An Efficient Context-free Parsing Algorithm For Natural Languages[1]

Masaru Tomita
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

This paper introduces an efficient context-free parsing algorithm and emphasizes its practical value in natural language processing. The algorithm can be viewed as an extended LR parsing algorithm which embodies the concept of a "graph-structured stack." Unlike the standard LR, the algorithm is capable of handling arbitrary non cyclic context-free grammars including ambiguous grammars, while most of the LR parsing efficiency is preserved. The algorithm seems more efficient than any existing algorithms including the Cocke Younger Kasami algorithm and Earley's algorithm, as far as practical natural language parsing is concerned, due to utilization of LR parsing tables. The algorithm is an all-path parsing algorithm; it produces all possible parse trees (a parse forest) in an efficient representation called a "shared-packed forest." This paper also shows that Earley's forest representation has a defect and his algorithm cannot be used in natural language processing as an all-path parsing algorithm.

1 Introduction

In past decades, many context-free parsing algorithms have been developed, and they can be classified into two groups: algorithms for programming languages and algorithms for general context-free languages. The former group of algorithms are intended to handle only a small subset of context-free grammars sufficient for programming languages. Such algorithms include the LL parsing algorithm, the operator precedence parsing algorithm, the predictive parsing algorithm and the LR parsing algorithm. They can handle only a subset of context free grammars called LL grammars, operator precedence grammars, predictive grammars and LR grammars, respectively [1]. These algorithms are tuned to handle a particular subset of context free grammars, and therefore they are very efficient with their type of grammars. In other words, they take advantage of inherent features of the programming language.

The other group of algorithms, often called general context-free parsing algorithms, are designed to handle arbitrary context-free grammars. This group of algorithms includes Earley's algorithm (9] and the Cocke Younger Kasami algorithm [19, 11]. General context-free languages include many difficult phenomena which never appear in programming languages, such as ambiguity and cycle. Algorithms in this group have not been widely used for programming languages, because their constant factors are too large to be used in practical compilers, as Earley admitted in his thesis [8]. This is not surprising, because those algorithms are not tuned for any particular subset of context-free grammars, and must be able to handle all difficult phenomena in context-free grammars. In other words, they do not take advantage of inherent features of the programming language. Intuitively speaking, algorithms in this group are efficient for "hard" grammars by sacrificing efficiency on "easy" grammars.

No parsing algorithm has been designed that takes advantage of inherent features of natural languages. Because natural languages include slightly more difficult phenomena than programming languages, we cannot simply use the first group of algorithms for natural languages. Natural languages are a little "harder" than programming languages, but they are still much "easier" than general context-free languages As we have seen above, we have context-free parsing algorithms at two extremes. The one is very efficient but not powerful enough to handle natural languages. The other is too powerful and it turns out to be inefficient. We need something in between.

This paper introduces such a context-free parsing algorithm, which can be viewed as an extended LR parsing algorithm which embodies the concept of a "graph-structured stack." The fragile point of the standard LR parsing algorithm is that it cannot handle a non-LR grammar, even if the grammar is almost LR. Unlike the standard LR parsing algorithm, our algorithm can handle non-LR grammars with little loss of LR efficiency, if its grammar is "close" to LR. Foriunateiy, natural language grammars are considerably "close" to LR, compared with other general context-free grammars.

A primitive version of the algorithm was described in the author's previous work [15]. Because the primitive algorithm used a "tree-structured stack", exponential time was required, whereas the current algorithm uses the "graph-structured stack" and runs in polynomial time. Also, the primitive algorithm was a recognizer; that is, it did not produce any parses, while the current algorithm produces all possible parses in an efficient representation. A "graph-structured stack" was proposed in the author's more recent work [16]. The algorithm was previously called the *MLR* parsing algorithm. All ideas presented in those two previous papers are included in this paper, and the reader does not need to refer to them to understand the current discussion.

2 The Standard LR Parsing Algorithm

The LR parsing algorithms [1, 2] were developed originally for programming languages. An LR parsing algorithm is a shift-reduce parsing algorithm which is 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[6, 7]. I do not describe the algorithms here, referring the reader to chapter 6 in Aho and Ullman [31. I assume that the reader is familiar with the standard LR parsing algorithm (not necessarily with the parsing tabie construction algorithm).

