AN EFFICIENT AUGMENTED-CONTEXT-FREE PARSING ALGORITHM[1]

Masaru Tomita

Computer Science Department
and
Center for Machine Translation
Carnegie-Mellon University
Pittsburgh, PA 15213

An efficient parsing algorithm for augmented context-free grammars is introduced, and its application to on-line natural language interfaces discussed. The algorithm is a generalized LR parsing algorithm, which precomputes an LR shift-reduce parsing table (possibly with multiple entries) from a given augmented context-free grammar. Unlike the standard LR parsing algorithm, it can handle arbitrary context-free grammars, including ambiguous grammars, while most of the LR efficiency is preserved by introducing the concept of a "graph-structured stack". The graph-structured stack allows an LR shift-reduce parser to maintain multiple parses without parsing any part of the input twice in the same way. We can also view our parsing algorithm as an extended chart parsing algorithm efficiently guided by LR parsing tables. The algorithm is fast, due to the LR table precomputation. In several experiments with different English grammars and sentences, timings indicate a five- to tenfold speed advantage over Earley's context-free parsing algorithm.

The algorithm parses a sentence strictly from left to right *on-line*, that is, it starts parsing as soon as the user types in the first word of a sentence, without waiting for completion of the sentence. A practical on-line parser based on the algorithm has been implemented in Common Lisp, and running on Symbolics and HP AI workstations. The parser is used in the multi-lingual machine translation project at CMU. Also, a commercial on-line parser for Japanese language is being built by Intelligent Technology Incorporation, based on the technique developed at CMU.

1 INTRODUCTION

Parsing efficiency is crucial when building practical natural language systems on smaller computers such as personal workstations. This is especially the case for interactive systems such as natural language database access, interfaces to expert systems, and interactive machine translation. This paper introduces an efficient on-line parsing algorithm, and focuses on its practical application to natural language interfaces.

The algorithm can be viewed as a generalized LR parsing algorithm that can handle arbitrary context-free grammars, including ambiguous grammars. Section 2 describes the algorithm by extending the standard LR parsing algorithm with the idea of a "graph-structured stack". Section 3 describes how to represent parse trees efficiently, so that all possible parse trees (the parse forest) take at most polynomial space as the ambiguity of a sentence grows exponentially. In section 4, several examples are given. Section 5 presents several empirical results of the algorithm's practical performance, including comparison with Earley's algorithm. In section 6, we discuss how to enhance the algorithm to handle augmented context-free grammars rather than pure context-free grammars. Section 7 describes the concept of on-line parsing, taking advantage of left-to-right operation of our parsing algorithm. The on-line parser parses a sentence strictly from left to right, and starts parsing as soon as the user types in the first word, without waiting for the end of line. Benefits of on-line parsing are then discussed. Finally, several versions of on-line parser have been implemented, and they are mentioned in section 8.

2 THE CONTEXT-FREE PARSING ALGORITHM

The LR parsing algorithms (Aho and Ullman 1972, Aho and Johnson 1974) were developed originally for programming languages. An LR parsing algorithm is a

0362-613X/87/010031-46$03.00

shift-reduce parsing algorithm deterministically guided by a parsing table indicating what action should be taken next. The parsing table can be obtained automatically from a context-free phrase structure grammar, using an algorithm first developed by DeRemer (1969, 1971). We do not describe the algorithms here, referring the reader to chapter 6 in Aho and Ullman (1977). We assume that the reader is familiar with the standard LR parsing algorithm (not necessarily with the parsing table construction algorithm).

The LR paring algorithm is one of the most efficient parsing algorithms. It is totally deterministic, and no backtracking or search is involved. Unfortunately, we cannot directly adopt the LR parsing technique for natural languages, because it is applicable only to a small subset of context-free grammars called LR grammars, and it is almost certain that any practical natural language grammars are not LR. If a grammar is non-LR, its parsing table will have multiple entries;[1] one or more of the action table entries will be multiply defined (Shieber 1983). Figures 2.1 and 2.2 show an example of a non-LR grammar and its parsing table. Grammar symbols starting with "*" represent pre-terminals. Entries "sh *n*" in the action table (the left part of the table) indicate the action "shift one word from input buffer onto the stack, and go to state *n*". Entries "re *n*" indicate the action "reduce constituents on the stack using rule *n*". The entry "acc" stands for the action "accept", and blank spaces represent "error". The goto table (the right part of the table) decides to what state the parser should go after a reduce action. These operations shall become clear when we trace the algorithm with example sentences in section 4. The exact definition and operation of the LR parser can be found in Aho and Ullman (1977).

We can see that there are two multiple entries in the action table; on the rows of state 11 and 12 at the column labeled "*prep". Roughly speaking, this is the situation where the parser encounters a preposition of a PP right after a NP. If this PP does not modify the NP, then the parser can go ahead to reduce the NP into a higher nonterminal such as PP or VP, using rule 6 or 7, respectively (re6 and re7 in the multiple entries). If, on the other hand, the PP does modify the NP, then the parser must wait (sh6) until the PP is completed so it can build a higher NP using rule 5.

It has been thought that, for LR parsing, multiple entries are fatal because once a parsing table has multiple entries, deterministic parsing is no longer possible and some kind of non-determinism is necessary. We handle multiple entries with a special technique, named a **graph-structured stack.** In order to introduce the concept, we first give a simpler form of non-determinism, and make refinements on it. Subsection 2.1 describes a simple and straightforward non-deterministic technique, that is, pseudo-parallelism (breadth-first search), in which the system maintains a number of stacks simultaneously, called the **Stack List.** A disadvantage of the stack list is then described. The next subsection describes the idea of stack combination, which was introduced in the author's earlier research (Tomita 1984), to make the algorithm much more efficient. With this idea, stacks are represented as trees (or a forest). Finally, a further refinement, the graph-structured stack, is described to make the algo-

```
-------------------------
(1)   S --> NP VP
(2)   S --> S PP
(3)  NP --> *n
(4)  NP --> *det *n
(5)  NP --> NP PP
(6)  PP --> *prep NP
(7)  VP --> *v NP
-------------------------
```

Figure 2.1. An example ambiguous grammar.

State	*det	*n	*v	*prep	$	NP	PP	VP	S
0	sh3	sh4				2			1
1				sh6	acc		5		
2			sh7	sh6			9	8	
3		sh10							
4			re3	re3	re3				
5				re2	re2				
6	sh3	sh4				11			
7	sh3	sh4				12			
8				re1	re1				
9			re5	re5	re5				
10			re4	re4	re4				
11			re6	re6,sh6	re6		9		
12				re7,sh6	re7		9		

Figure 2.2. LR parsing table with multiple entries.

rithm even more efficient; efficient enough to run in polynomial time.

2.1 Handling Multiple Entries with Stack List

The simplest idea would be to handle multiple entries non-deterministically. We adopt pseudo-parallelism (breadth-first search), maintaining a list of stacks (the Stack List). The pseudo-parallelism works as follows.

A number of *processes* are operated in parallel. Each process has a stack and behaves basically the same as in standard LR parsing. When a process encounters a multiple entry, the process is split into several processes (one for each entry), by replicating its stack. When a process encounters an error entry, the process is killed, by removing its stack from the stack list. All processes are synchronized; they shift a word at the same time so that they always look at the same word. Thus, if a process encounters a shift action, it waits until all other processes also encounter a (possibly different) shift action.

Figure 2.3 shows a snapshot of the stack list right after shifting the word *with* in the sentence *I saw a man on the bed in the apartment with a telescope* using the grammar in Figure 2.1 and the parsing table in Figure 2.2. For the sake of convenience, we denote a stack with vertices and edges. The leftmost vertex is the bottom of the stack, and the rightmost vertex is the top of the stack. Vertices represented by a circle are called **state vertices,** and they represent a state number. Vertices represented by a square are called **symbol vertices,** and they represent a grammar symbol. Each stack is exactly the same as a stack in the standard LR parsing algorithm. The distance between vertices (length of an edge) does not have any significance, except it may help the reader understand the status of the stacks. In the figures, "*p" stands for *prep, and "*d" stands for *det throughout this paper.

