, SUBJ, or ADVL'. 'To' is accepted from the left as an AUXL. The boundaries of the verb phrase are expanded to cover 'to drive', and ASLOTS has been altered as follows:



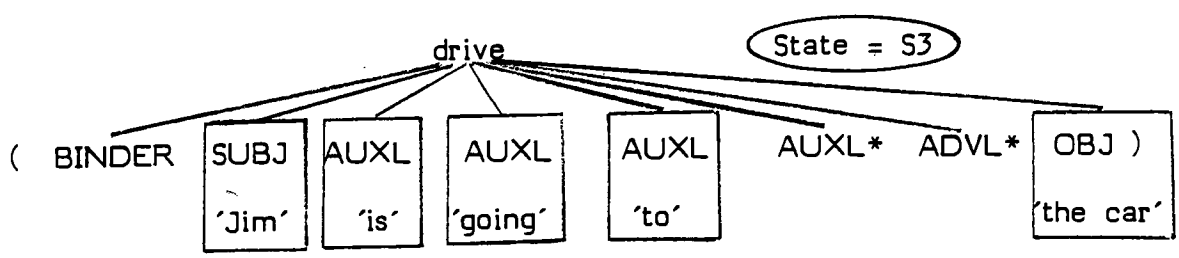
Boxes around the ASLOTS that have been filled are used here to indicate that these slots are now filled and no longer available. These filled slots are placed in a new list (FSLOTS) of filled slots.

Adding an AUXL at S1 does not advance the state, so remaining in S1 and looking to the left again, the AUXL 'going' is encountered. Another AUXL slot in ASLOTS is filled. Control remains in S1, looking left again for more new acquisitions. 'Jim', a noun phrase, is found. Since a noun phrase is the required filler for SUBJ, that slot is filled. SUBJ is an advancing state in S1, so control passes to S2. ASLOTS now

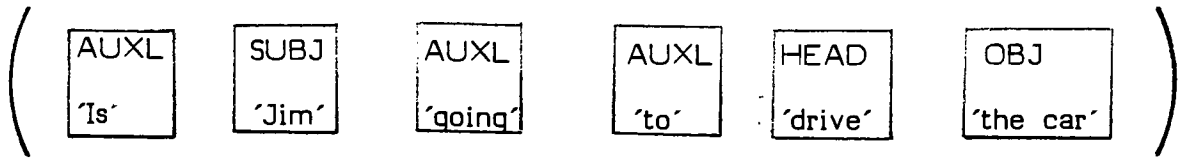
has this configuration:



S2 looks to the left for an AUXL, and finds 'is' there. The verb phrase boundaries have been expanded at this point to encompass all the words from 'is' to 'drive'. AUXL is an advancing state in S2 (note that AUXL is not an advancing state in S1, however), so the state advances to S3. S3 looks to the right for an OBJ filler. The noun phrase 'the car' is found there. This is the correct filler for the OBJ slot, so that slot is filled. Note that this simplified syntax diagram does not have a state with IOBJ (indirect object) as a syntactic category. For this sentence, even if there had been a state accepting IOBJ (as does a more complete SYNTAX listed later), this parse would not look for an IOBJ. The parse is driven by the ASLOTS list, and since the verb 'drive' has only OBJ as a sister dependency, an IOBJ will never be sought. The final ASLOTS list is:



The filled slots would have been put in FSLOTS maintaining the sentence word order. FSLOTS would be:



Adding 'the car' to the verb phrase extended the right boundary of the verb phrase to the end of the sentence. To terminate a parse in an accepting state, it is necessary to have a verb phrase that spans the entire sentence, as this one does.

Note also that it is not necessary to add a word at every state. State 2 is used only by questions. Declarative sentences pass through state 2 because after accepting the subject, there are no more words to be found to the left. In the complete version of verb syntax that McCord provides (McCord, 1980, p.41), finding an AUXL in state 2 has the effect of adding the feature 'question' to the phrase frame for the main verb.

All the parsers described so far in this paper have been top-down. The parser that is used for the SLOTS system is bottom-up. Because a bottom-up parser does not have the benefit of a general overview of the parsing in progress (the predictive aspect of top-down parsing), it is usually necessary for bottom-up parsers to construct every possible intermediate constituent in the course of a parse. As a consequence, bottom-up parsers are often inefficient.

To provide some insight into how a bottom-up parser typically constructs intermediate constituents, the next chapter informally describes the Cocke-Younger-Kasami algorithm, a tabular bottom-up parsing method

applicable to context-free grammars. This is followed by a thorough description of McCord's SLOT parser, which builds intermediate constituents in much the same way as the Cocke-Younger-Kasami algorithm.

CHAPTER 13

The Cocke-Younger-Kasami algorithm

Bottom-up parsing is sometimes called 'data-driven' parsing because it starts at 'the bottom', or the leaves of the parse tree (the input string). In bottom-up parsing, these leaves are assembled into intermediate constituents using the rules of the grammar. The intermediate constituents thus formed are then combined, and so on until the 'start' symbol is reached. Bottom-up parsers are sometimes called 'well-formed substring' parsers, as these intermediate constituents are legal expressions derived using the production rules. In bottom-up parsing, the input string leads you back to the 'start' symbol. In top-down, or 'predictive' parsing, the 'start' symbol is the beginning point and parsing proceeds toward the leaves.

The Cocke-Younger-Kasami algorithm parses bottom-up. Its efficiency is $O(n^3)$ for all cases. It is described here because it builds intermediate constituents in much the same way that the parser for McCord's SLOT grammar does. A number of linguists also believe that humans parse sentences in a bottom-up fashion, or at least that the human mechanism has significant bottom-up aspects.

The Cocke-Younger-Kasami algorithm is tabular, creating a parse table of all the possible intermediate constituents that the input string can generate. The algorithm is described most simply working on

context-free grammars that are in Chomsky normal form, with no empty productions.

The table is constructed in the following way:

(1) The bottom row (row 0) contains all the terminal symbols in the input string.

(2) Row 1 contains each non-terminal that can be derived from the terminal symbol immediately beneath it.

(3) The presence of a non-terminal symbol in any other position in the table indicates that this symbol derives all the symbols below itself and also the ones diagonally below it to the right. These other symbols may be derived either directly (in one step from a production) or through other intermediate constituents.

Consider the diagram below. The presence of 'C' in the table implies that 'C' can derive 'A' (which is directly below it) and 'B' (which is diagonally below it to the right), by means of a rule such as
C --> A B.

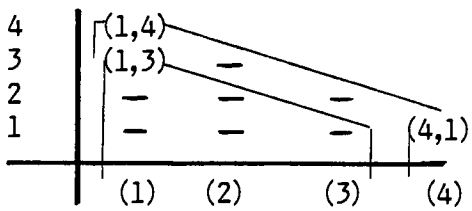
row 2	C	
row 1	A	B
row 0	'a'	'b'

As parsing progresses, the algorithm tries all the combinations that might derive constituents that are below it and to the right. For example, to fill table entry t(1,4) in the table below, three attempts are made

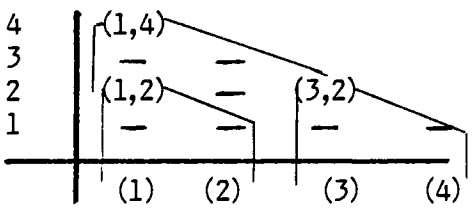
to combine intermediate constituents.[1]

row 4	t(1,4)			
row 3	t(1,3)	t(2,3)		
row 2	t(1,2)	t(2,2)	t(3,2)	
row 1	t(1,1)	t(2,1)	t(3,1)	t(4,1)
row 0	a(1)	a(2)	a(3)	a(4)

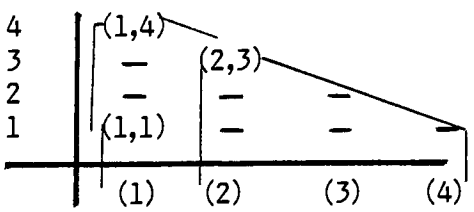
The three diagrams below represent the attempted combinations:



t(1,4) --> t(1,3) t(4,1)



t(1,4) --> t(1,2) t(3,2)



t(1,4) --> t(1,1) t(2,3)

To see how this table is constructed in practice, consider the context-free grammar:

S --> B C

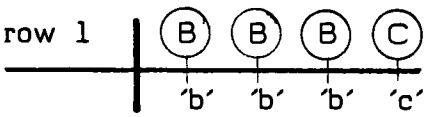
[1]Using Aho and Ullman's labelling convention for the Cocke-Younger-Kasami algorithm, table element t(c,r) occupies column 'c' and row 'r'.