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 authors 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.

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-tree grammars called LR grammars, and it is almost certain that any practical natural language grammars are not LR. If a orammar is non-LR, its parsing table will have multiple entries ; one or more of the action table entries will be multiply defined.

Figures 1 and 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 "ace" stands for the action "accept", and blank spaces represent "error". Goto table (the right part of the table) decides to what state the parser should go after a reduce action. The exact definition and operation of the LR parser can be found in Aho and Ullman [3].

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". 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. However, in this paper, we extend a stack of the LR parsing algorithm to be "graph-structured," so that the algorithm can handle multiple entries with little loss of LR efficiency.

3 Handling Multiple Entries

As mentioned above, once a parsing table has multiple entries, determimst'C parsing is no longer possible and some kind of non-determinism is necessary. We handle multiple entries with a special technique, named a *graph-structuied stack.* In order to introduce the idea of a graph-structured stack, I first give a simpler non determinism, and make refinements on it. Subsection 3.1 describes a simple and straightforward non-determinism, i.e. pseudo-parallelism (breath-first search), in which the system maintains a number of stacks simultaneously. I call the list of stacks *Stack List.* A disadvantage of the stack list is then described. The next subsection describes the idea of stack combination, which was introduced in my earlier research [15], 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 algorithm even more efficient; efficient enough to run in polynomial time.

3.1 With Stack List

The simplest idea is to handle multiple entries non-deterministically. I adopt pseudo-parallelism (breath first search), maintaining a list of stacks called a *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 duplicating 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 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 1 and the parsing table in figure 2. For the sake of convenience, I 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.

We notice that some stacks in the stack list appear to be identical. They are, however, internally different because they have reached the current state in different ways. Although we shall describe a method to compress them into one stack in the next section, we consider them to be different in this section.

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. For example, these 14 processes in figure 3 will parse the rest of the sentence "the telescope" 14 times in exactly the same way. This can be avoided by using a tree structured stack, which is described in the following subsection.

3.2 With a Tree-structured 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. I 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