Since the sentence is 14-way ambiguous, the stack has been split into 14 stacks. For example, the sixth stack

(0 S 1 *p 6 NP 11 *p 6)

is in the status where *I saw a man on the bed* has been reduced into S, and *the apartment* has been reduced into NP. From the LR parsing table, we know that the top of the stack, *state 6,* is expecting *det or *n and eventually a NP. Thus, after *a telescope* comes in, a PP *with a telescope* will be formed, and the PP will modify the NP *the apartment,* and *in the apartment* will modify the S *I saw a man.*

We notice that some stacks in the stack list appear to be identical. This is because they have reached the current state in different ways. For example, the sixth and seventh stacks are identical, because *I saw a man on the bed* has been reduced into S in two different ways.

A disadvantage of the stack list method is that there are no interconnections between stacks (processes), and there is no way in which a process can utilize what other processes have done already. The number of stacks in the stack list grows exponentially as ambiguities are encountered.[3] For example, these 14 processes in Figure 2.3 will parse the rest of the sentence *the telescope* 14

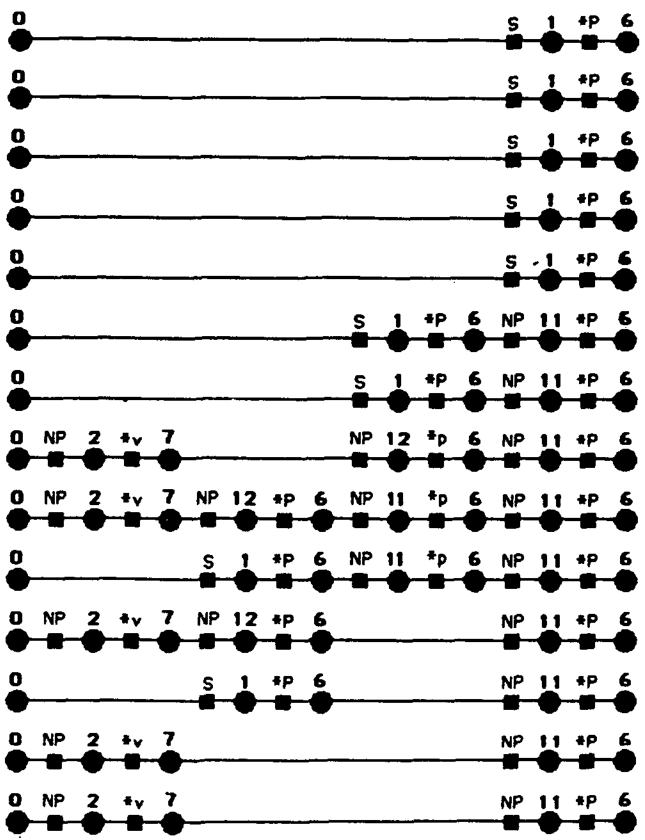


Figure 2.3. Stack list: after shifting *with* in
I saw a man on the bed in the apartment with a telescope
(with the the grammar and the table in Figures 2.1 and 2.2).

times in exactly the same way. This can be avoided by using a tree-structured stack, which is described in the following subsection.

2.2 With a Tree-Structure Stack

If two processes are in a common state, that is, if two stacks have a common state number at the rightmost vertex, they will behave in exactly the same manner until the vertex is popped from the stacks by a reduce action. To avoid this redundant operation, these processes are unified into one process by combining their stacks. Whenever two or more processes have a common state number on the top of their stacks, the top vertices are unified, and these stacks are represented as a tree, where the top vertex corresponds to the root of the tree. We call this a tree-structured stack. When the top vertex is popped, the tree-structured stack is split into the original number of stacks. In general, the system maintains a number of tree-structured stacks in parallel, so stacks are represented as a forest. Figure 2.4 shows a snapshot of the tree-structured stack immediately after shifting the word *with.* In contrast to the previous example, *the telescope* will be parsed only once.

Although the amount of computation is significantly reduced by the stack combination technique, the number of branches of the tree-structured stack (the number of

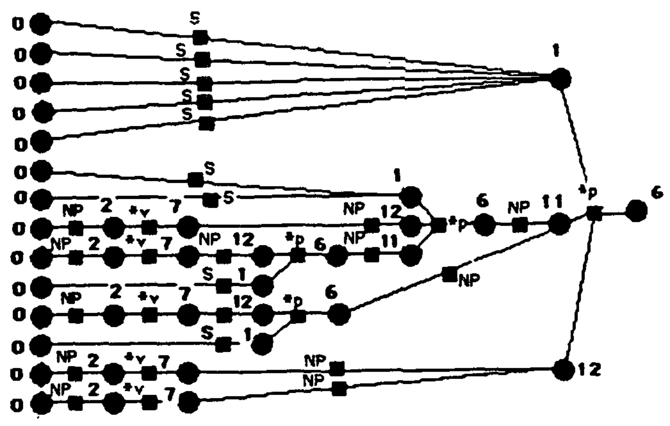


Figure 2.4. A tree-structured stack.

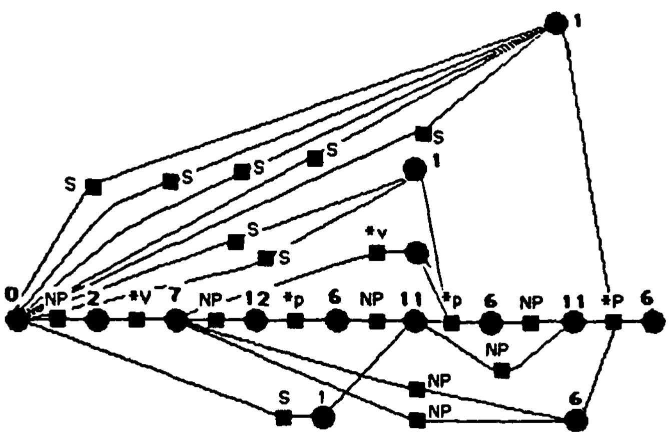


Figure 2.5. A graph-structured stack.

bottoms of the stack) that must be maintained still grows exponentially as ambiguities are encountered. In the next subsection, we describe a further modification in which stacks are represented as a directed acyclic graph, in order to avoid such inefficiency.

2.3 WITH A GRAPH-STRUCTURE STACK

So far, when a stack is split, a copy of the whole stack is made. However, we do not necessarily have to copy the whole stack: even after different parallel operations on the tree-structured stack, the bottom portion of the stack may remain the same. Only the necessary portion of the stack should therefore be split. When a stack is split, the stack is thus represented as a tree, where the bottom of the stack corresponds to the root of the tree. With the stack combination technique described in the previous subsection, stacks are represented as a directed acyclic graph. Figure 2.5 shows a snapshot of the graph stack. It is easy to show that the algorithm with the graph-structured stack does not parse any part of an input sentence more than once in the same way. This is because, if two processes had parsed a part of a sentence in the same way, they would have been in the same state, and they would have been combined as one process.

The graph-structured stack looks very similar to a chart in chart parsing. In fact, one can also view our algorithm as an extended chart parsing algorithm that is guided by LR parsing tables. The major extension is that nodes in the chart contain more information (LR state numbers) than in conventional chart parsing. In this paper, however, we describe the algorithm as a generalized LR parsing algorithm only.

So far, we have focussed on how to accept or reject a sentence. In practice, however, the parser must not only accept or reject sentences but also build the syntactic structure(s) of the sentence (parse forest). The next section describes how to represent the parse forest and how to build it with our parsing algorithm.

3 An Efficient Representation of a Parse Forest

Our parsing algorithm is an **all-path parsing** algorithm; that is, it produces all possible parses in case an input sentence is ambiguous. Such all-path parsing is often needed in natural language processing to manage temporarily or absolutely ambiguous input sentences. The ambiguity (the number of parses) of a sentence may grow exponentially as the length of a sentence grows (Church and Patil 1982). Thus, one might notice that, even with an efficient parsing algorithm such as the one we described, the parser would take exponential time because exponential time would be required merely to print out all parse trees (parse forest). We must therefore provide an efficient representation so that the size of the parse forest does not grow exponentially.

