Current Trends on Parsing - A Survey

Hozumi Tanaka

Professor, Department of Computer Science,
Tokyo Institute of Technology, Tokyo-152, Japan

1 Introduction

The history of parsing natural languages began with the formal linguistic theory developed by N.Chomsky who classified languages into four classes, unrestricted languages, context-sensitive languages, context-free languages and regular languages. These languages are produced by applying rewriting rules, or otherwise called production rule, in the form of $\alpha \rightarrow \beta$ in which a string α is rewritten as another string β. Chomsky pointed out that only context-free grammar is not enough to specify a natural language, and he insisted on the necessity of context sensitiveness. However from the point of practical parsing, it is very difficult to device a parsing algorithm for context-sensitive grammar. Recently, Gazdar et.al., advocate the power of context-free grammar in covering broad range of natural languages [Gazdar, 85]. As many efficient parsing algorithms have been developed for context-free grammars, most natural language processing systems make use of context-free grammars. The following is a sample of English *context-free grammar (CFG).*

(1)	S	→	NP VP	(8)	n	→	'I'
(2)	S	→	S PP	(9)	n	→	man
(3)	NP	→	n	(10)	n	→	park
(4)	NP	→	det n	(11)	v	→	saw
(5)	NP	→	NP PP	(12)	det	→	a
(6)	PP	→	p NP	(13)	det	→	the
(7)	VP	→	v NP	(14)	p	→	in

Figure 1-1: A sample of English context-free grammar

Each rule in CFG is in the form of lhs → rhs as in fig.1-1 where lhs (left hand side) is allowed to have only one symbol. The symbols appeared on the lhs of each rule are called *nonterminal* symbols. In case of fig.1-1, *"S, NP, PP, VP, n, v, det, p"* are nonterminal symbols. The symbols not appeared on the left hand side are termed as *terminal symbols* such as *"'I', man, park, saw, a, the, in".* The nonterminal symbols, which directly produce terminal symbols are called *preterminal symbols* such as *"n, v, det, p"* in fig.1-1. A symbol S is called a *start symbol* and if an input sentence is derived from S applying CFG rules, the input sentence is recognized or parsed by the CFG. The following is a *derivation* of the sentence "I saw a man" from S:

S ⇒ NP VP ⇒ n VP ⇒ 'I' VP ⇒ 'I' v NP ⇒ 'I' saw NP ⇒ 'I' saw det n ⇒ 'I' saw a n ⇒ 'I' saw a man, which corresponds to the following tree derivation process.