B --> B B

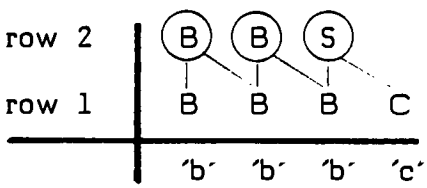
B --> 'b'

C --> 'c'

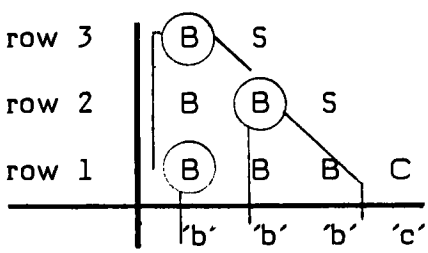
The string 'bbbc' would build a parse table in this way.



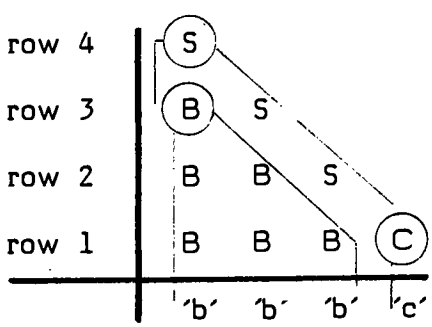
These are derived directly using productions B --> 'b' and C --> 'c'.



Row 2 is constructed using row 1.
t(1,2) is from B --> B B
using the B's at t(1,1) and t(2,1).
S(3,2) is from B(3,1) and C(4,1)



The B at (1,3) is from (1,1) and (2,2). Note that (1,3) tries to combine (1,2) and (3,1), without success.
S(2,3) is from B(2,2) and C(4,1).



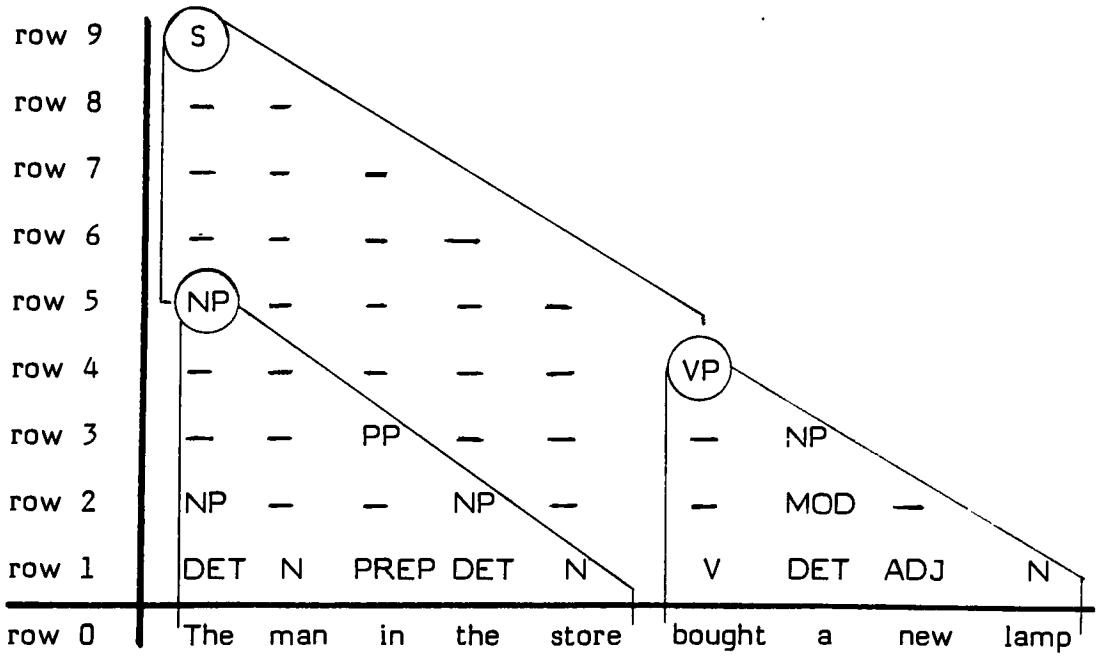
S(1,4) is derived from B(1,3) and C(4,1).

The presence of 'S' in position (1,4) indicates a successful parse.

The following example is drawn from natural language processing. This grammar is weakly equivalent to the one used previously as an example for Earley's algorithm, and the same sample string is used. Note that the grammar is in Chomsky normal form.

```
S       -->  NP  VP  |  N   VP
VP      -->  V   NP  |  V   N
PP      -->  P   NP  |  P   N
NP      -->  NP  PP  |  N   PP
        -->  DET N   |  MOD N
MOD     -->  DET ADJ
ADJ     -->  ADJ ADJ
N       -->  man  |  lamp  |  store
V       -->  bought
ADJ     -->  new
PREP    -->  in
DET     -->  the  |  a
```

The final parse table is given below. It does not describe all the unsuccessful attempted pairings. Note that the final accepting state S(1,9) was derived from NP(1,5) = 'the man in the store' and VP(6,4) = 'bought a new lamp'.



The table shows all intermediate constituents that were derived in the course of this bottom-up parse. By maintaining this record of all possible well-formed substrings, the intermediate steps are made clearer

and unnecessary reduplication of parsing is avoided.

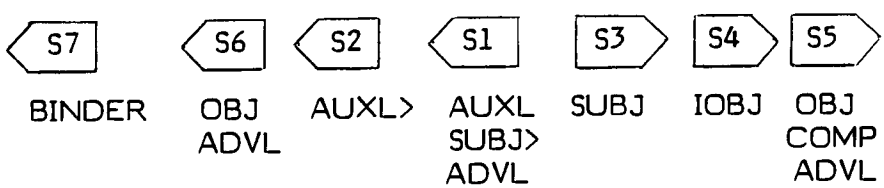
CHAPTER 14

SLOT grammar parser

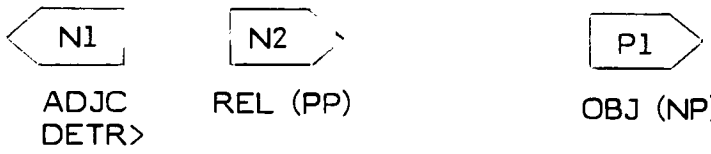
McCord's algorithm builds a chart of intermediate constituents in much the same way that the Cocke-Younger-Kasami algorithm does. The grammar is not a context-free grammar, however, so the mechanics of the actual parsing differ.

Below are syntax diagrams for a simplified English grammar. The grammar can handle sentences like 'Which house does Jim own?', and 'The man in the store thought you were going to buy a chair'.

The verb phrase grammar that follows is capable of handling more complex constructions than the simple verb phrase grammar given earlier. State S3 accepts constructions in which the subject occurs after the main verb, such as 'Is the man happy?'. The BINDER category (at S7) is for subjunction ('if', 'whether', etc.), and COMP is used to accept verb complements at state S5.



Verb Phrase Grammar
Default Slots: BINDER SUBJ AUXL ADVL



Noun Phrase Grammar
Default Slots: ADJC
DETR
REL

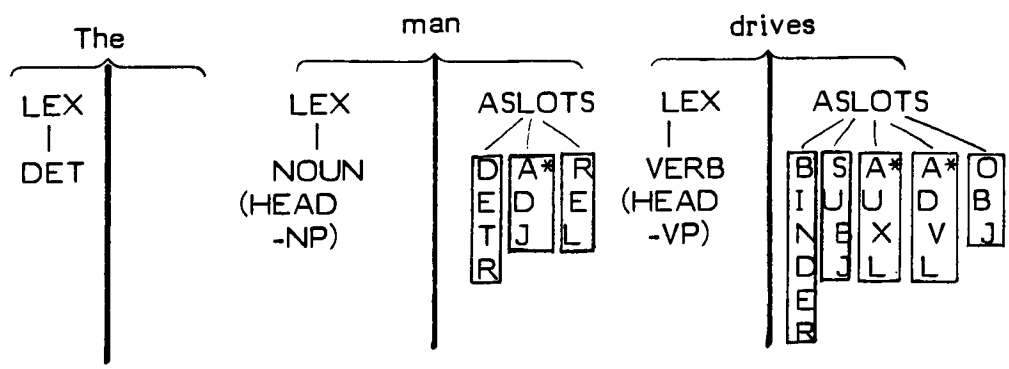
Prepositional Phrase Grammar
Default Slots: OBJ

When a sentence is parsed, an initial pass is made to set up lexical information for each word in the sentence. A phrase frame is set up for each word that can be the 'head word' in a phrase. The phrase frame contains an initial list of available slots (ASLOTS) for these head words, retrieving this information from the sister dependencies in the dictionary and the default slots in the grammar.

Examine the simple sentence:

'The man drives.'

After the preliminary pass has been made, the following initial structure will have been built:



It is not necessary to gather this preliminary information in a separate pass; it could be done while the words are accepted in the left-to-right parse. Separating these two steps, however, clarifies the

operation of the parser itself.

The most important part of the parsing algorithn is a function named MODIFY. The form of the function call is:

(MODIFY NEWFRAME HEADFRAME DIR)

This constructs all frames resulting when NEWFRAME modifies (or fills a slot in) HEADFRAME, coming from direction DIR. (DIR = LEFT means that NEWFRAME is on the immediate left of HEADFRAME, and DIR = RIGHT means that NEWFRAME is on the immediate right of HEADFRAME).