This section describes two techniques for providing an efficient representation: subtree sharing and local ambiguity packing. It should be mentioned that these two techniques are not completely new ideas, and some existing systems (e.g., Earley's (1970) algorithm) have already adopted these techniques, either implicitly or explicitly.

3.1 SUB-TREE SHARING

If two or more trees have a common subtree, the subtree should be represented only once. For example, the parse forest for the sentence *I saw a man in the park with a telescope* should be represented as in Figure 3.1.

To implement this, we no longer push grammatical symbols on the stack; instead, we push pointers to a node of the shared forest.[4] When the parser "shifts" a word, it creates a leaf node labeled with the word and the pre-terminal, and, instead of the pre-terminal symbol, a pointer to the newly created leaf node is pushed onto the stack. If the exact same leaf node (i.e., the node labeled with the same word and the same pre-terminal) already exists, a pointer to this existing node is pushed onto the stack, without creating another node. When the parser "reduces" the stack, it pops pointers from the stack,

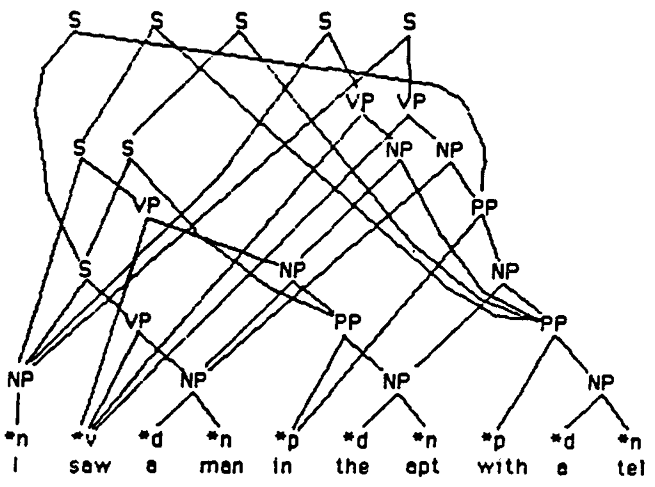

S S S S S
VP VP
S S NP NP
VP PP
S NP NP
VP PP PP
NP NP NP NP
*n *v *d *n *p *d *n *p *d *n
I saw a man in the apt with a tel

Figure 3.1. Unpacked shared forest.

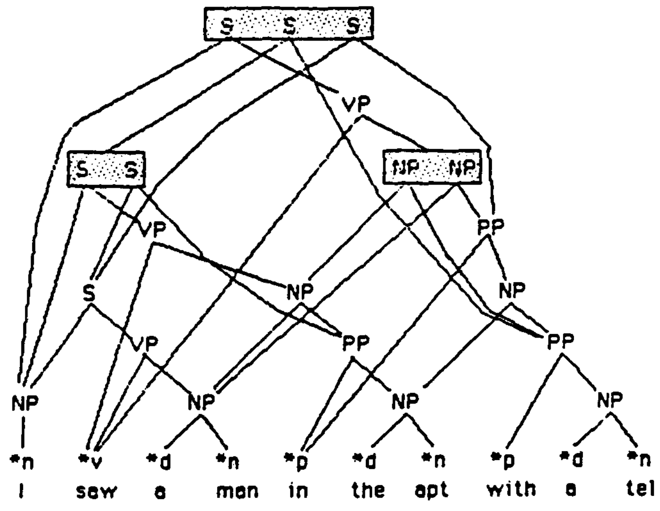

S S S
VP
S S NP NP
VP PP
S NP NP
VP PP PP
NP NP NP NP
*n *v *d *n *p *d *n *p *d *n
I saw a man in the apt with a tel

Figure 3.2. Packed shared forest.

creates a new node whose successive nodes are pointed to by those popped pointers, and pushes a pointer to the newly created node onto the stack.

Using this relatively simple procedure, our parsing algorithm can produce the shared forest as its output without any other special book-keeping mechanism, because it never does the same reduce action twice in the same manner.

3.2 LOCAL AMBIGUITY PACKING

We say that two or more subtrees represent **local ambiguity** if they have common leaf nodes and their top nodes are labeled with the same non-terminal symbol. That is to say, a fragment of a sentence is locally ambiguous if the fragment can be reduced to a certain non-terminal symbol in two or more ways. If a sentence has many local ambiguities, the total ambiguity would grow exponentially. To avoid this, we use a technique called **local ambiguity packing,** which works in the following way. The top nodes of subtrees that represent local ambiguity are merged and treated by higher-level structures as if there were only one node. Such a node is called a **packed node,** and nodes before packing are called **subnodes** of the packed node. An example of a shared-packed forest is shown in Figure 3.2. Packed nodes are represented by boxes. We have three packed nodes in Figure 3.2; one with three subnodes and two with two subnodes.

Local ambiguity packing can be easily implemented with our parsing algorithm as follows. In the graph-structured stack, if two or more symbol vertices have a common state vertex immediately on their left and a common state vertex immediately on their right, they represent local ambiguity. Nodes pointed to by these symbol vertices are to be packed as one node. In Figure 2.5, for example, we see one 5-way local ambiguity and two 2-way local ambiguities. The algorithm is made clear by the example in the following section.

Recently, the author (Tomita 1986) suggested a technique to disambiguate a sentence out of the shared-packed forest representation by asking the user a minimal number of questions in natural language (without showing any tree structures).

4 Examples

This section presents three examples. The first example, using the sentence *I saw a man in the apartment with a telescope,* is intended to help the reader understand the algorithm more clearly.

The second example, with the sentence *That information is important is doubtful,* is presented to demonstrate that our algorithm is able to handle multi-part-of-speech words without any special mechanism. In the sentence, *that* is a multi-part-of-speech word, because it could also be a determiner or a pronoun.

The third example is provided to show that the algorithm is also able to handle unknown words by considering an unknown word as a special multi-part-of-speech word whose part of speech can be anything. We use an example sentence *I * a *,* where *s represent unknown words.

4.1 The Example

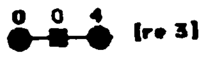
This subsection gives a trace of the algorithm with the grammar in Figure 2.1, the parsing table in Figure 2.2, and the sentence *I saw a man in the park with a telescope.*

At the very beginning, the stack contains only one vertex labeled 0, and the parse forest contains nothing. By looking at the action table, the next action, "shift 4", is determined as in standard LR parsing.

Next Word = 'I'
0
[sh 4]

When shifting the word *I*, the algorithm creates a leaf node in the parse forest labeled with the word *I* and its preterminal **n*, and pushes a pointer to the leaf node onto the stack. The next action, "reduce 3, is determined from the action table.

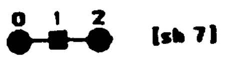
Next Word = 'saw'



```
0 [*n 'I']
```

We reduce the stack basically in the same manner as standard LR parsing. It pops the top vertex "4" and the pointer "0" from the stack, and creates a new node in the parse forest whose successor is the node pointed to by the pointer. The newly created node is labeled with the left-hand side symbol of rule 3, namely "NP". The pointer to this newly created node, namely "1", is pushed onto the stack. The action, "shift 7", is determined as the next action. Now, we have

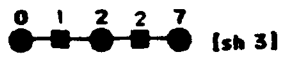
Next Word = 'saw'



```
0 [*n 'I']
1 [NP (0)]
```

After executing "shift 7", we have

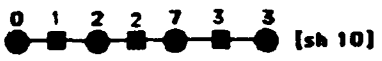
Next Word = 'a'



```
0 [*n 'I']
1 [NP (0)]
2 [*v 'saw']
```

After executing "shift 3", we have

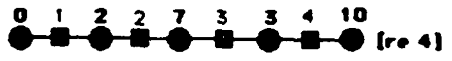
Next Word = 'man'



```
0 [*n 'I']
1 [NP (0)]
2 [*v 'saw']
3 [*det 'a']
```