2 They are often called *conflict.*

```
(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 1: An Example Non-LR 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: LR Parsing Table with Multiple Entries

in parallel, so stacks are represented as a forest. Figure 4 shows a snapshot of the tree-structured stack immediately after shifting the word "with".

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

3.3 With a Graph-structured 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 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.

So far, we have focussed on how to accept or reject a sentence. In practice, however, the parser must not only simply 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.

4 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 grows exponentially as the length of a sentence grows. 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 ail 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: sub-tree sharing and local ambiguity packing. It should be mentioned that these two techniques are not completely new ideas, and some existing systems already adopted these techniques, either implicitly or explicitly. To the author's knowledge, however, no existing system has explicitly adopted both techniques at the same time.

4.1 Sub-tree Sharing

If two or more trees have a common sub-tree, the sub-tree 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 6. Our parsing algorithm is very well suited for building this kind of shared forest as its output, as we shall see in the following.

To implement this, we no longer push grammatical symbols on the stack; instead, we push pointers to a node[3] of the shared

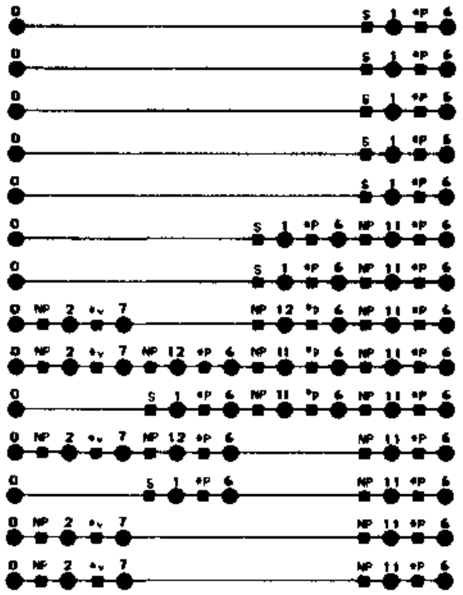


Figure 3: Stack List (simple parallelism)

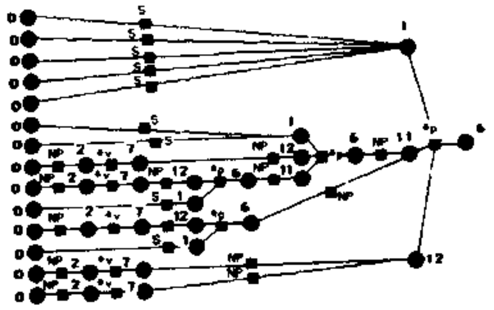


Figure 4: Tree-Structured Stack

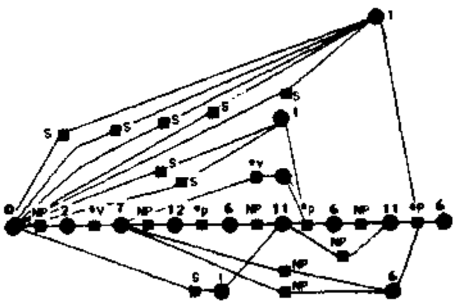


Figure 5: Graph-Structured Stack

[3]The term node is used for forest representation, whereas the term vertex is used for the graph-structured stack representation.

forest. When the parser "shifts" a word, it creates a leaf node labeled with the word and the preterminal, and instead of pushing 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, 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 the algorithm never does the same reduce action twice in the same manner.

4.2 Local Ambiguity Packing

I define 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. Examples of a shared-packed forest is shown in figure 7.

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 5 for example, we see one 5-way local ambiguity and two 2-way local ambiguities.

The algorithm will be made clear by an example in the next section.

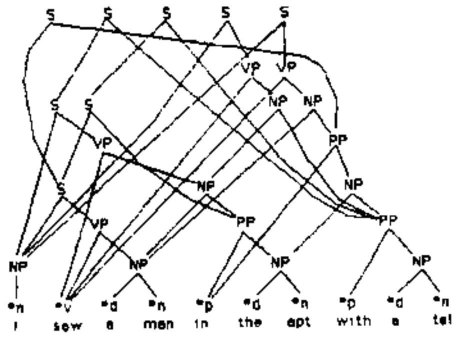


Figure 6: Shared Forest

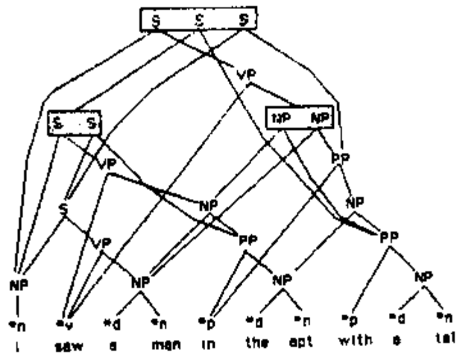
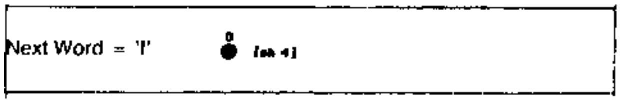


Figure 7: Shared-Packed Forest

5 The Example

This section gives a trace of the algorithm with the grammar In figure 1, the parsing table in figure 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.



Figu re 8: Trace of the Parser

When shifting the word " I " , me 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.