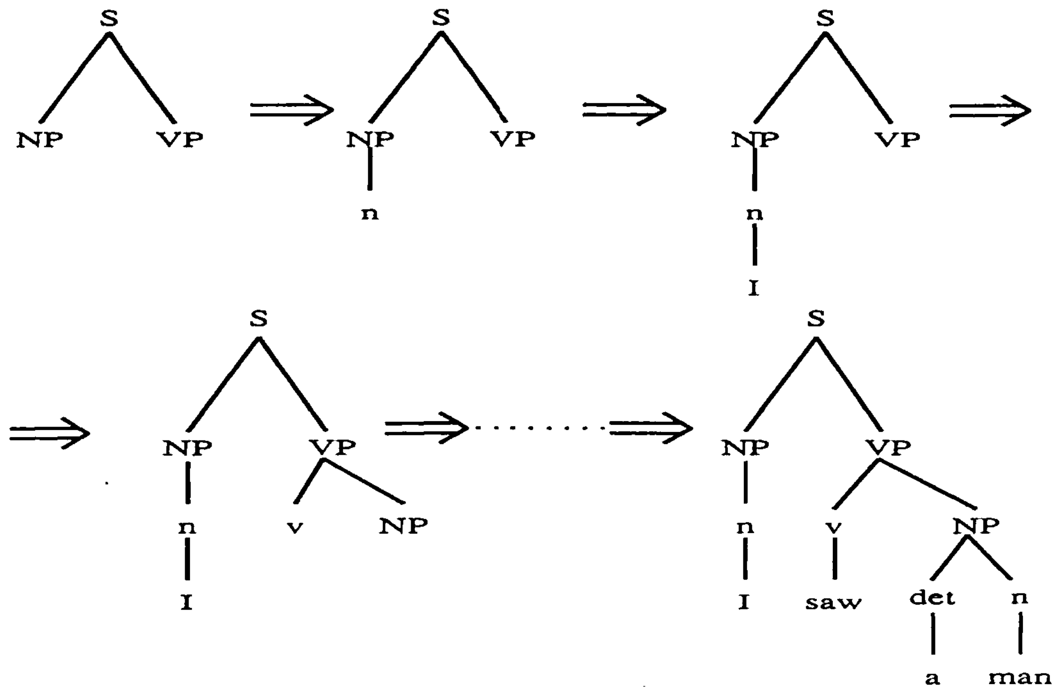


Figure 1-2 : Derivation tree of the input "'I' saw a man"

The above derivation is called a *leftmost derivation*, which rewrites the leftmost nonterminal symbols in each step. Similarly we can define a *rightmost derivation* by rewriting the rightmost nonterminal symbols.

Now we can give an informal definition of parsing:
Given a CFG and an input sentence, find a derivation of the input sentence from a start symbol S applying CFG rules.

There are two strategies to find these derivations; one is *top-down* and the other is *bottom-up*. In case of top-down parsing, the parser reads a CFG rule in a usual way along with the direction of the arrow in CFG rules. The lhs symbol (goal) is rewritten into rhs symbols (subgoals) until a sequence of terminal symbols (an input sentence to be parsed) is obtained. For example, using the rule (1) in fig.1-1, in order to accomplish S, we have to examine whether NP and VP will be able to be accomplished under the constraints that the terminal symbols derived from NP and VP are adjacent in the input sentence (see fig.1-3).

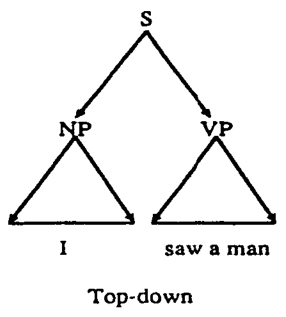


Figure 1-3 : Top-down parsing

On the contrary, in case of bottom-up parsing, the parser reads a CFG rule in a reverse direction. In case of rule (1), after accomplishing NP and VP, S is accomplished combining the NP and VP (see fig.1-4).

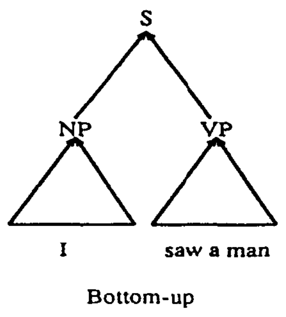


Figure 1-4 : Bottom-up parsing

In the current trends on natural language parsing, Earley's parsing algorithm has been used because of its efficiency in parsing. In section 2, we will explain Earley's parsing algorithm [Earley, 70],[Aho, 72].

In section 3, Chart parsing algorithm will be explained which is developed by M.Kay [Kay, 80] and its bottom-up version has been used in practical natural language processing systems as well as speech recognition systems. Both the Earley's and Chart parsers can handle any general CFG.

Knuth developed a LR parsing algorithm [Knuth, 65] which could parse an input sentence deterministically and efficiently, but the input sentences were limited to the ones generated by LR grammar, which is a proper subset of CFG. In section 4, we will explain generalized LR parsing algorithm [Tomita, 86], [Kipps, 89], [Tanaka, 92], which can handle any sentences generated by CFG. Empirically, generalized LR parsing algorithm is the most efficient parsing algorithm.

In general, these parsing algorithms might produce a tremendous amount of parsing results (parsing trees) for a long input sentence. From "I saw a baby with the toy", we will get at least two parsing results by applying the rules from (1) to (7) of the CFG in fig.1-1. Two are correct parsing results from the syntactic point of view, but only one of them is a plausible parsing result from the semantic point of view. However, semantic processing is one of the most difficult tasks of natural language processing. Recently, many researchers pay attention to statistical methods to select probable parsing results. For this purpose, probabilistic grammar [Wetherell, 80], [Wright, 90], [Fujisaki, 91] has been used. This will be explained briefly in section 5.

For the simplicity, we avoid the epsilon rule (null production as $X \rightarrow \epsilon$) in the following discussions. The readers interested in handling the epsilon rule should consult the related works in references. As a final comment on fig.1-1, rules from (8) to (14) are often called lexical entries in a dictionary.