After executing "shift 10", we have

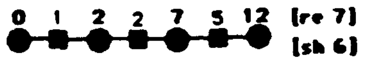
Next Word = 'in'



```
0 [*n 'I']
1 [NP (0)]
2 [*v 'saw']
3 [*det 'a']
4 [*n 'man']
```

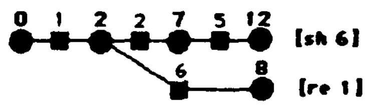
The next action is "reduce 4". It pops pointers, "3" and "4", and creates a new node in the parse forest such that node 3 and node 4 are its successors. The newly created node is labeled with the left-hand side symbol of rule 4, i.e., "NP". The pointer to this newly created node, "5", is pushed onto the stack. We now have

Next Word = 'in'



```
0 [*n 'I']
1 [NP (0)]
2 [*v 'saw']
3 [*det 'a']
4 [*n 'man']
5 [NP (3 4)]
```

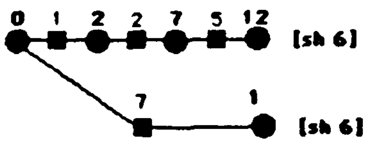
At this point, we encounter a multiple entry, "reduce 7" and "shift 6", and both actions are to be executed. Reduce actions are always executed first, and shift actions are executed only when there is no remaining reduce action to execute. In this way, the parser works strictly from left to right; it does everything that can be done before shifting the next word. After executing "reduce 7", the stack and the parse forest look like the following. The top vertex labeled "12" is not popped away, because it still has an action not yet executed. Such a top vertex, or more generally, vertices with one or more actions yet to be executed, are called "active". Thus, we have two active vertices in the stack above: one labeled "12", and the other labeled "8". The action "reduce 1" is determined from the action table, and is associated with the latter vertex.



Next Word = 'in'

```
0 [*n 'I']
1 [NP (0)]
2 [*v 'saw']
3 [*det 'a']
4 [*n 'man']
5 [NP (3 4)]
6 [VP (2 5)]
```

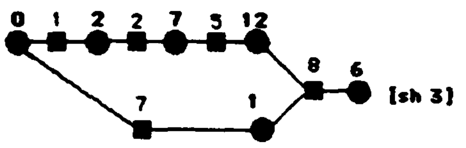
Because reduce actions have a higher priority than shift actions, the algorithm next executes "reduce 1" on the vertex labeled "8". The action "shift 6" is determined from the action table.



Next Word = 'in'

```
0 [*n 'I']
1 [NP (0)]
2 [*v 'saw']
3 [*det 'a']
4 [*n 'man']
5 [NP (3 4)]
6 [VP (2 5)]
7 [S (1 6)]
```

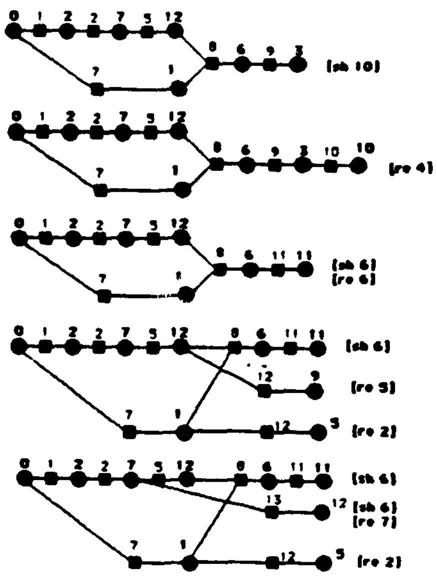
Now we have two "shift 6"'s. The parser, however, creates only one new leaf node in the parse forest. After executing two shift actions, it combines vertices in the stack wherever possible. The stack and the parse forest look like the following, and "shift 3" is determined from the action table as the next action.

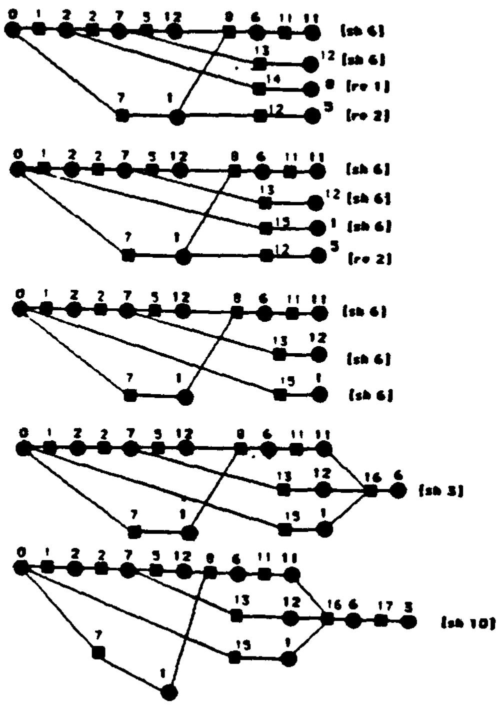


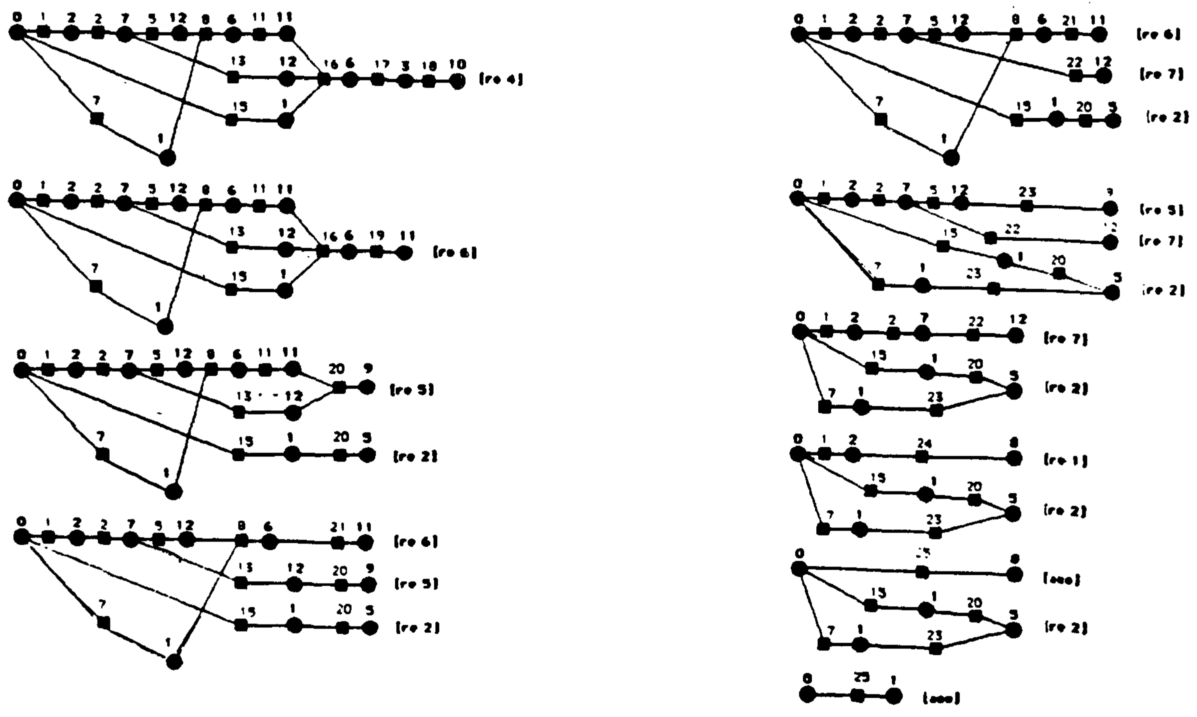
Next Word = 'the'

```
0 [*n 'I']
1 [NP (0)]
2 [*v 'saw']
3 [*det 'a']
4 [*n 'man']
5 [NP (3 4)]
6 [VP (2 5)]
7 [S (1 6)]
8 [*prep 'in']
```

After about 20 steps (see below), the action "accept" is finally executed. It returns "25" as the top node of the parse forest, and halts the process.