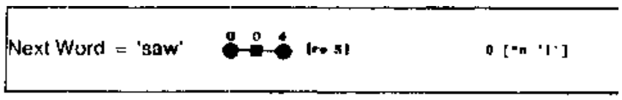


Figure 9: Trace of the Parser (cont.)

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 figure 10.

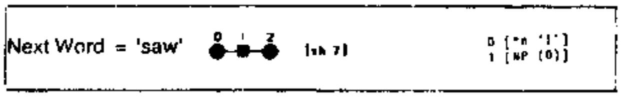


Figure 10: Trace of the Parser (cont.)

After executing "shift 7", we have figure 11.

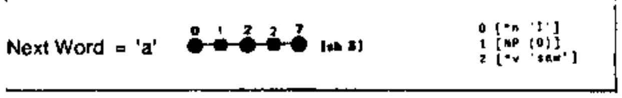


Figure 11: Trace of the Parser (cont.)

After executing "shift 3", we have figure 12.

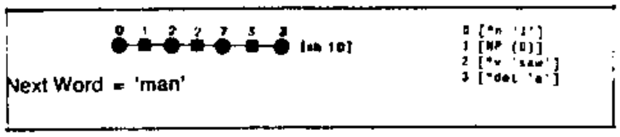


Figure 12: Trace of the Parser (cont.)

After executing "shift 10", we have figure 13.

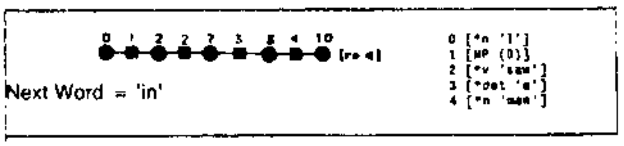
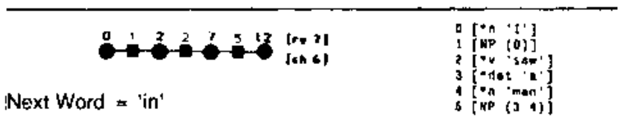


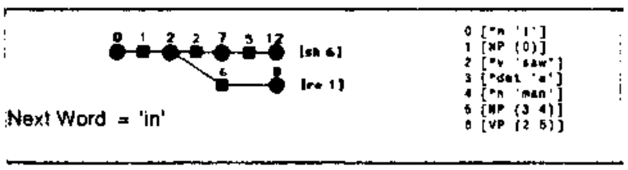
Figure 13: Trace of the Parser (cont.)

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 figure 14.



Figu re 14: Trace of the Parser (cont.)

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 reduce action to execute. 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 which is 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.



Figu re 15: Trace of the Parser (cont.)

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.

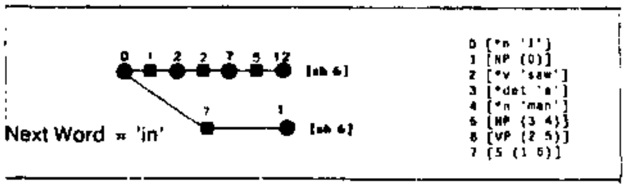
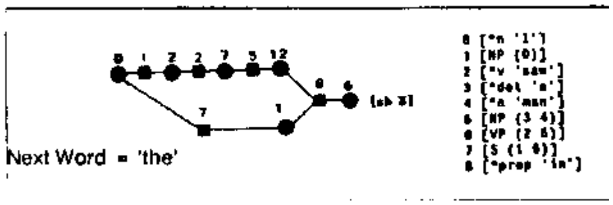


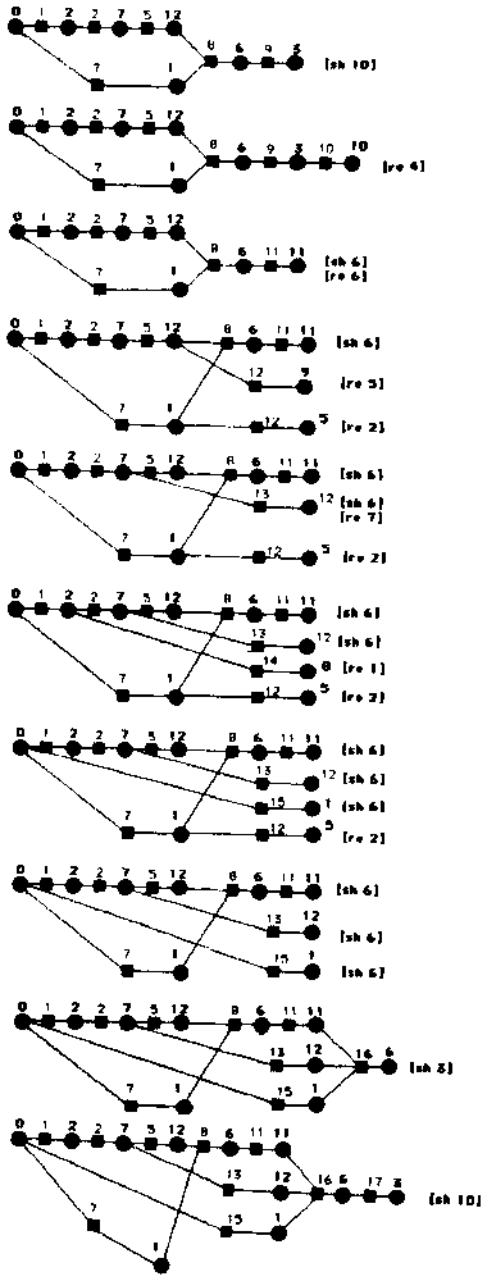
Figure 16: Trace of the Parser (cont.)

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.



Figu re 17: Trace of the Parser (cont.)

After about 20 steps (figure 18), 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 shown in figure 19.



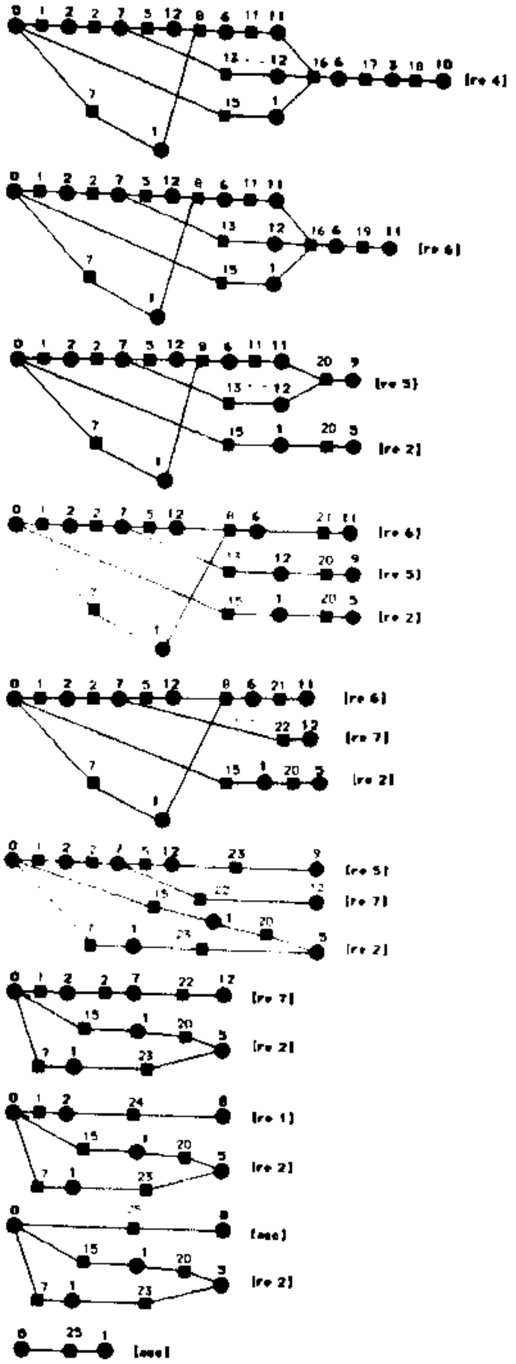


Figure 18: Trace of the Parser (cont.)