2 Earley's parsing algorithm

In this section we briefly sketch *Earley's parsing algorithm.* This is basically a top-down parsing algorithm in which all possible parses are carried simultaneously in such a way that all identical subparses can be combined in order to avoid duplicated efforts on computation. Thus the time complexity is restricted in the order of n^3, where n is the length of an input sentence.

Earley's algorithm uses dotted CFG rules called *items*, which has a dot in its rhs. Let the input sentence be "${}_0 I_1 saw_2 a_3 man_4 in_5 the_6 park_7$". Here the numbers appeared between words are called position numbers. Considering the CFG rule (1) of fig.1-1, we will have three types of dotted items as shown below, the notation of which is slightly different from the one given by Earley [Earley, 70].

$[S \rightarrow \cdot \text{ NP VP}, 0, 0]$
$[S \rightarrow \text{NP} \cdot \text{VP}, 0, 1]$
$[S \rightarrow \text{NP VP} \cdot, 0, 4]$

The first item indicates that the input sentence is going to be parsed applying the rule $S \rightarrow$ NP VP from position 0. The second item indicates that a portion of the input sentence from position number 0 to 1 has been parsed as NP and the remainder left to be satisfied as VP. The third item indicates that the portion of input sentence from position number 0 to 4 has been parsed as NP VP and thus S is accomplished. Note that the combination of the second item and the third item will produce a parsing tree in which NP covers the portion of the input sentence from 0 to 1 and VP covers from 1 to 4. If we have the fourth item $[S \rightarrow \text{NP VP} \cdot$, 0, 7], then the combination of the second item and fourth item will produce another parsing tree where NP covers the portion from 0 to 1 and VP covers from 1 to 7. Thus the second item is shared by both parsing trees. The figure 2-1 will further clarify the situation.[1]

[1]A set of items created by the Completer and Scanner operations explained in this section, are used to generate parsing trees. The tree generation algorithm is shown in [Aho, 72] and if CFG include ambiguities, it takes time in the order of n^2 to generate a parsing tree.

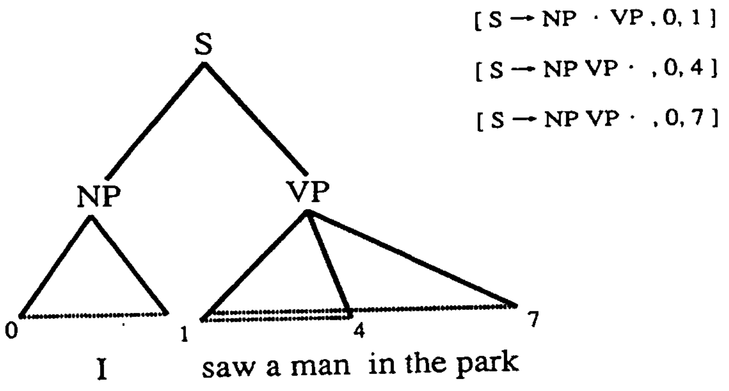


Figure 2-1 : Sharing of a partially parsed tree

Earley's algorithm uses three operators, *Predictor, Scanner* and *Completer*. In the following α, β, γ are a sequence of terminal and/or nonterminal symbols and S, A, B are nonterminal symbols.

Predictor operation:

For an item of the form [A $\rightarrow \alpha \cdot$ B β, i, j], create [B $\rightarrow \cdot\gamma$, j, j] for each production of the form B $\rightarrow \gamma$.

Completer operation:

For an item of the form [B $\rightarrow \gamma\cdot$, j, k], create [A $\rightarrow \alpha$ B $\cdot\beta$, i, k] ($i < j < k$) for each item in the form of [A $\rightarrow \alpha\cdot$ B β, i, j] if exists.

Scanner operation:

For an item of the form [A $\rightarrow \alpha\cdot$ w β, i, j], create [A $\rightarrow \alpha$ w $\cdot\beta$, i, j+1] if w is a terminal symbol appeared in the input sentence between j and j+1.

The Predictor operations correspond to top-down parsing. Figure 2-2 illustrates the above three operations.