The final parse forest is

```
0 [*n 'I']       10 [*n 'park']          20 [PP (16 19)]
1 [NP (0)]       11 [NP (9 10)]          21 [NP (11 20)]
2 [*v 'saw']     12 [PP (8 11)]          22 [NP (13 20)]
3 [*det 'a']     13 [NP (5 12)]          23 [PP (8 21)]
4 [*n 'man']     14 [VP (2 13)]          24 [VP (2 22)]
5 [NP (3 4)]     15 [S (1 14) (7 12)]    25 [S (1 24) (15 22) (7 23)]
6 [VP (2 5)]     16 [*prep 'with']
7 [S (1 6)]      17 [*det 'a']
8 [*prep 'in']   18 [*n 'scope']
9 [*det 'the']   19 [NP (17 18)]
```

4.2 Managing Multi-Part-of-Speech Words

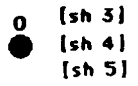
This subsection gives a trace of the algorithm with the sentence *That information is important is doubtful,* to demonstrate that our algorithm can handle multi-part-of-speech words (in this sentence, *that*) just like multiple entries without any special mechanism. We use the grammar at the right and the parsing table below.

```
(1)   S  --> NP VP
(2)   NP --> *det *n
(3)   NP --> *n
(4)   NP --> *that S
(5)   VP --> *be *adj
```

State	*adj	*be	*det	*n	*that	$	NP	S	VP
0			sh5	sh4	sh3		2	1	
1						acc			
2		sh6							7
3			sh5	sh4	sh3		2	8	
4		re3							
5				sh9					
6	sh10								
7		re1				re1			
8		re4							
9		re2							
10		re5				re5			

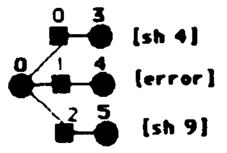
At the very beginning, the parse forest contains nothing, and the stack contains only one vertex, labeled 0. The first word of the sentence is *that,* which can be categorized as *that, *det or *n. The action table tells us that all of these categories are legal. Thus, the algorithm behaves as if a multiple entry is encountered. Three actions, "shift 3", "shift 4", and "shift 5", are to be executed.

Next Word = 'that'



After executing those three shift actions, we have

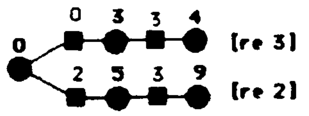
Next Word = 'information'



```
0 [*that 'that']
1 [*n 'that']
2 [*det 'that']
```

Note that three different leaf nodes have been created in the parse forest. One of the three possibilities, *that* as a noun, is discarded immediately after the parser sees the next word *information.* After executing the two shift actions, we have

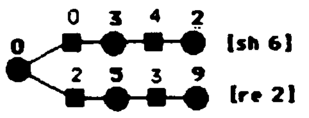
Next Word = 'is'



```
0 [*that 'that']
1 [*n 'that']
2 [*det 'that']
3 [*n 'information']
```

This time, only one leaf node has been created in the parse forest, because both shift actions regarded the word as belonging to the same category, i.e., noun. Now we have two active vertices, and "reduce 3" is arbitrarily chosen as the next action to execute. After executing "reduce 3", we have

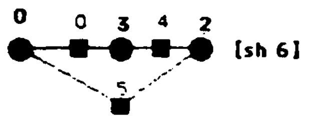
Next Word = 'is'



```
0 [*that 'that']
1 [*n 'that']
2 [*det 'that']
3 [*n 'information']
4 [NP (3)]
```

After executing "reduce 2", we have

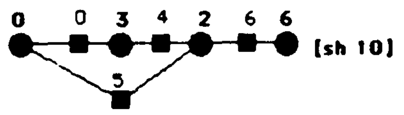
Next Word = 'is'



```
0 [*that 'that']
1 [*n 'that']
2 [*det 'that']
3 [*n 'information']
4 [NP (3)]
5 [NP (2 3)]
```

After executing "shift 6", we have

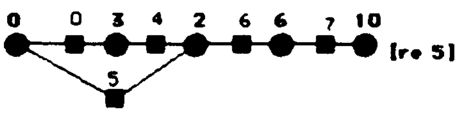
Next Word = 'important'



```
0 [*that 'that']
1 [*n 'that']
2 [*det 'that']
3 [*n 'information']
4 [NP (3)]
5 [NP (2 3)]
6 [*be 'is']
```

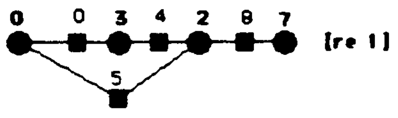
After executing "shift 10", we have

Next Word = 'is'



```
0 [*that 'that']
1 [*n 'that']
2 [*det 'that']
3 [*n 'information']
4 [NP (3)]
5 [NP (2 3)]
6 [*be 'is']
7 [*adj 'important']
```

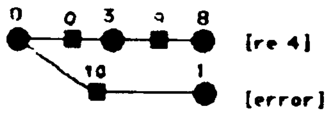
After executing "reduce 5", we have



```
0 [*that 'that']
1 [*n 'that']
2 [*det 'that']
3 [*n 'information']
4 [NP (3)]
5 [NP (2 3)]
6 [*be 'is']
7 [*adj 'important']
8 [VP (6 7)]
```

Next Word = 'is'

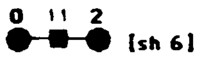
Now, there are two ways to execute the action "reduce 1". After executing "reduce 1" in both ways, we have



```
0 [*that 'that']
1 [*n 'that']
2 [*det 'that']
3 [*n 'information']
4 [NP (3)]
5 [NP (2 3)]
6 [*be 'is']
7 [*adj 'important']
8 [VP (6 7)]
9 [S (4 8)]
```

Next Word = 'is'

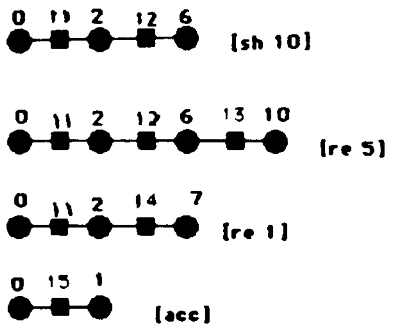
An error action is finally found for the possibility, *that* as a determiner. After executing "reduce 4", we have



```
0 [*that 'that']          8 [VP (6 7)]
1 [*n 'that']             9 [S (4 8)]
2 [*det 'that']          10 [S (5 8)]
3 [*n 'information'] 11 [NP (0 9)]
4 [NP (3)]
5 [NP (2 3)]
6 [*be 'is']
7 [*adj 'important']
```

Next Word = 'is'

After executing "shift 6", and several steps later, we have



```
 0 [*that 'that']
 1 [*n 'that']
 2 [*det 'that']
 3 [*n 'information']
 4 [NP (3)]
 5 [NP (2 3)]
 6 [*be 'is']
 7 [*adj 'important']

 8 [VP (6 7)]
 9 [S (4 8)]
10 [S (5 8)]
11 [NP (0 9)]
12 [*be 'is']
13 [*adj 'doubtful']
14 [VP (12 13)]
15 [S (11 14)]
```

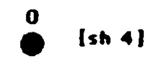
Next Word = '$'

The parser accepts the sentence, and returns "15" as the top node of the parse forest. The forest consists of only one tree which is the desired structure for *That information is important is doubtful.*

4.3 Managing Unknown Words

In the previous subsection, we saw the parsing algorithm handling a multi-part-of-speech word just like multiple entries without any special mechanism. That capability can also be applied to handle unknown words (words whose categories are unknown). An unknown word can be thought of as a special type of a multi-part-of-speech word whose categories can be anything. In the following, we present another trace of the parser with the sentence *I * a **, where *s represent an unknown word. We use the same grammar and parsing table as in the first example (Figures 2.1 and 2.2).