```
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)]
```

Figure 19: Final Parse Forest

6 Comparison with Other Algorithms

There have been several general parsing algorithms that run in polynomial timo. Theoretically speaking, the fastest algorithm at present is Valiant's algorithm. Valiant [18] reduced the context-free parsing problem to the Boolean Matrix Multiplication problem [10], and his algorithm runs in time $0(n^{?B1})$. This algorithm is, however, of only theoretical interest, because the coefficient of n^{281} is so large that the algorithm runs faster than conventional n algorithms only when an input sentence is tremendously long. Practically speaking, on the other hand, the most well-known parsing algorithm is Earley's algorithm [9, 8,1,11], which runs In time $0(n^3)$.

All other practical algorithms seem to bear some similarity with or relation to Earley's algorithm. Another algorithm which is as well-known as Earley's algorithm is the Cocke-Younger-Kasami (CYK) algorithm [19, 11, 1]. Graham *ei al.* [12], however, revealed that the CYK algorithm is "almost" identical to Earley's algorithm, by giving an improved version of the CYK algorithm which is very similar to Earley's algorithm. The *chart parsing algorithm* is basically the same as the CYK algorithm. The *active chart parsing algorithm* is basically the same as Earley's algorithm, although it does not necessarily have to parse from left to right. Bouckaert *et al.* [4] extended Earley's algorithm to perform tests similar to those introduced in LL and LR algorithms. *Improved nodal* span [5] and LINGOL [131 are also similar to Earley's algorithm, but both of them require grammars to be in Chomsky Normal Form (CNF).

These all practical general parsing algorithms seem to be like Earley's algorithm, in that they employ the tabular parsing method; they all construct *well formed substring tables* [14]. In chart parsing, such tables are called *charts.* The representation of one well formed substring is called an "edge" in active chart parsing, a "state" in Earley's algorithm, a "dotted rule" in Grahams algorithm and an "item" in Aho and Ullman[1]. Throughout this paper, we call a well-formed substring an *item.*

6.1 Recognition time

No existing general parsing algorithm utilizes LR parsing tables. All of the practical algorithms mentioned above construct *sets of items* by adding an item to a set, one by one, during parsing. Our algorithm, on the other hand, is sufficiently different; it precomputes sets of items in advance during the time of parsing table construction, and maintains pointers (i.e., state numbers) to the precomputed sets of items, rather than maintaining items themselves.

Because of this major difference, our algorithm ha3 the following three properties.

- It is more efficient, if a grammar is "close" to LR: that is, if its LR parsing table contains relatively few multiple entries. In general, less ambiguous grammars tend to have fewer multiple entries in their parsing table. In an extreme case, if a grammar is LR, our algorithm is as efficient as an LR parsing algorithm, except for minor overheads.
- It is less efficient, if a grammar is "densely" ambiguous as in figure 20. This kind of grammar tends to have many multiple entries in its LR parsing table. Our algorithm may take more than $0(n^3)$ time with "densely" ambiguous grammars.