For every slot in the ASLOTS of HEADFRAME, MODIFY determines whether or not NEWFRAME can fill the slot. If it can, HEADFRAME is updated in the following way:

(1) The slot that can be filled is set equal to NEWFRAME.

(2) ASLOTS is modified as necessary.

(3) STATE is advanced if necessary.

(4) If DIR = LEFT, then LB (left boundary) of HEADFRAME is set equal to LB of NEWFRAME. If DIR = RIGHT, then RB (right boundary) of HEADFRAME is set equal to RB of NEWFRAME.

The old version of HEADFRAME is saved for possible later use, and another function named INSERT is called to 1.) enter this new version of HEADFRAME, and 2.) see if this new version can interact with any of the other phrase frames that currently exist (and are contiguous with HEADFRAME).

The function INSERT needs access to a list generated by the controlling function PARSE. PARSE creates boundary markers for each word as it goes along. Each of these boundary markers has a property list, RESULTS, associated with it. RESULTS lists all the word and phrase frames whose right boundary is the current boundary marker. INSERT does its updating by:

(1) putting the current frame (the new HEADFRAME) on the RESULTS list of its right boundary, and

(2) for every frame in RESULTS-LB (the RESULTS list of the left boundary of the current frame), calling:

(MODIFY HEADFRAME RESULTS-LB RIGHT)

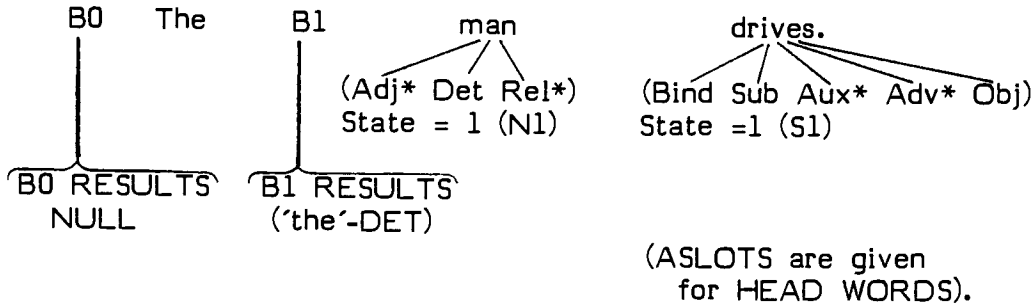
(MODIFY RESULTS-LB HEADFRAME LEFT)

This tries to use HEADFRAME as filler for all the frames listed in the left boundary RESULTS list to fill slots in HEADFRAME.

PARSE oversees and guides the building of intermediate constituents. It proceeds from left to right, one word at a time, setting up new boundary markers as it goes. For each word encountered, INSERT is called to find all the different ways this new word might possibly interact (fill slots) in previously existing frames. Note the recursion present - INSERT calls MODIFY, and MODIFY calls INSERT. The recursion halts when no more modifications can be made (in which case INSERT is not called).

Once again, consider the sentence 'The man drives'. PARSE accepts the first word ('the') and sets up a boundary (B0) to the left of the word,

and another (B1) after the word. The word frame 'the-DET' is put on the RESULTS list of B1 (as the only word or phrase encountered so far whose right boundary is B1), and B0 is set equal to NULL. INSERT is going to be called now. The situation before the call to INSERT is:



INSERT is called. 'The' is already on the RESULTS list of B1 (put there by PARSE), so two modify calls are made:

(MODIFY 'The' B0-RESULTS RIGHT)

(MODIFY B0-RESULTS 'The' LEFT)

Since the B0 RESULTS list is NULL, no slot fillings are possible, and PARSE proceeds to 'man'. Boundary marker B2 is created between 'man' and 'drives'. The entry 'man-HEAD NP' is put on the RESULTS list of B2, and INSERT makes its two calls to MODIFY.

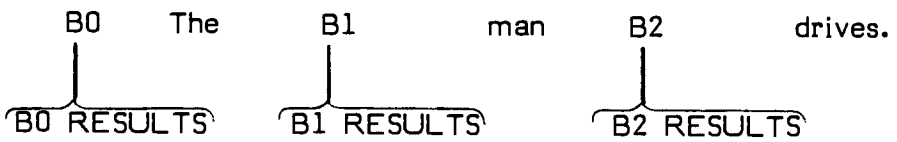
(MODIFY 'man' 'the' RIGHT)

(MODIFY 'the' 'man' LEFT)

The first call to MODIFY fails (since 'man' can not fill a slot in 'the'), but the second one works - 'the' can fill the DET slot in 'man-HEAD NP', since NP is in state 1 (N1), allowing a DET for filler. This creates 'the man', a noun phrase (NP) with ASLOTS = (ADJ* REL*). The state is advanced to N2 (noun phrase state 2), since DET is a state

advancer in N1. The left boundary (LB) of the phrase 'the man' is set to B0 (the LB of 'the'). MODIFY calls INSERT recursively on this newly created frame, to see if it can fill a slot in any other adjacent frame (and vica versa), but since there are no words on the left of 'the man', and since the word on the right of 'the man' has not yet been parsed, all that can be done now is to add the new frame 'the man' to the RESULTS list of B2.

A complete description of the sentence as parsed so far is:



NULL

'the-DET'
LB=B0, RB=B1

'man-HEAD NP'
LB=B1, RB=B2
ASLOTS=
(ADJ* DETR REL*)
STATE = N1

'the man-HEAD NP'
LB=B0, RB=B2
ASLOTS=
(ADJ* REL*)
STATE = N2

PARSE now accepts 'drives', creating boundary marker B3 at the end of the sentence. INSERT is called on both verb phrase frames in B2, first on 'man-HEAD NP':

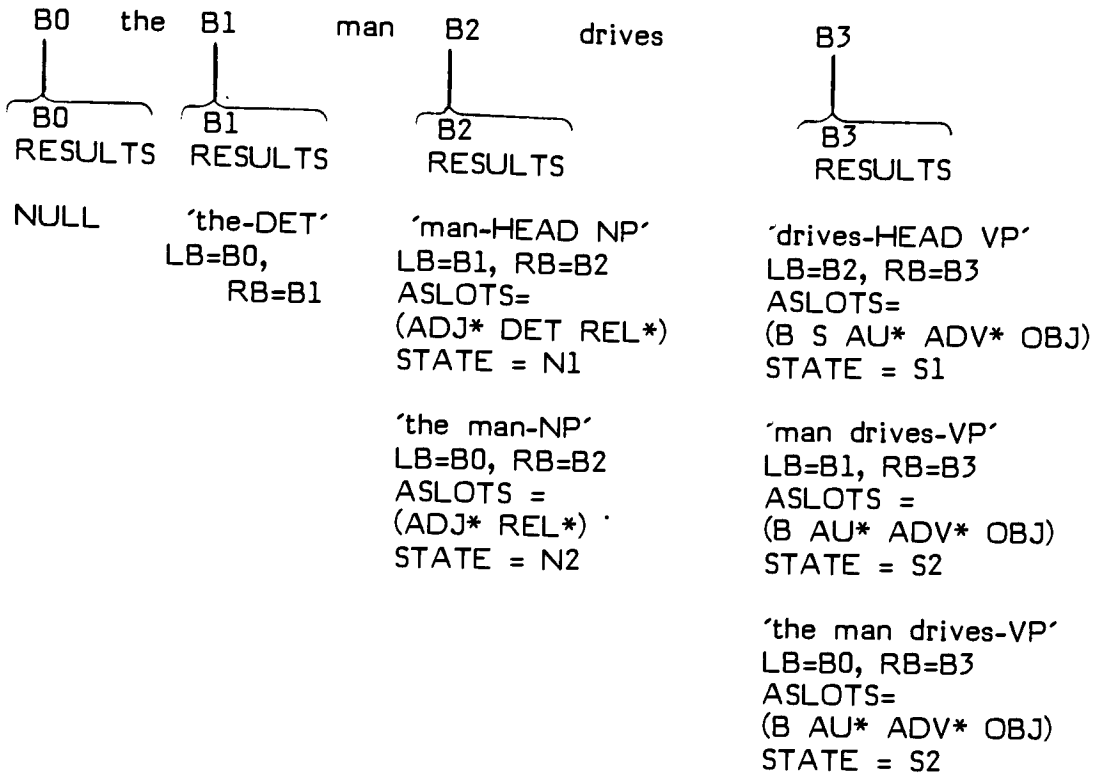
> INSERT asks "can the VP 'drives' be filler in the NP 'man'?" The answer is clearly 'no'. It then asks if the NP 'man' can be filler in the VP 'drives'. The answer is 'yes', it can. It can because the VP is in state 1 (S1), and a NP is allowed for filler of the SUBJ slot in ASLOTS (if the NP is found on the

left side of the VP). As a result, a new VP 'man drives' is added to B3. INSERT is called once again (recursively) on the RESULTS list of B1. Since 'man drives' can not modify the DET 'the', and since 'the' is not able to fill a slot in the ASLOTS of VP 'man drives' (a VP can not take a DET as filler - only a NP can), the recursion halts after adding the VP 'man drives' to B3.