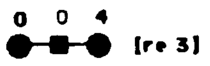
At the very beginning, we have



Next Word = 'I'

After executing "shift 4", we have

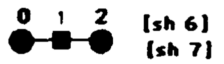
Next Word = '*'



```
0 [*n 'I']
```

At this point, the parser is looking at the unknown word, "*"; in other words, a word whose categories are *det, *n, *v and *prep. On row 4 of the action table, we have only one kind of action, "reduce 3". Thus the algorithm executes only the action "reduce 3", after which we have

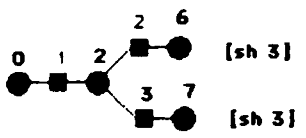
Next Word = '*'



```
0 [*n 'I']
1 [NP (0)]
```

On row 2 of the action table, there are two kinds of actions, "shift 6" and "shift 7". This means the unknown word has two possibilities, as a preposition or a verb. After executing both actions, we have

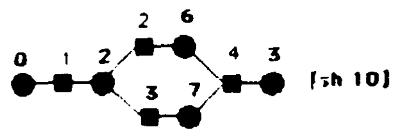
Next Word = 'a'



```
0 [*n 'I']
1 [NP (0)]
2 [*prep '*']
3 [*v '*']
```

After executing "shift 3" twice, we have

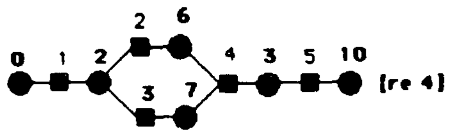
Next Word = '*'



```
0 [*n 'I']
1 [NP (0)]
2 [*prep '*']
3 [*v '*']
4 [*det 'a']
```

At this point, the parser is again looking at the unknown word, "*". However, since there is only one entry on row 3 in the action table, we can uniquely determine the category of the unknown word, which is a noun. After shifting the unknown word as a noun, we have

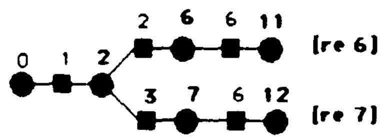
Next Word = '$'



```
0 [*n 'I']
1 [NP (0)]
2 [*prep '*']
3 [*v '*']
4 [*det 'a']
5 [*n '*']
```

After executing "reduce 4", we have

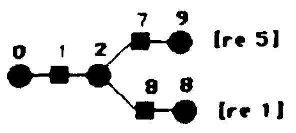
Next Word = '$'



```
0 [*n 'I']
1 [NP (0)]
2 [*prep '*']
3 [*v '*']
4 [*det 'a']
5 [*n '*']
6 [NP (4 5)]
```

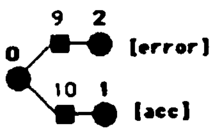
After executing both "reduce 6" and "reduce 7", we have

Next Word = '$'



```
0 [*n 'I']
1 [NP (0)]
2 [*prep '*']
3 [*v '*']
4 [*det 'a']
5 [*n '*']
6 [NP (4 5)]
7 [PP (2 6)]
8 [VP (3 6)]
```

After executing both "reduce 5" and "reduce 1", we have



```
 0 [*n 'I']
 1 [NP (0)]
 2 [*prep '*']
 3 [*v '*']
 4 [*det 'a']
 5 [*n '*']
 6 [NP (4 5)]
 7 [PP (2 6)]
 8 [VP (3 6)]
 9 [NP (1 7)]
10 [S (1 8)]
```

Next Word = '$'

The possibility of the first unknown word being a preposition has now disappeared. The parser accepts the sentence in only one way, and returns "10" as the root node of the parse forest.

We have shown that our parsing algorithm can handle unknown words without any special mechanism.

5 Empirical Results

In this section, we present some empirical results of the algorithm's practical performance. Since space is limited, we only show the highlights of the results, referring the reader to chapter 6 of Tomita (1985) for more detail. Figure 5.1 shows the relationship between parsing time of the Tomita algorithm and the length of input sentence, and Figure 5.2 shows the comparison with Earley's algorithm (or active chart parsing), using a sample English grammar that consists of 220 context-free rules and 40 sample sentençes taken from actual publications. All programs are run on DEC-20 and written in MacLisp, but *not* compiled. Although the experiment is informal, the result show that the Tomita algorithm is about 5 to 10 times faster than Earley's algorithm, due to the pre-compilation of the grammar into the LR table. The

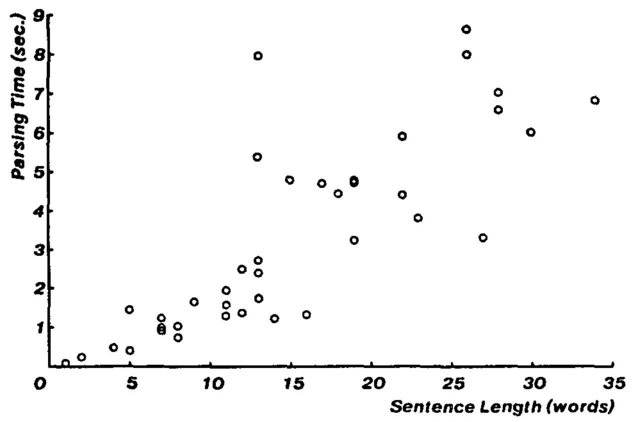


Figure 5.1. Parsing time and sentence length.

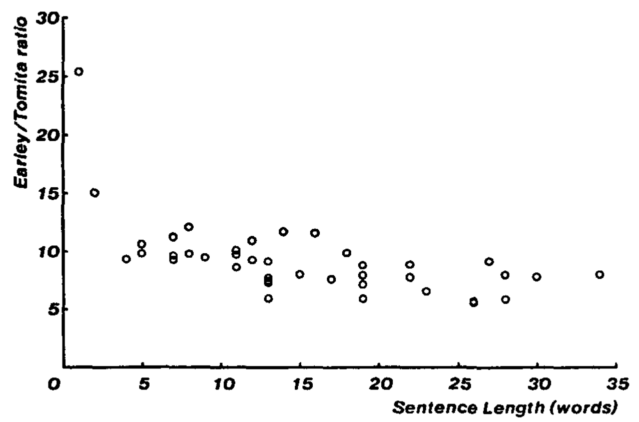


Figure 5.2. Earley/Tomita ratio.

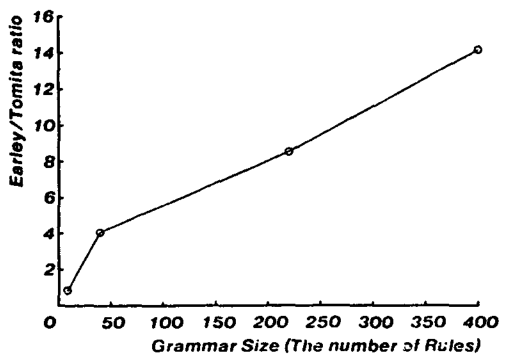


Figure 5.3. Earley/Tomita ratio and grammar size.

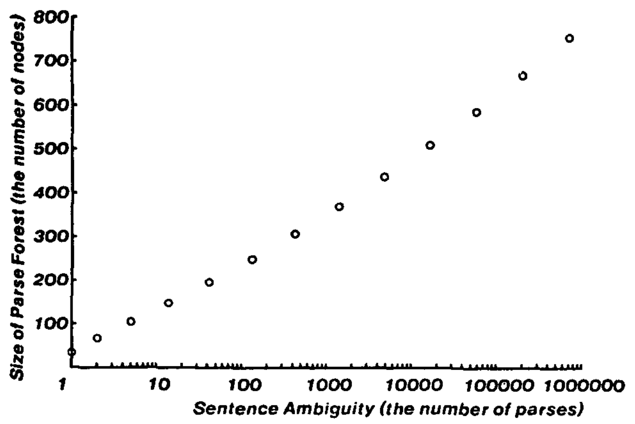


Figure 5.4. Size of parse forest and ambiguity.

Earley/Tomita ratio seems to increase as the size of grammar grows as shown in Figure 5.3. Figure 5.4 shows the relationship between the size of a produced shared-packed forest representation (in terms of the number of nodes) and the ambiguity of its input sentence (the number of possible parses). The sample sentences are created from the following schema.

noun verb det noun (prep det noun)n-1

An example sentence with this structure is

I saw a man in the park on the hill with a telescope ....