```
S --> S S S S
S --> S S S
S --> S S
S --> x

sentence = 'xxxxxx'
```

Figure 20: Heavy Ambiguity

- It is not able to handle infinitely ambiguous grammars and cyclic grammars[4] (figure 21 and 22), although it can handle e grammars and left recursive grammars. If a grammar is cyclic, our algorithm never terminates. The existing general parsing algorithms can parse those sentences (figure 20, 21 and 22) still in time proportional ton..

```
S --> S S
S --> e
S --> x

sentence = 'xxx'
```

Figure 21: Infinite Ambiguity

```
S --> S
S --> x

sentence = 'x'
```

Figure 22: Cyclic Grammar

It is certain that no natural language grammars have infinite ambiguity or cyclic rules. It is also extremely unlikely that a natural language grammar has dense ambiguity such as that shown in figure 20. It is therefore safe to conclude that our algorithm is more efficient than any existing general parsing algorithms in terms of recognition time as far as practical natural language grammars are concerned.

6.2 Parse Forest Representation

Some of the existing general parsing algorithms leave a well-formed substring table as their output. In my opinion, these well-formed substring tables are not appropriate as a parser's final output, because it is not straightforward to simply enumerate all possible parse trees out of the tables; another search must be involved. Thus we define a *parse forest* as a representation of all possible parse trees out of which we can trivially enumerate all trees without any substantial computation.

For most natural language grammars, our shared-packed forest representation, described in section 4, takes less than or equal to $O(n^3)$ space. This representation, however, occasionally takes more than $0(n^3)$ space with densely ambiguous grammars. For example, it takes $0(n^5)$ space with the grammar in figure 20.

Earley, on the other hand, gave in his thesis [8] a parse forest representation which takes at most $0(n^3)$ space for arbitrary context-free grammars. However, the next subsection shows that his representation has a defect, and should not be used in natural language processing.[5] There exist some other algorithms that produce a parse forest in $0(n_3)$ space, but they require their grammars to be Chomsky Normal Form (CNF). Theoretically speaking every context free grammar can be mechanically transformed into CNF. Practically speaking, however, it is usually not a good idea to mechanically transform a grammar into CNF, because the parse forest obtained from a CNF grammar will make little sense in practical applications; it is often hard to figure out a parse forest in accordance with its original grammar.

6.3 Defect of Earley's Forest Representation

This subsection identifies the defect of Earley's representation. Consider the following grammar G1 and the sentence in figure 23. Figure 24 is the parse forest produced by Earley's algorithm. The individual trees underlying in this representation are shown in figure 25. They are exactly what should be produced from the grammar and the sentence.