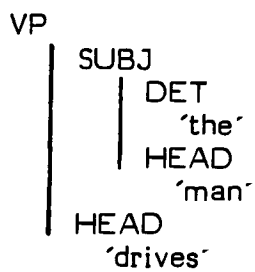
INSERT is now called on the second entry (the NP 'the man') in the RESULTS list of B2, with these results:

Can the VP 'drives' fill a slot in the NP 'the man'? No. Can the NP 'the man' fill a slot in the ASLOTS of the VP 'drives'? Yes. We are in state 1 of the VP looking for an AUXL, SUBJ, or ADVL. Since a NP can be filler for SUBJ, 'the man' is accepted, creating a VP 'the man drives'. The phrase is put on the RESULTS list of B3 during the first recursive call, and the LB of the phrase is set to B0. The recursive calls all fail, since LB=B0 and B0 RESULTS = NULL.

There are no more words to process, so PARSE checks to see if it is in an accepting state. It is, since there is a VP that spans the entire sentence. The final boundaries and their RESULTS lists are:



A parse tree like the following would be printed:



Assume that the paragraph had been a little more complicated. For example, 'The man drives cars'. The processing would be unchanged up through the word 'drives'. Upon accepting the word 'cars', PARSE would create a new boundary marker (B4), and then call INSERT for each of the three frames on the B3 results list. Since each of the frames on B3 RESULTS are verb phrases, they will not be able to fill slots in 'cars', a

noun phrase. The current word 'cars', however, will modify (fill a slot in) each of the three verb phrases on B3 RESULTS, as described below.

Since the word 'cars' is coming from the right and no words remain on the left, all the states with direction left will be jumped over. State S3 also is skipped because the SUBJ slot has already been filled. State S4 is jumped over because there is no ASLOTS entry for indirect object. Each verb phrase then is filled (or at least an attempt is made to fill each one) in these ways:

(1) The verb phrase 'drives' can accept 'cars' at S3, filling the SUBJ slot with a noun phrase. However, the verb phrase requires a singular subject, so no slot filling occurs.

(2) The verb phrase 'man drives' will accept 'cars' as OBJ at S5. It is able to get to S5 because its ASLOTS has SUBJ already filled, so the state advances through S3 and S4. As mentioned 'drives' will not have an indirect object as a sister dependency, so state S4 will not be considered. Advancing to state 5, the OBJ slot will be filled (noun phrase filler required).

(3) In like manner, 'the man drives' will accept 'cars' as OBJ at S5.

As a result of this, B4 RESULTS contains three new frames: 1.) the NP 'cars', 2.) the VP 'man drives cars', and 3.) the VP 'The man drives cars'. Since only the third frame spans the entire sentence, it would be output.

Notice how the phrase frames were built in a 'middle-out' manner. The noun phrase was able to be constructed completely (once the noun

had been parsed) by adding words found to the left of the head noun. The verb phrase was a slower process, adding words or phrases first on the left side of the verb, and then on the right side as those words were parsed.

The parsing algorithm that McCord used for this SLOT grammar is not very efficient. The constant rechecking of completed constituents, while typical of bottom-up parsers, detracts from some of the useful ideas presented in the grammar itself. It can be argued that slots and slot-filling may well be related to the technique that humans use to accept sentences, but it is unlikely that anyone would claim that human parsing is done using the parsing algorithm presented here.

There is a reasonable likelihood that people use some top-down strategies when parsing sentences. We frequently finish incomplete sentences, predicting what the next constituents will be. A recurring question in natural language processing is how can top-down and bottom-up parsing be combined in one system so as to utilize the 'predictive' aspects of top-down parsing as well as the 'data-driven' aspects (intermediate constituents) of bottom-up parsing. The following chapters examine recent work in this area.

CHAPTER 15

Combining top-down and bottom-up parsing

Both top-down and bottom-up parsing strategies have certain strengths and weaknesses. Top-down approaches have some psychological appeal. Humans seem to use "prediction" when understanding sentences. We also work on one "most likely" parse rather than developing all possible parses simultaneously. Computationally, top-down parsers tend to be easy to write, and the backtracking available in many of these methods is a convenient way to handle alternate parses. At the same time, backtracking word by word tends to be inefficient and time consuming, as well as psychologically unrealistic. A top-down parser also may not be able to provide much information about a parse that fails, since it will have backtracked its way to the beginning of the sentence.

Bottom-up parsers, on the other hand, can provide a lot of information about a failed parse, since successfully constructed phrases are available for examination. All possible phrases and constituents have been constructed, but there is no discrimination between likely and unlikely intermediate constituents. A related weakness is that a bottom-up parser tends to be redundant and inefficient, due to its enumerative method.

There are advantages and disadvantages to both top-down and bottom-up parsers. Neither method by itself can provide the most efficient solution to the natural language parsing problem. A parsing stra-

tegy has been sought that combines the best features of both top-down and bottom-up parsers.

Woods' ATN parser made some initial steps in this direction. The maintenance of the well-formed substring table (WFST) as parsing proceded yielded some of the benefits of bottom-up parsing, and it greatly eased the backtracking process. Since Woods' LUNAR program, however, other researchers have attempted to more completely integrate top-down and bottom-up parsing.

Typically, these hybrid parsing schemes use the bottom-up component to create phrases and other intermediate constituents, but without trying to construct every possible combination. The bottom-up component is guided in its attempts by the top-down component, which provides a constraining framework based on the results of the parts of the sentence already parsed. The net result of this marriage of parsing techniques can be a significant reduction in the number of parsing combinations attempted, and hence a more efficient parser.

One of the first attempts at directly combining top-down and bottom-up parsers for natural language analysis was undertaken by Vaughan Pratt at M.I.T. (Pratt, 1975). His program was called LINGOL, and was intended for language translation. While it contained a semantic component, the program's first concern was to produce a syntactic parse of the sentence. LINGOL used a modified form of Earley's algorithm for the top-down parsing, and a version of Cocke-Younger-Kasami for the bottom-up parsing.

The parsing of a sentence began with the bottom-up component deriving an initial phrase from the beginning of the input sentence. Any completed phrases were sent to the top-down component. Initially, the top-down component used these phrases to predict what were acceptable constructions that might follow. These predictions then limited the bottom-up component in its next attempt to parse an intermediate component.

As an example, consider the sentence "The boy in the car drove recklessly". The bottom-up component would send "The boy" to the top-down component as a noun phrase. The top-down component would use that information to make its predictions. In a simplified scheme, the top-down component might predict that a noun-phrase in this position in the sentence must be followed by another noun, a prepositional phrase, an adverb, or a verb phrase. All these predictions would remain "active", and the bottom-up component would then go back to work constrained by these predictions. These top-down predictions limited what components the bottom-up component could attempt to construct. This interaction would continue until the sentence was parsed.

This general method of combining top-down and bottom-up parsing strategies for natural language analysis is capable of effecting some real economies in the parsing. Pratt compared LINGOL's parsing algorithm to the basic Cocke-Younger-Kasami algorithm and found that the combined method built only one-fifth the number of nodes (partial parses) that the straight bottom-up approach created.

A more recent natural language parser that combines top-down and bottom-up parsing is the PARSIFAL program written by Mitchell Marcus at M.I.T. (Marcus, 1980). This parser, allowing only a very constrained look-ahead, does not allow any backtracking. It is also designed so that once a constituent in a sentence is built, it is permanent. Once a constituent is accepted, it can not then be discarded and rebuilt at a later time. Marcus calls this combination of features deterministic parsing. The structures and methods used in PARSIFAL are different from any of the others used up to this point. The next chapter examines PARSIFAL in some detail.

CHAPTER 16

PARSIFAL

In the late 1970's, Mitchell Marcus was a graduate student at M.I.T. He had a strong interest in linguistics, which he studied under Noam Chomsky.

Marcus concluded that natural language could be parsed deterministically, without recourse to either the backtracking of top-down approaches or the inefficient enumerative approach characteristic of many bottom-up parsing systems.

The parser Marcus constructed, called PARSIFAL (Marcus, 1980), combines top-down and botom-up techniques to achieve this. Two data structures are used: a last-in first-out stack of incomplete constituents called the active node stack, and a three cell buffer containing the grammatical constituents that have been built. Once a piece of the sentence is placed in the buffer, the parser is not allowed to change its mind and reparse it. The constituents in the buffer can be added to, but not destroyed.

Marcus does not claim that PARSIFAL can parse all sentences. He says that his program will parse all sentences "which people can parse without conscious difficulty."[1] It is able to succesfully parse (deterministically) some sentences that would require backtracking in, say, an ATN.

---

[1] (Marcus, 1980), p.6.

The parser was constructed to demonstrate some linguistic ideas. It uses trace theory, a recent development in transformational grammar that Chomsky advocates. Trace theory claims that when a component in a sentence is found out of place, there is a "trace" left in its original position to indicate the position shift.