The result shows that all possible parses can be represented in almost O(log n) space, where n is the number of possible parses in a sentence.[5]

Figure 5.5 shows the relationship between the parsing time and the ambiguity of a sentence. Recall that within the given time the algorithm produces all possible parses in the shared-packed forest representation. It is concluded that our algorithm can parse (and produce a forest for) a very ambiguous sentence with a million possible parses in a reasonable time.

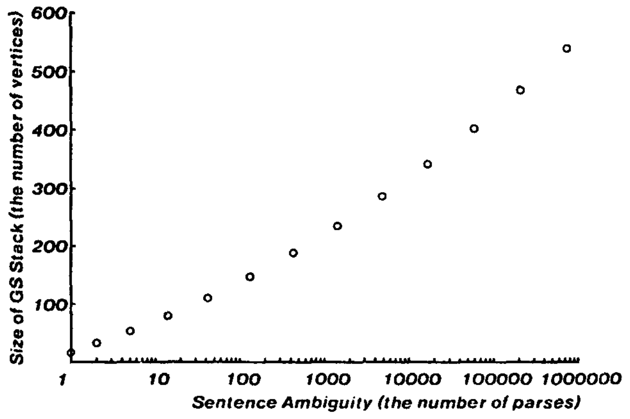


Figure 5.5. Parsing time and ambiguity.

6 Augmented Context-Free Grammars

So far, we have described the algorithm as a pure context-free parsing algorithm. In practice, it is often desired for each grammar nonterminal to have **attributes,** and for each grammar rule to have an augmentation to define, pass, and test the attribute values. It is also desired to produce a functional structure (in the sense of functional grammar formalism (Kay 1984, Bresnan and Kaplan 1982) rather than the context-free forest. Subsection 6.1 describes the augmentation, and subsection 6.2 discusses the shared-packed representation for functional structures.

6.1 The Augmentation

We attach a Lisp function to each grammar rule for this augmentation. Whenever the parser reduces constituents into a higher-level nonterminal using a phrase structure rule, the Lisp program associated with the rule is evaluated. The Lisp program handles such aspects as construction of a syntax/semantic representation of the input sentence, passing attribute values among constituents at different levels and checking syntactic/semantic constraints such as subject-verb agreement.

If the Lisp function returns NIL, the parser does not do the reduce action with the rule. If the Lisp function returns a non-NIL value, then this value is given to the newly created non-terminal. The value includes attributes of the nonterminal and a partial syntactic/semantic representation constructed thus far. Notice that those Lisp functions can be precompiled into machine code by the standard Lisp compiler.

6.2 Sharing and Packing Functional Structures

A functional structure used in the functional grammar formalisms (Kay 1984, Bresnan and Kaplan 1982, Shieber 1985) is in general a directed acyclic graph (dag) rather than a tree. This is because some value may be shared by two different attributes in the same sentence (e.g., the "agreement" attributes of subject and main verb). Pereira (1985) introduced a method to share dag structures. However, the dag structure sharing method is much more complex and computationally expensive than tree structure sharing. Therefore, we handle only tree-structured functional structures for the sake of efficiency and simplicity.[6] In the example, the "agreement" attributes of subject and main verb may thus have two different values. The identity of these two values is tested explicitly by a test in the augmentation. Sharing tree-structured functional structures requires only a minor modification on the subtree sharing method for the shared-packed forest representation described in subsection 3.1.

Local ambiguity packing for augmented context-free grammars is not as easy. Suppose two certain nodes have been packed into one packed node. Although these two nodes have the same category name (e.g., NP), they may have different attribute values. When a certain test in the Lisp function refers to an attribute of the packed node, its value may not be uniquely determined. In this case, the parser can no longer treat the packed node as one node, and the parser will unpack the packed node into two individual nodes again. The question, then, is how often this unpacking needs to take place in practice. The more frequently it takes place, the less significant it is to do local ambiguity packing. However, most of sentence ambiguity comes from such phenomena as PP-attachment and conjunction scoping, and it is unlikely to require unpacking in these cases. For instance, consider the noun phrase:

a man in the park with a telescope,

which is locally ambiguous (whether *telescope* modifies *man* or *park*). Two NP nodes (one for each interpretation) will be packed into one node, but it is unlikely that the two NP nodes have different attribute values which are referred to later by some tests in the augmentation. The same argument holds with the noun phrases:

pregnant women and children
large file equipment

Although more comprehensive experiments are desired, it is expected that only a few packed nodes need to be unpacked in practical applications.

6.3 THE LFG COMPILER

It is in general very painful to create, extend, and modify augmentations written in Lisp. The Lisp functions should be generated automatically from more abstract specifications. We have implemented the LFG compiler that compiles augmentations in a higher level notation into Lisp functions. The notation is similar to the Lexical Functional Grammar (LFG) formalism (Bresnan and Kaplan 1982) and PATR-II (Shieber 1984). An example of the LFG-like notation and its compiled Lisp function are shown in Figures 6.1 and 6.2. We generate only non-destructive functions with no side-effects to make sure that a process never alters other processes or the parser's control flow. A generated function takes a list of arguments, each of which is a value associated with each right-hand side symbol, and returns a value to be associated with the left-hand side symbol. Each value is a list of f-structures, in case of disjunction and local ambiguity.

That a semantic grammar in the LFG-like notation can also be generated automatically from a domain semantics specification and a purely syntactic grammar is discussed further in Tomita and Carbonell (1986). The discussion is, however, beyond the scope of this paper.

```
(<S>  <==>  (<NP> <VP>)
       (((x1 case) = nom)
        ((x2 form) =c finite)
        (*OR*
         (((x2 :time) = present)
          ((x1 agr) = (x2 agr)))
         (((x2 :time) = past)))
        ((x0) = (x2))
        ((x0 :mood) = dec)
        ((x0 subj) = (x1))))
```

Figure 6.1. Example grammar rule in the LFG-like notation.

```
(<S> <==> (<NP> <VP>)
(LAMBDA (X1 X2)
 (LET ((X (LIST (LIST (CONS (QUOTE X2) X2) (CONS (QUOTE X1) X1)))))
  (AND
   (SETQ X (UNIFYSETVALUE* (QUOTE (X1 CASE)) (QUOTE (NOM))))
   (SETQ X (C-UNIFYSETVALUE* (QUOTE (X2 FORM)) (QUOTE (FINITE))))
   (SETQ X (APPEND
           (LET ((X X))
            (SETQ X (UNIFYSETVALUE* (QUOTE (X2 :TIME)) (QUOTE (PRESENT)))))
            (SETQ X (UNIFYVALUE* (QUOTE (X2 AGR)) (QUOTE (X1 AGR))))
            X)
           (LET ((X X))
            (SETQ X (UNIFYSETVALUE* (QUOTE (X2 :TIME)) (QUOTE (PAST))))
            X)))
   (SETQ X (UNIFYVALUE* (QUOTE (X0)) (QUOTE (X2))))
   (SETQ X (UNIFYSETVALUE* (QUOTE (X0 :MOOD)) (QUOTE (DEC))))
   (SETQ X (UNIFYVALUE* (QUOTE (X0 SUBJ)) (QUOTE (X1))))
   (GETVALUE* X (QUOTE (X0))))))
```

Figure 6.2. The compiled grammar rule.

7 The On-Line Parser

Our parsing algorithm parses a sentence strictly from left to right. This characteristics makes **on-line parsing** possible; i.e., to parse a sentence as the user types it in, without waiting for completion of the sentence. An example session of on-line parsing is presented in Figure 7.1 for the sample sentence *I saw a man with a telescope.*

```
>_                                    Starts accepting a sentence.
>I_
>I _                                  Starts parsing "I".
>I s_
>I sa_
>I saw_
>I saw _                              Starts parsing "saw".
>I saw a_
>I saw a _                            Starts parsing "a".
>I saw a b_
>I saw a bi_
>I saw a big_
>I saw a big _                        Starts parsing "big".
>I saw a big m_
>I saw a big ma_                      User changes his mind.
>I saw a big m_
>I saw a big _
>I saw a big_                         Starts unparsing "big".
>I saw a bi_
>I saw a b_
>I saw a _
>I saw a m_
>I saw a ma_
>I saw a man_
>I saw a man _                        Starts parsing "man".
>I saw a man w_
>I saw a man wi_
>I saw a man wit_
>I saw a man with_
>I saw a man with _                   Starts parsing "with".
>I saw a man with a_
>I saw a man with a _                 Starts parsing "a".
>I saw a man with a t_
>I saw a man with a te_
>I saw a man with a tel_
>I saw a man with a tele_
>I saw a man with a teles_
>I saw a man with a telesc_
>I saw a man with a telesco_
>I saw a man with a telescop_
>I saw a man with a telescope_
>I saw a man with a telescope._  Starts parsing "telescope".
>I saw a man with a telescope.   User hits <return>. Ends parsing
```

Figure 7.1. Example of on-line parsing.