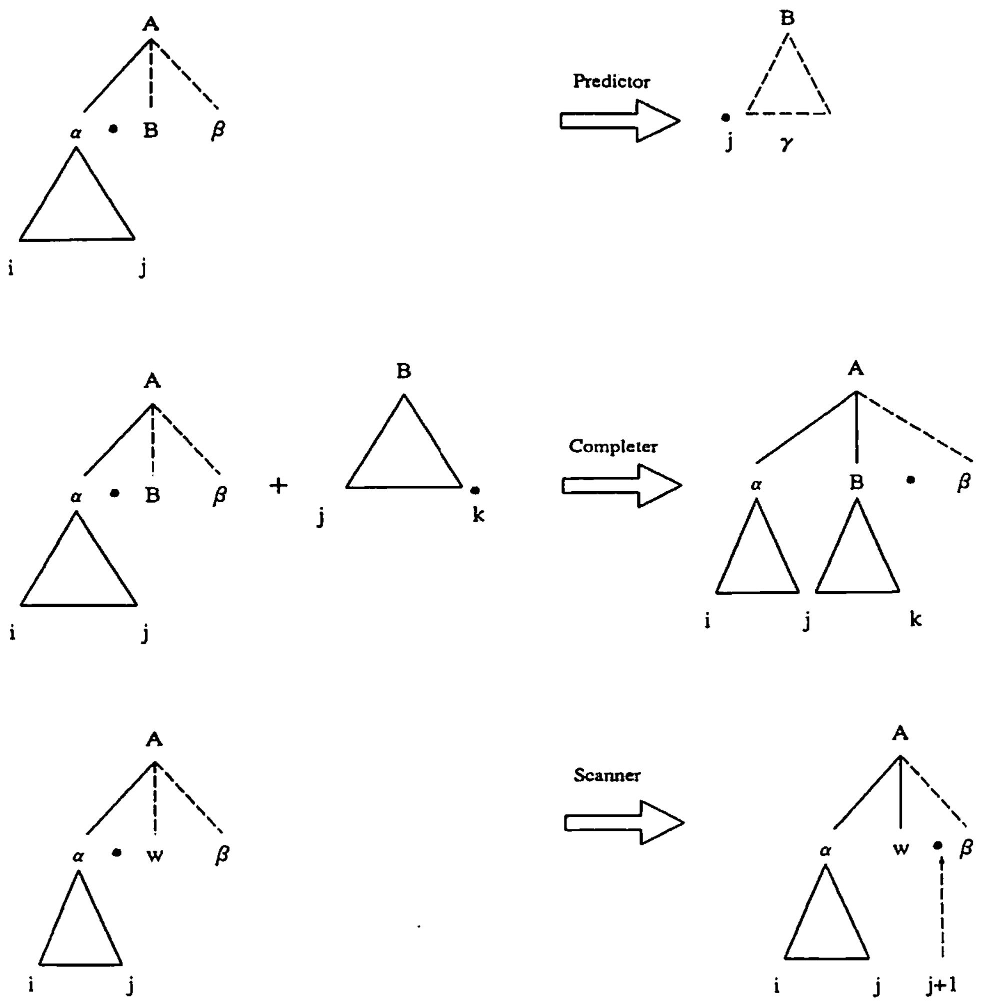


Figure 2-2 : Predictor, Completer and Scanner operations

Now we will give Earley's parsing algorithm:

```
For each production S → $\alpha$, create [S → $\cdot\alpha$, 0, 0].
For j = 0 to n do
    for each item in the form of [A → $\alpha\cdot$ B $\beta$, i, j], apply Predictor operation
    while a new item is created.
    for each item in the form of [B → $\gamma\cdot$ i, j], apply Completer operation
    while a new item is created.
    for each item in the form of [A → $\alpha\cdot$ w $\beta$, i, j], apply Scanner operation.
```

If we find an item of the form [S → $\alpha\cdot$, 0, n], then accept else reject, where n is the length of the input sentence.

Let us consider a grammar in fig.1-1 and an input sentence is "I saw a man in the park." Earley's parsing will proceed as follows.