The grammar is realized in a set of rule packets that are attached to the nodes on the stack, each packet containing a series of pattern match tests and associated actions. The rules in these packets are given priorities, so that if more than one of them is satisfied, the highest priority rule is chosen. The patterns are compared to the buffer contents, and if a match is found, the actions are performed. When these rules complete the construction of a constituent on the stack, the constituent is popped off the stack, and then either put in the buffer or attached to the new node at the top of the stack. The only rule packets that are "active" at a given time are those that are attached to the current active node (the top of the stack).

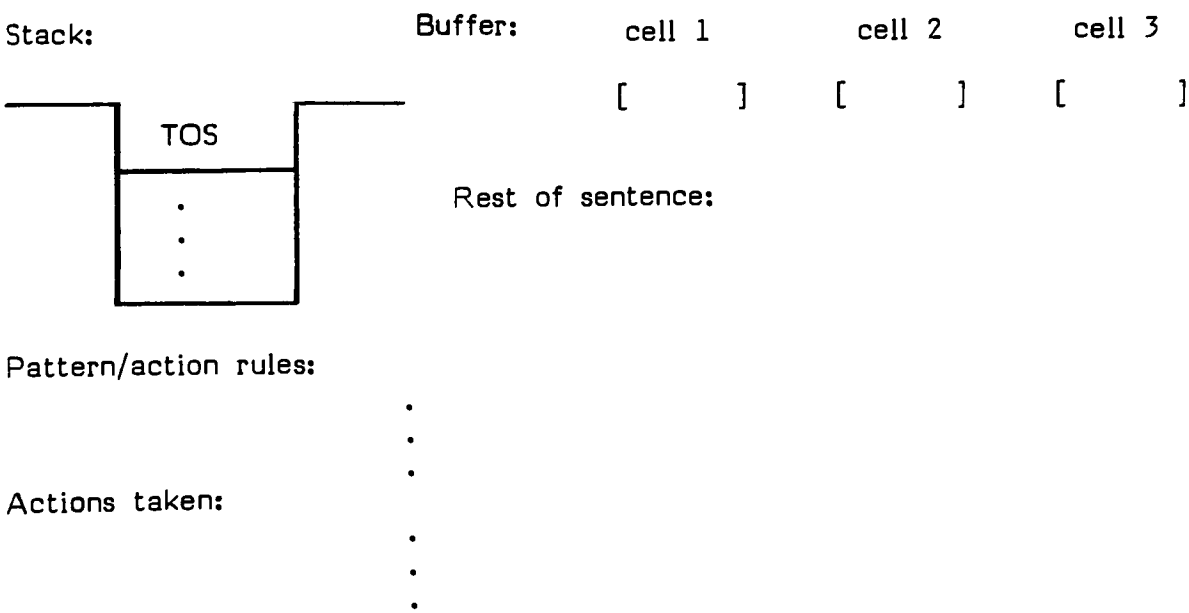
Generally, the top-down aspects of the parsing occur in the stack, with the rule packets operating in much the same way that phrase structure grammar rules do. These rules are "predictive", stating the possible constituents that would allow the parse to continue successfully. The buffer builds individual constituents from adjacent words and components, attempting to combine them into a higher level structure, just as a bottom-up parser does.

Following a parse of a sentence through all its steps is a good way to see how PARSIFAL operates. The simple example which follows does

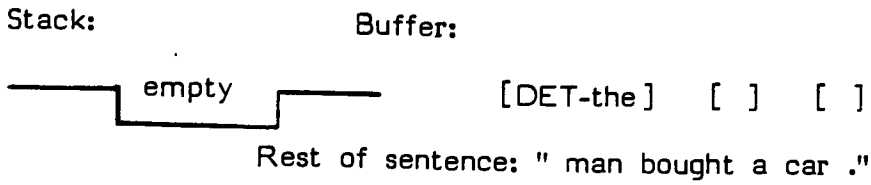
not show all the complexities of PARSIFAL's grammar or operation, but it does demonstrate the workings of the stack and the buffer, as well as their interaction.

The following format will be used to illustrate the states and actions in the parse. The stack and buffer cells will be shown, as below. The pattern/action rules associated with the node on the top of the stack will be stated, and the actions that are taken follow that.

The grammar rules used here are a subset of the ones that Marcus lists in Appendix D of (Marcus, 1980). Only the applicable rules will be mentioned here.

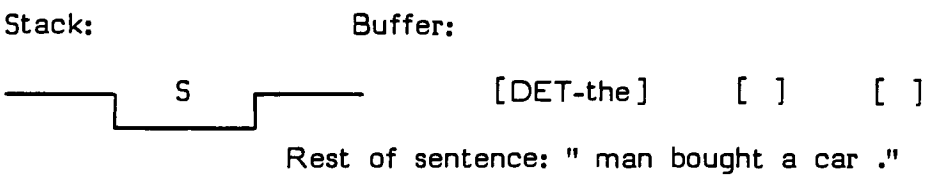


Consider the sentence: "The man bought a car." Initially, the parse is configured as below.



Pattern/action rules: For any word at the beginning of a sentence, create a new S (sentence) node and activate the SENTENCE-START and CLAUSE-POOL rule packets.

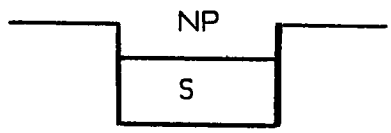
Actions taken: The new node is created. The rule packets are searched for a rule requiring a DET in the first buffer cell. The rule NGSTART in CLAUSE-POOL is satisfied. NGSTART needs a "noun group starting word" (e.g., N, PRO, DET, or ADJ). This is reflected in the state below.



Pattern/action rules: In S, if the word in buffer cell 1 can start a NP, then create a new NP node on the top of the stack. If buffer cell 1 is a DET, then activate rule packet PARSE-DET else activate rule packet PARSE-QUANT.

Actions taken: Buffer cell 1 is a DET, so a new NP node is created and PARSE-DET is activated. A rule requiring a DET in buffer cell 1 is found in PARSE-DET. The parse is in the state below.

Stack:



Buffer:

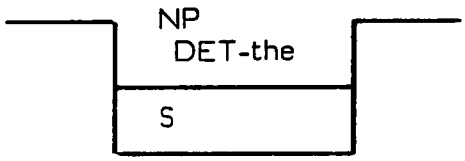
[DET-the] [ ] [ ]

Rest of sentence: " man bought a car ."

Pattern/action rules: The PARSE-DET rule packet is active. If buffer cell 1 is a DET, then attach the contents of buffer cell 1 to the currently active node. Deactivate PARSE-DET and activate the PARSE-QUANT rule packet.

Actions taken: Since that condition was met, buffer cell 1 [DET-the] is attached to NP on the stack, and PARSE-QUANT is activated.

Stack:



Buffer:

[N-man] [ ] [ ]

Rest of sentence: " bought a car ."

Pattern/action rules: If buffer cell 1 is a quantifier, then a set of actions is performed similar to those done for a DET. If buffer cell 1 is not a quantifier, then deactivate PARSE-QUANT and and activate PARSE-ADJ.

Actions taken: Since 'man' is not a quantifier, the second pattern/action rule given above is chosen. The configuration of the parse does not change, and PARSE-ADJ ends up meeting the same fate as PARSE-QUANT. The pattern/action rule for PARSE-ADJ is:

If buffer cell 1 is an ADJ, then attach it to the currently

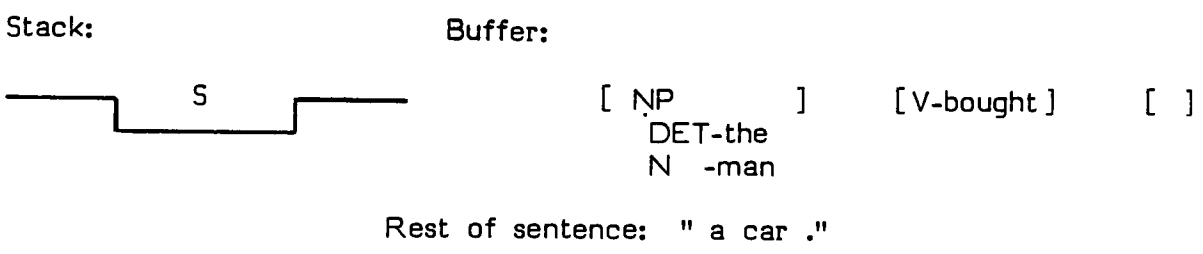
active node as an ADJ.

If buffer cell 1 is not an ADJ, then deactivate PARSE-ADJ and activate PARSE-NOUN.