```
S --> e
S --> S J
J --> F
J --> I
F --> x
I --> x
sentence = 'xx'
```

Figure 23: Grammar G1

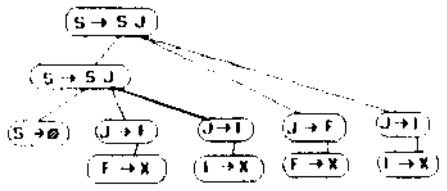


Figure 24: Earley's Parse Forest

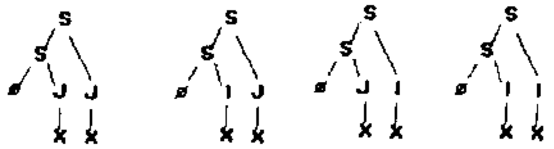


Figure 25: Underlying Parse Trees

Next consider the grammar G2 and the sentence in figure 26.

```
S --> S S
S --> x

sentence = 'xxx'
```

Figure 26: Grammar G2

The two possible parse trees out of this sentence are shown in figure 27.

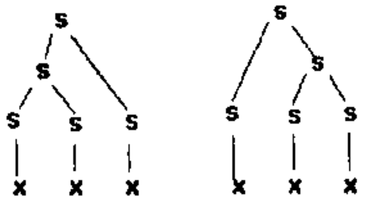


Figure 27: Correct Parse Trees

However, Earley's parsing algorithm produces the following representation.

Those two kinds of grammars are equivalent.

Several existing chart parsers seem to build a parse forest by adding pointers between edges. Since none of them gave a specification of the parse forest representation, we cannot make any comparisons In any event, however, if they adopt Earley's representation then they must have the defect, and if they adopt my representation then they must occasionally take more than 0(n) time.

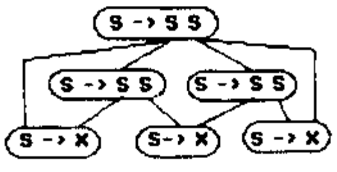


Figure 28: Defective Representation

This representation over-represents the trees; it represents not only the intended two parse trees, but also two other incorrect trees which are shown in figure 29.[6]

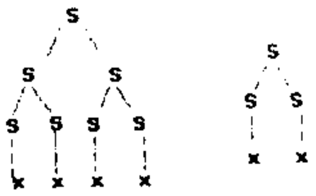


Figure 29: Wrong Parse Trees

Similarly, out of the sentence 'xxxx' with the same grammar G2, the algorithm produces a representation which over-represents 36 trees including 31 wrong trees along with 5 correct parse trees.

A grammar like G2 is totally unrealistic in the domain of programming language, and this kind of defect never appears as a real fault in that context. Productions like

S -> SS

in G2 look rather tricky and one might suspect that such a problem would arise only in a purely theoretical argument.

Unfortunately, that kind of production is often included in practical natural language grammars. For example, one might often include a production rule like

N -> NN

to represent compound nouns. This production rule says that two consecutive nouns can be compounded as a noun, as in 'file equipment' or 'bus driver.' This production rule is also used to represent compound nouns that consist of three or more nouns such as 'city bus driver or 'IBM computer file equipment.' In this case, the situation is exactly the same as the situation with the grammar G2 and the sentence 'xxx' or 'xxxx', making the defect described in the previous section real in practice.

Another defective case is that using conjunctive rules such as

NP -> NP conj NP
VP -> VP conj VP

which are even more often included in practical grammars. The same problem as that above arises when the algorithm parses a sentence with the form:

NP and NP and NP.

Yet another defective case which looks slightly different but which causes the same problem is that with the following productions:

NP -> *HP* PP
PP -> prep NP

These represent prepositional phrase attachment to noun phrases. The fault occurs when the algorithm parses sentences with the form:

NP prep NP prep NP. . . . .

As we have seen, it is highly likely for a practical grammar to have defective rules like those above, and we conclude that Earleys representation of a parse forest cannot be used for natural languages.

7 Concluding Remarks

Our algorithm seems more efficient than any of the existing algorithms as far as practical natural language parsing Is concerned, due to its utilization of LR parsing tables. Our shared-packed representation of a parse forest seems to be one of the most efficient representations which do not require CNF.

The following extensions of this paper can be found in my doctorate dissertation [17]:

- The algorithm is implemented and tested against four sample English grammars and about 50 sample sentences, to verify the feasibility of the algorithm to be used in practical systems.
- Earleys algorithm is also implemented and practical comparisons are made. The experiments show that our parsing algorithm is about 5 to 10 times faster than Earleys algorithm, as far as natural language processing is concerned.
- The algorithm's precise specification, as well as the source program, is presented.
- Multi part-of speech words and unknown words are handled by the algorithm without any special mechanism.
- An interactive disambiguation technique out of the shared-packed forest representation is described.
- An application to natural language interface, called *left-to-right on-line parsing,* is discussed, taking advantage of the algorithms left-to-right-ness.

Acknowledgement

I would like to thank Jaime Carbonell, Phil Hayes, Herb Simon and Ralph Grishman for thoughtful comments on an earlier version of this paper, and Cynthia Hibbard for helping to produce this document.

[6] We could think of an algorithm that takes the defective representation as its argument, and enumerate only the intended parse trees, by checking the consistency of leaf nodes of each tree. Such an algorithm would, however, require the non-trivial amount of computation, violating our definition of parse forest.

References

[I] Aho, A. V. and Ullman, J. D.
1 he theory of Parsing, Translation and Compiling.
Prentice-Hall, Englewood Cliffs, N. J., 1972.

[2] Aho, A. V. and Johnson, S. C.
LR parsing.
Computing Surveys 6:2:99 124, 1974.

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

[4] Bouckaert, M., Pirotte, A. and Snelling, M.
Efficient Parsing Algorithms for General Context-free Grammars.
Inf. Sci. 8(1)-1-26, Jan, 1975.

[5] Cocke, J. and Schwartz, J. I.
Programming Languages and Their Compilers.
Courant Institute of Mathematical Sciences, New York University, New York, 1970.

[6] Deremer, F. I.
Practical Translators for LR(k) Languages.
PhD thesis, MIT, 1969.

[7] DeRemer, F. L.
Simple LR(k) grammars.
Comrn. ACM 14:7:453 460, 1971.

[8] Earley, J.
An Efficient Context free Parsing Algorithm.
PhD thesis, Computer Science Department, Carnegie-Mellon University, 1968.

[9] Earley, J.
An Efficient Context free Parsing Algorithm.
Communication of ACM 6(8):94-102. February, 1970.

[10] Fischer, M. J. and Meyer, A. R.
Boolean Matrix Multiplication and Transitive Closure.
In *IEEF Conf. Rec. Symp. Switching Automata Theory,* pages 129-131. 1971.

[11] Graham, S. L. and Harrison, M. A.
Parsing of General Context-free Languages.
Academic Press, New York, 1976, pages 77-185.

[12] Graham, S. L., Harrison, M. A. and Ruzzo, W. L.
An Improved Context free Recognizer.
ACM Transactions on Programming Languages and Systems 2(3):415-4G2, July, 1980.

[13] Pratt, V. R.
LINGOL - A Progress Report.
In *Proc. of 4th IJCAI,* pages pp.327-381. August, 1975.

[14] Sheil, B.
Observations on context free parsing.
Statistical Methods in Linguistics :71-109, 1976.

[15] Tomita, M.
LR Parsers For Natural Languages.
In *COLING'84.* 1984.

[16] Tomita, M.
An Efficient All-Paths Parsing Algorithm for Natural Langauges.
Technical Report CMU-CS-84-163, Computer Science Department, Carnegie-Mellon University, Oct., 1984.

[17] Tomita, M.
An Efficient Context-free Parsing Algorithm for Natural Languages and Its Applications.
PhD thesis, Computer Science Department, Carnegie-Mellon University, May, 1985.

[18] Valiant, L.
General Context-free Recognition in Less than Cubic Time.
J. Comput Syst. Sci. 10:308-315, 1975.

[19] Younger, D. H.
Recognition and Parsing of Context-free Languages in time n^3.
Information and Control 10(2): 189-208, 1967.

Table of Contents