As in this example, the user often wants to hit the "backspace" key to correct previously input words. In the case in which these words have already been processed by the parser, the parser must be able to "unparse" the words, without parsing the sentence from the beginning all over again. To implement unparsing, the parser needs to store system status each time a word is parsed. Fortunately, this can be nicely done with our parsing algorithm; only pointers to the graph-structured stack and the parse forest need to be stored.

It should be noted that our parsing algorithm is not the only algorithm that parses a sentence strictly from left to right; Other left-to-right algorithms include Earley's (1970) algorithm, the active chart parsing algorithm (Winograd 1983), and a breadth-first version of ATN (Woods 1970). Despite the availability of left-to-right algorithms, surprisingly few on-line parsers exist. NLMenu (Tennant et al. 1983) adopted on-line parsing for a menu-based system but not for typed inputs.

In the rest of this section, we discuss two benefits of on-line parsing, quicker response time and early error detection. One obvious benefit of on-line parsing is that it reduces the parser's response time significantly. When the user finishes typing a whole sentence, most of the input sentence has been already processed by the parser. Although this does not affect CPU time, it could reduce response time from the user's point of view significantly. On-line parsing is therefore useful in interactive systems in which input sentences are typed in by the user on-line; it is not particularly useful in batch systems in which input sentences are provided in a file.

Another benefit of on-line parsing is that it can detect an error almost as soon as the error occurs, and it can warn the user immediately. In this way, on-line parsing could provide better man-machine communication. Further studies on human factors are necessary.

8 Conclusion

This paper has introduced an efficient context-free parsing algorithm, and its application to on-line natural language interfaces has been discussed.

A pilot on-line parser was first implemented in MacLisp at the Computer Science Department, Carnegie-Mellon University (CMU) as a part of the author's thesis work (Tomita 1985). The empirical results in section 5 are based on this parser.

CMU's machine translation project (Carbonell and Tomita 1986) adopts on-line parsing for multiple languages. It can parse unsegmented sentences (with no spaces between words, typical in Japanese). To handle unsegmented sentences, its grammar is written in a character-based manner; all terminal symbols in the grammar are characters rather than words. Thus, morphological rules, as well as syntactic rules, are written in the augmented context-free grammar. The parser takes about 1–3 seconds CPU time per sentence on a Symbolics 3600 with about 800 grammar rules; its response time (real time), however, is less than a second due to on-line parsing. This speed does not seem to be affected very much by the length of sentence or the size of grammar, as discussed in section 5. We expect further improvements for fully segmented sentences (such as English) where words rather then characters are the atomic units.

A commercial on-line parser for Japanese language is being developed in Common Lisp jointly by Intelligent Technology Incorporation (ITI) and Carnegie Group Incorporation (CGI), based on the technique developed at CMU.

Finally, in the continuous speech recognition project at CMU (Hayes et al. 1986), the on-line parsing algorithm is being extended to handle speech input, to make the speech parsing process efficient and capable of being pipelined with lower level processes such as acoustic/phonetic level recognition (Tomita 1986).

Aknowledgments

I would like to thank Jaime Carbonell, Phil Hayes, James Allen, Herb Simon, Hozumi Tanaka, and Ralph Grishman for their helpful comments on the early version of this paper. Kazuhiro Toyoshima and Hideto Kagamida have implemented the runtime parser and the LR table compiler, respectively, in Common Lisp. Lori Levin, Teruko Watanabe, Peggy Anderson, and Donna Gates have developed Japanese and English grammars in the LFG-like notation. Hiroaki Saito has implemented the algorithm for speech input. Ron Kaplan, Martin Kay, Lauri Karttunen, and Stuart Shieber provided useful comments on the implementation of LFG and dag structure sharing.

References

Aho, A.V. and Johnson, S.C. 1974 LR Parsing. *Computing Survey* 6(2): 99-124.

Aho, A.V. and Ullman, J.D. 1972 *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, Englewood Cliffs, New Jersey.

Aho, A.V. and Ullman, J.D. 1977 *Principles of Compiler Design.* Addison Wesley.

Bresnan, J. and Kaplan, R. 1982 Lexical-Functional Grammar: A Formal System for Grammatical Representation. *The Mental Representation of Grammatical Relations.* MIT Press, Cambridge, Massachusetts: 173-281.

Carbonell, J.G. and Tomita, M. 1986 Knowledge-Based Machine Translation, the CMU Approach. *Machine Translation: Theoretical and Methodological Issues.* Cambridge University Press.

Church, K. and Patil, R. 1982 Coping with Syntactic Ambiguity, or How to Put the Block in the Box on the Table. Technical Report MIT/LCS/TM-216, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts.

DeRemer, F.L. 1969 Practical Translators for LR(k) Languages. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts.

DeRemer, F.L. 1971 Simple LR(k) Grammars. *Communications ACM* 14(7): 453-460.

Earley, J. 1970 An Efficient Context-Free Parsing Algorithm. *Communications ACM* 6(8): 94-102.

Hayes, P.J.; Hauptmann, A.G.; Carbonell, J.G.; and Tomita, M. 1986 Parsing Spoken Language: A Semantic Caseframe Approach. In *Proceedings of the 11th International Conference on Computation Linguistics (COLING86).*

Kay, M. 1984 Functional Unification Grammar: A Formalism for Machine Translation. In *Proceedings of the 10th International Conference on Computational Linguistics:* 75-78.

Pereira, F.C.N. 1985 A Structure-Sharing Representation for Unification-Based Grammar Formalisms. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics:* 137-144.

Shieber, S.M. 1983 Sentence Disambiguation by a Shift-Reduce Parsing Technique. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics:* 113-118.

Shieber, S.M. 1984 The Design of a Computer Language for Linguistic Information. In *Proceedings of the 10th International Conference on Computational Linguistics:* 362-366.

Shieber, S.M. 1985 Using Restriction to Extend Parsing Algorithms for Complex-Feature-Based Grammar Formalisms. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics:* 145-152.

Tennant, H.R.; Ross, K.M.; Saenz, R.M.; Thompson, C.W.; and Miller, J.R. 1983 Menu-Based Natural Language Understanding. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics:* 151-158.

Tomita, M. 1984 LR Parsers for Natural Language. In *Proceedings of the 10th International Conference on Computational Linguistics (COLING84).*

Tomita, M. 1985 *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers, Boston, Massachusetts.

Tomita, M. 1986a Sentence Disambiguation by Asking. *Computers and Translation* 1(1): 39-51.

Tomita, M. 1986b An Efficient Word Lattice Parsing Algorithm for Continuous Speech Recognition. In *Proceedings of IEEE-IECE-ASJ International conference on Acoustics, Speech, and Signal Processing (ICASSP86).*

Winograd, T. 1983 *Language as a Cognitive Process.* Addison Wesley.

Woods, W.A. 1970 Transition Network Grammars for Natural Language Analysis. *Communications of ACM* 13: 591-606.

Notes

1. This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

   The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.
2. The situation is often called **conflict.**
3. Although it is possibly reduced if some processes reach error entries and die.
4. The term **node** is used for forest representation, whereas the term **vertex** is used for graph-structured stack representation.
5. In practice; not in theory.
6. Although we plan to handle dag structures in the future, tree structures may be adequate, as GPSGs use tree structures rather than dag structures.