The second option is the correct one, and PARSE-NOUN proceeds to find a noun in buffer cell 1. The associated actions are:

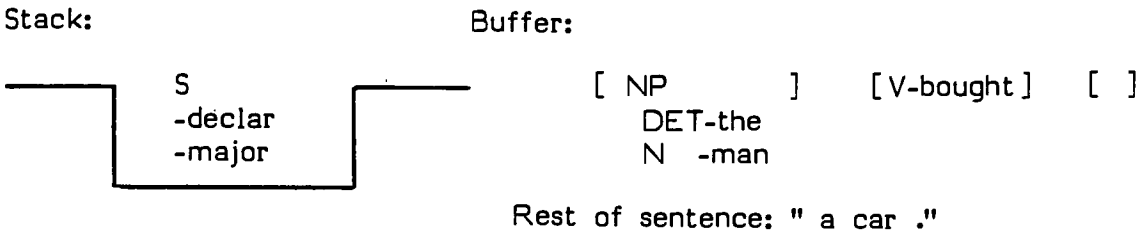
Attach buffer cell 1 to the currently active node as a noun (N).

Now 'pop' the currently active node into buffer cell 1, and move the next word into the buffer, yielding:



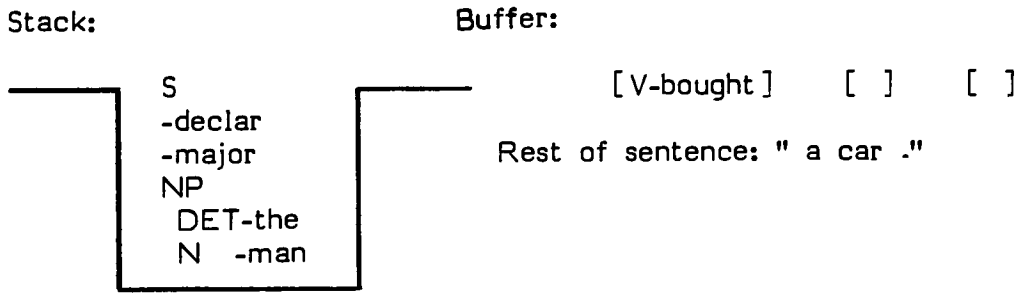
Pattern/action rules: Since S is at the top of the stack (and packets CLAUSE-POOL and SENTENCE-START were never deactivated), examine CLAUSE-POOL and SENTENCE-START for a match with NP in buffer cell 1. One of the SENTENCE-START rule packets seeks a [NP] [V] pair in buffer cells 1 and 2. This rule infers that the sentence must be a 'declarative' sentence, and that the parse is currently working on the 'major clause'.

Actions taken: In accordance with that rule packet, the sentence is labelled 'declarative' and 'major'. Rule packet PARSE-SUBJ is activated. This situation is diagrammed below.



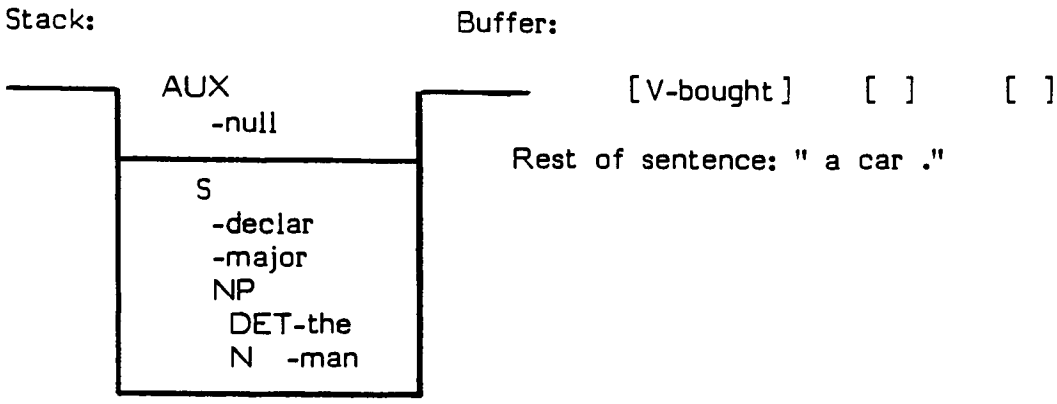
Pattern/action rules: In PARSE-SUBJ, a match is found with [NP] [V], allowing the NP to be accepted as the subject of the sentence.

Actions taken: The contents of buffer cell 1 are attached to the currently active node, acting as the subject of the sentence. PARSE-SUBJ is deactivated and PARSE-AUX is activated, looking for auxiliary verb constituents. This situation results:



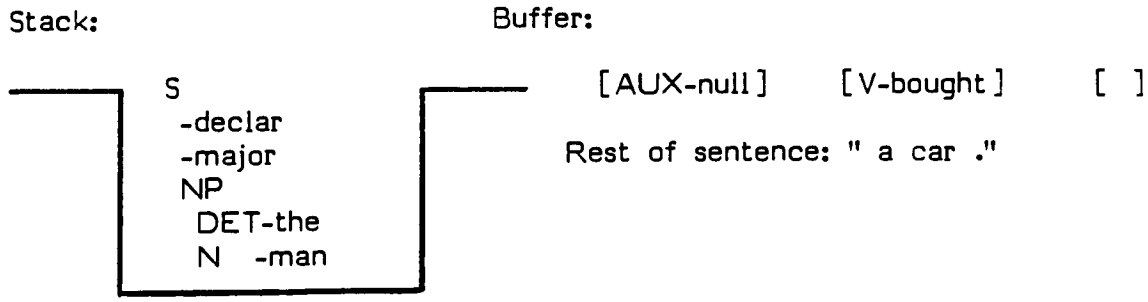
Pattern/action rules: The grammar rules at this point look for a 'to' verb, an 'aux' verb, or a 'main' verb.

Actions taken: Since a main verb is found, an AUX node is created on the stack, with value initialized to 'null'. Rule packets BUILD-AUX and CLAUSE-POOL are activated.



Pattern/action rules: The rules at this point are looking for complex verb structures (using 'do', 'be', and 'have'), and these rules are unsuccessful. A default rule packet is provided for this situation, and it is used here.

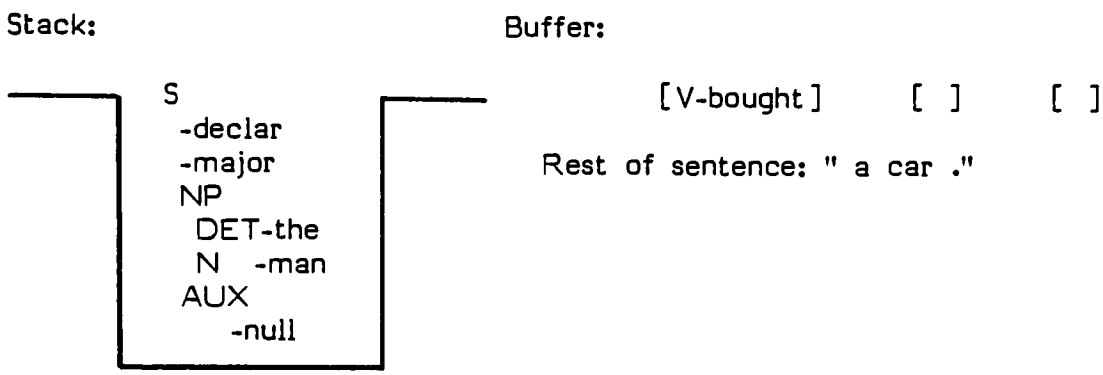
Actions: The currently active node is dropped into buffer cell 1. Since 'S' has migrated back to the top of the stack, its rule packets (CLAUSE-POOL and PARSE-AUX) are reactivated.



Pattern/action rules: PARSE-AUX looks in buffer cell 1 for a 'to' verb, a 'main' verb, or an 'aux' verb. It finds an AUX (not caring that its value is 'null') and proceeds.

Actions taken: Buffer cell 1 is attached to the currently active node as

an AUX. PARSE-AUX is deactivated, and PARSE-VP is activated. (CLAUSE-POOL also remains an active rule packet.)

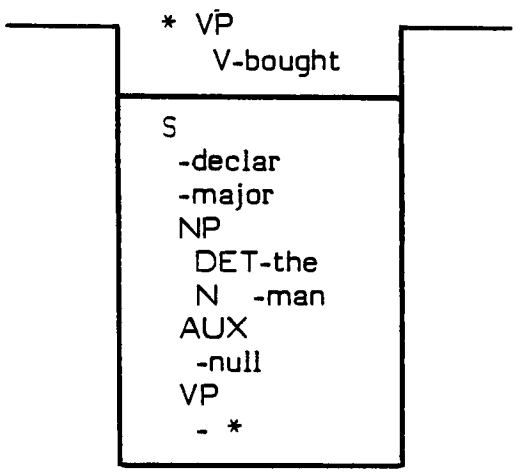


Pattern/action rules: In rule packet PARSE-VP, only a [V] is sought in buffer cell 1. Depending on what features have been encountered (e.g., WH-question, passive sentence, etc.), various actions may be taken.

Actions taken: Since this is a declarative sentence and a major clause, the rule packet SENTENCE-FINAL is activated, and PARSE-VP is deactivated. The clause being parsed is 'major', and this causes rule packet SENTENCE-VP to be activated. CLAUSE-POOL is also declared active at this level.

A new VP node is created on the top of the stack. This new node is linked to the 'S' node through a 'VP' that is created there. The verb in buffer cell 1 is attached to the new VP node as a verb.

Stack:



Buffer:

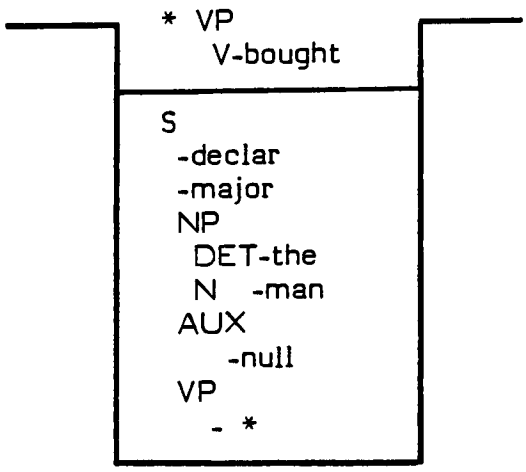
[DET-a] [ ] [ ]

Rest of sentence: "car ."

Pattern/action rules: CLAUSE-POOL looks for a word that can begin a NP (as it did at the beginning of the parse), and it succeeds.

Actions taken: In the next few steps, the NP "a car" is constructed and popped from the stack into the buffer. After the 'pop', rule packets CLAUSE-POOL, SENTENCE-VP, and SENTENCE-FINAL are declared active.

Stack:



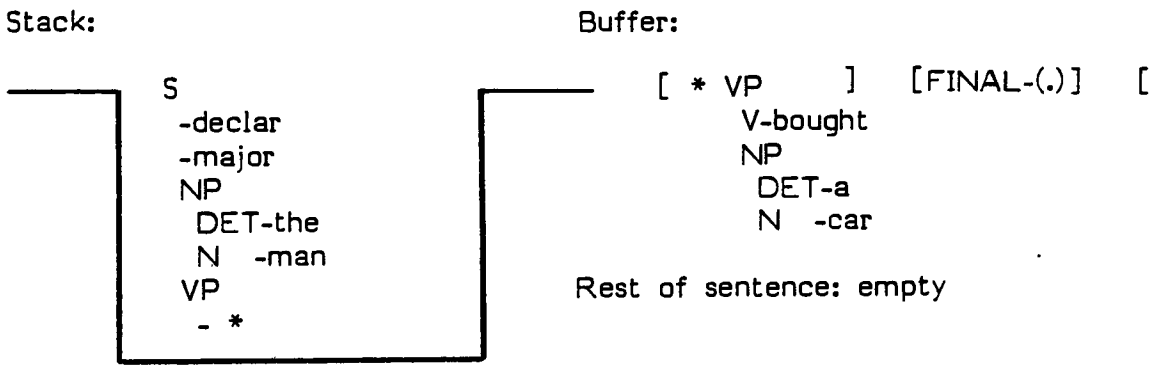
Buffer:

[NP] [ ] [ ]
DET-a
N -car

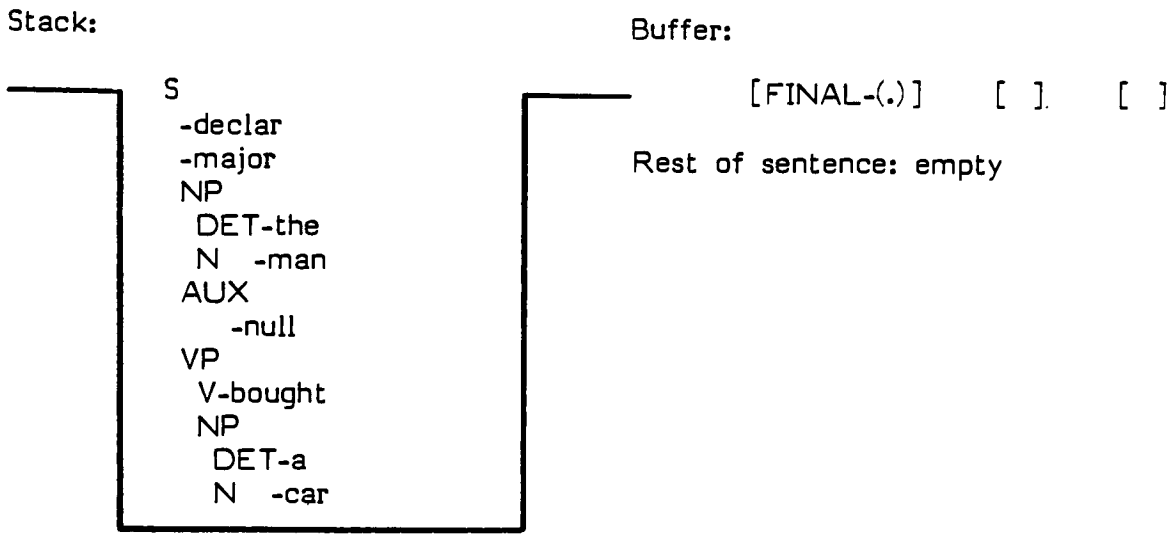
Rest of sentence: " . "

Pattern/action rules: Packet SENTENCE-VP looks for a NP in buffer cell 1, hoping to find a noun phrase that can act as object in the sentence.

Actions taken: Buffer cell 1 is attached to the currently active node as a NP. The period (.) is put into buffer cell 1. The VP node is now complete, so it is popped into the buffer. Note that it is still linked to node 'S'.



Pattern/action rules: Before any of these rules can be applied, the link between the stack and the buffer causes the VP in 'S' to accept the [VP] in the buffer as part of the sentence 'S'. The final view of the stack is shown below:



Pattern/action rules: The packets CLAUSE-POOL, SENTENCE-VP, and SENTENCE-FINAL are all active. SENTENCE-FINAL looks for a period, question mark, or exclamation point in buffer cell 1, and it succeeds.

Actions taken: Buffer cell 1 is attached to the currently active node. The currently active node is popped as a completed sentence.

This parse shows fairly clearly how the top-down and bottom-up components act in PARSIFAL. The stack provides top-down predictions in the form of rule packets, and the buffer cells combine into partial constituents. For example, after the initial NP (the man) is accepted as subject of the sentence, the PARSE-AUX rule packet provides parse continuations only for those verb constructions that could continue a successful parse of the sentence. (Actually, the NP 'the man' is acepted only as a noun phrase at this point. If the sentence had been passive, this NP would have become the object of the sentence, using a mechanism much

like the HOLD register in Woods' ATN.)

The cells in the buffer provide the bottom-up parsing capability. Note that intermediate constituents, consisting of groups of adjacent words, always ended up in the buffer before being accepted as a major constituent by the clause or sentence.

As for Marcus' claim that PARSIFAL parses 'deterministically', what success it has in this area is due to the principle of least commitment (Swartout, 1978). PARSIFAL does not need to backtrack because it keeps all its options open until the buffer contains a completed constituent. To see the principle of least commitment in action, look back at the previous example to the spot where the verb 'bought' was parsed. A verb phrase node was created, and three rule packets were activated-CLAUSE-POOL, SENTENCE-VP, and SENTENCE-FINAL. Parsifal did not have to choose a final sentence level rule packet until more of the sentence was processed. A strict top-down parser, on the other hand, would have have chosen one (most likely) path to traverse at that point. If it had failed on that path, it would have had to backtrack to try another.

The three cell buffer of PARSIFAL will handle many simple sentences. However, Marcus found it necessary to extend the buffer size to five cells to handle some complex noun phrases. The idea of a deterministic parser (as defined by Marcus) seems to be reasonably successful and powerful. PARSIFAL demonstrates both the power and efficiencies obtainable by combining top-down and bottom-up parsing strategies.

CHAPTER 17

Conclusions

Syntactic parsing of natural language has grown in its ability to successfully parse complex sentences. In the course of achieving this, natural language parsing has been applied to theories in linguistics and research in psychology. It has also become more widely available, and it is now not just limited to research systems.

One of the most influential systems developed for syntactic parsing was the augmented transition network parser developed by William Woods. It applied the theories of transformational grammar successfully for the first time in a large scale computer system. It has also been argued that an ATN is a good model of the psychology of human parsing (Wanner, 1980).[1] The argument has also been made that the HOLD mechanism of an ATN corresponds closely to the short term memory that humans use for sentence comprehension (Wanner and Maratsos, 1978; Miller, 1981).

Both of the systemic grammar based systems that were examined (SHRDLU and SLOT) have been held up as possible models of human sentence parsing. The integration of syntactic parsing and semantic consistency checks at the phrase level in SHRDLU can be compared to the syntactic/semantic interaction in humans. The "middle-out" phrase construction used in McCord's SLOT grammar ("find the head word and then

[1] For a rebuttal, see (Fodor and Frazier, 1980).

add modifiers or components to the left and right of it") also holds promise as a model of part of the human parsing mechanism.

Finally, PARSIFAL's linguistic contribution was in the field of trace theory in transformational grammar. Psychologically, the combined top-down and bottom-up parsing algorithm used in PARSIFAL is able to parse those sentences that humans can parse without difficulty. As such, it may have some validity as a model of human syntactic parsing.

Aside from these research concerns, there are signs that the use of natural language by computers is beginning to "come of age" in a commercial setting. Some data bases now allow querying of the contents of the data base in natural language. One example is INTELLECT, a commercially available product that uses Woods' ATN for its syntactic component (Harris, 1980). Using a sales data base, this system can parse (and answer the question!) queries such as "Give me a report of names, second quarter sales, and YTD sales for northeast states other than MA, broken down by state." (AIC, 1982).

It should be obvious that a parser can not handle a sentence like the one above using only syntactic information. There must be some semantic processing that occurs as well. The semantic area is where most of the current work in computers and natural language is taking place. Parsing for syntax is now a reasonably well understood process. Progress in the syntactic area of natural language analysis now tends to be incremental rather than dramatic.

Some current researchers in the field believe that syntax is not a necessary component in successfully "understanding" natural language.

Basing their work on the theories of Roger Schank (Schank, 1972) and others at Yale University, these researchers believe that context and past experience are the important concerns in language comprehension. Gerald DeJong, as a graduate student at Yale, wrote a program that produced abstracts of newspaper stories (with limited success), and he used only seven syntax rules for this "semantic parsing" (DeJong, 1979).

While it is true that the emphasis has shifted from syntax to semantics, this does not mean that parsing for syntax is unimportant. Syntactic parsing tends to be easier to implement (and hence cheaper) and it is also better understood than the current semantic approaches. Syntactic parsing of natural language has proven itself a useful tool in linguistics and psychology, and it is just now beginning to receive wider exposure and greater use in commercial data processing settings.

Bibliography

Aho and Ullman (1972), The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing, Prentice-Hall.

AIC (1982), INTELLECT Query System User's Guide, Artificial Intelligence Corporation, Waltham, Mass.

Akmajian and Heny (1975), An Introduction to the Principles of Transformational Syntax, M.I.T. Press.

Barr and Feigenbaum (1981), The Handbook of Artificial Intelligence : Volume 1, William Kaufman, Inc.

Bates (1978), "The Theory and Practice of Augmented Transition Networks", in Lecture Notes in Computer Science (63) - Natural Language Communication with Computers, Bolc, ed., Springer-Verlag.

Bobrow and Webber (1980), "PSI-KLONE", in Proceedings of the Third Biennial Conference of the Canadian Society for Computational Studies of Intelligence, University of Victoria, British Columbia.

Bratley and Dakin (1968), "A Limited Dictionary for Syntactic Analysis", in Machine Intelligence 3, American Elsevier.

Cattell (1969), The New English Grammar: A Descriptive Introduction, M.I.T. Press.

Charniak, Riesbeck, and McDermott (1980), Artificial Intelligence Programming, Lawrence Erlbaum Associates.

Chomsky (1957), Syntactic Structures, Mouton, The Hague.

Chomsky (1959), "On Certain Formal Properties of Grammars", in Information and Control 2.

Chomsky (1965), Aspects of the Theory of Syntax, M.I.T. Press.

DeJong (1979), Skimmimg Stories in Real Time: An Experiment in Integrated Understanding, Ph. D. thesis, research report no. 158, Yale University Department of Computer Science.

DiBacco and Wilcox (1981), Syntax Parsing Using an Augmented Transition Network, final paper for independent study course ICSS 899 (Fall 1981-82), Rochester Institute of Technology.

Dreyfus (1979), What Computers Can't Do, Harper Colophon Books.

Earley (1968), An Efficient Context-Free Parsing Algorithm, Ph.D. thesis, Carnegie-Mellon University, Computer Science Department.

Earley (1970), "An Efficient Context-Free Parsing Algorithm", in Communications of the ACM, vol. 13, no. 2.

Fillmore (1968), "The Case for Case", in Universals in Linguistic Theory, Bach and Harms, eds., Holt, Rinehart, and Winston.

Fodor and Frazier (1980), "Is the human sentence parsing mechanism an ATN?", in Cognition, vol. 8, Elsevier Sequoia S.A., Lausanne.

Ginsparg (1978), Natural Language Processing in an Automatic Programming Domain, Ph.D. thesis, memo AIM-316, Stanford University, Computer Science Department.

Greibach (1981), "Formal Languages: Origins and Directions", in Annals of the History of Computing, vol. 3, no. 1.

Griffiths and Petrick (1965), "On the Relative Efficiencies of Context-Free Grammar Recognizers", in Communications of the ACM, vol. 8, no. 5.

Halliday (1970), "Language Structure and Language Function", in New Horizons in Linguistics, Lyons, ed., Penguin Books.

Harris (1980), "Using the Data Base as a Semantic Component to Aid in the Parsing of Natural Language Data Base Queries", in Journal of Cybernetics, no. 10.

Hayes and Mouradian (1981), "Flexible Parsing", in American Journal of Computational Linguistics, vol. 7, no. 4.

Hudson (1971), English Complex Sentences: An Introduction to Systemic Grammar, North-Holland.

Hudson (1976), Arguments for a Non-transformational Grammar, University of Chicago Press.

Hunt (1975), Artificial Intelligence, Academic Press.

Kuno and Oettinger (1963), "Multiple Path Syntactic Analyzer", in Information Processing 1962, North Holland.

Kuno (1965), "The Predictive Analyzer and a Path Elimination Technique", in Communications of the ACM, vol. 8, no. 7.

LaPalombara (1976), An Introduction to Grammar: Traditional, Structural, Transformational, Winthrop Publishers, Inc., Cambridge, Mass.

Lyons (1970), "Generative Syntax", in New Horizons in Linguistics, Lyons, ed., Penguin Books.

Marcus (1980), A Theory of Syntactic Recognition for Natural Language, M.I.T. Press.

Martin, Church, and Patil (1981), Preliminary Analysis of a Breadth-First Parsing Algorithm: Theoretical and Experimental Results, technical report MIT/LCS/TR-261, M.I.T. Laboratory for Computer Science.

McCarthy et al, (1962), LISP 1.5 Programmer's Manual, M.I.T. Press.

McCord (1975), "On the form of a systemic grammar", in Journal of Linguistics, England.

McCord (1977), "Procedural Systemic Grammars", in International Journal of Man-Machine Studies, no. 9.

McCord (1980), "Slot Grammars", in American Journal of Computational Linguistics, vol. 6, no. 1.

McCorduck (1979), Machines Who Think, W.H. Freeman and Co.

Miller (1981), Language and Speech, W. H. Freeman and Company.

Postal (1964), "Limitations of Phrase Structure Grammars", in The Structure of Language, Fodor and Katz, eds., Prentice-Hall, Inc.

Pratt (1975), "LINGOL - A Progress Report", in Proceedings of the Fourth International Joint Conference on Artificial Intelligence, M.I.T.

Raphael (1976), The Thinking Computer - Mind Inside Matter, W.H. Freeman and Company.

Ritchie (1977), Computer Modelling of English Grammar, Ph.D. thesis, University of Edinburgh, Department of Computer Science.

Rubin (1973), Grammar for the People: Flowcharts of SHRDLU's Grammar, A.I. memo no. 282, M.I.T. Artificial Intelligence Laboratory.

Sampson (1975), The Form of Language, Weidenfeld and Nicolson, London.

Schank (1972), "Conceptual Dependency: A Theory of Natural Language Understanding", in Cognitive Psychology 3, Academic Press.

Swartout (1978), A Comparison of PARSIFAL with Augmented Transition Networks, A.I. memo 462, M.I.T. Artificial Intelligence Laboratory.

Tennant (1981), Natural Language Processing, Petrocelli Books, Inc.

Thorne et al (1968), "The Syntactic Analysis of English by Machine", in Machine Intelligence 3, American Elsevier.

Thorne et al (1969), "A Program for the Syntactic Analysis of English Sentences", in Communications of the ACM, vol. 12, no. 8.

Wanner and Maratsos (1978), "An ATN Approach to Comprehension", in Linguistic Theory and Psychological Reality, Halle, Bresnan, and Miller, eds., M.I.T. Press.

Wanner (1980), "The ATN and the Sausage Machine: Which One is Baloney?", in Cognition 8, Elsevier Sequoia S.A., Lausanne.

Weizenbaum (1976), Computer Power and Human Reason, W. H. Freeman and Co.

Winograd (1972), Understanding Natural Language, Academic Press.

Winograd (1980), "What Does It Mean to Understand Language", in Cognitive Science 4, Ablex Publishing Corp.

Winograd (1983), Language as a Cognitive Process, Volume 1: Syntax, Addison Wesley.

Winston (1977), Artificial Intelligence, Addison Wesley.

Winston and Horn (1981), LISP, Addison Wesley.

Woods (1969), Augmented Transition Networks for Natural Language Analysis, Harvard Computation Laboratory report CS-1, Harvard University, Department of Computer Science.

Woods (1970), "Transition Network Grammars for Natural Language Analysis", in Communications of the ACM, vol. 13, no. 10.

Woods (1972), The Lunar Sciences Natural Language Information System: Final Report, Bolt, Beranek, and Newman, Inc.