Initialization.
1) [S → · NP VP, 0, 0]
2) [S → · S PP, 0, 0]
Apply Predictor to 1) and 2)
3) [NP → · n, 0, 0]
4) [NP → · det n, 0, 0]
5) [NP → · NP PP, 0, 0]
Apply Predictor to 3)
6) [n → · 'I', 0, 0]
Apply Scanner to 6)
7) [n → 'I' ·, 0, 1]
Apply Completer to 7) with 3)
8) [NP → n ·, 0, 1]
Apply Completer to 8) with 1) and 5)
9) [S → NP · VP, 0, 1]
10) [NP → NP · PP, 0, 1]
Apply Predictor to 9)
11) [VP → · v NP, 1, 1]
Apply Predictor to 11)
12) [v → · saw, 1, 1]
Apply Predictor to 10)
13) [PP → · p NP, 1, 1]
Apply Scanner to 12)
14) [v → saw ·, 1, 2]
Apply Completer to 14) with 11)
15) [VP → v · NP, 1, 2]
Apply Predictor to 15)
16) [NP → · n, 2, 2]
17) [NP → · det, n, 2, 2]
18) [NP → · NP PP, 2, 2]
... ...

The reader can easily continue the rest of the parsing. When applying Predictor operations, Earley's algorithm often creates a set of similar items such as 3),4),5) and 16),17),18) expecting NP in the future. On using LR table, the GLR parsing algorithm can avoid creating such similar items by Predictor operations, and thus make an efficient parsing.

The careful reader will be able to understand how Earley's algorithm blocks left recursive derivations. Considering the item 2) with a *left recursive rule* [S → · S PP, 0,0], and applying Predictor operation, it seems to create the same item as in 2) repeatedly, but Earley's algorithm prevents this by saying "apply Predictor operation while a new item is created". In other words, only new items, which were not created before, are generated by Predictor operation.

3 Chart parsing algorithm

In recent years, *chart parsing algorithm* [Kay 80] have been used broadly. The algorithm uses a data structure called *chart* as a book-keeping storage for all the information produced while parsing. The information in the chart is refered to avoid doing redundant works. This is essentially similar to the purpose of items in Earley's algorithm. Hence, the time complexity of parsing a sentence of length n is restricted in the order of n^3 for general CFGs as in the case of using Earley's algorithm.

The information kept in the chart are divided into a set of *active edges* and a set of *complete(inactive) edges*. A complete edge represents a constituent that has been completed. An active edge represents a constituent with some elements called *remainder* left to be satisfied. As "edges" in chart algorithm are essentially the same as "items" in Earley's algorithm, we will specify an edge by using the same representation for dotted items in last section. The elements after the dot then represent the remainder. For example, [S $\rightarrow$ NP $\cdot$ VP, 0, 1] represents an active edge of constituent S starting at position 0, ending at position 1, which has already found its NP, and still needs a VP to be completed. [S $\rightarrow$ NP VP $\cdot$, 0, 4] is an example of a complete edge whose all elements have been found.

Chart provides a very general framework to natural language analysis. The edges in the chart can be used by any parsing strategy up to the way they are processed. In fact, Earley's algorithm can be viewed as a specialization of chart parsing algorithm which proceeds in the top-down manner. For comparison, in this section we will discuss the bottom-up version of chart parsing.

Now we will give a left to right bottom-up chart parsing algorithm. In the following α, β, γ are a sequence of terminal and/or nonterminal symbols, and A,B,C are nonterminal symbols. S is a special symbol representing the goal symbol.

Input $w_1 w_2 \ldots w_n$

Output a chart

Algorithm[2]

For $k = 0$ to $n - 1$ do

(a) For each entry C $\rightarrow w_{k+1}$ in the lexicon, span an inactive edge [C $\rightarrow w_{k+1}\cdot$, k, $k+1$] between positions k and $k+1$

Then for each inactive edge between positions j and $k+1 (j < k+1)$ [B $\rightarrow \gamma\cdot$, $j, k+1$], do the following until no new item can be created:

(b) for each rule A $\rightarrow$ B, span an inactive edge [A $\rightarrow$ B $\cdot$, $j, k+1$].

(c) for each rule A $\rightarrow$ B β, span an active edge [A $\rightarrow$ B $\cdot\beta$, $j, k+1$].

[2] In this case eliminations of epsilon rule makes the following descriptions a slightly longer one.

(d) for each active edge starting at position i, ending at position j and having B as the leftmost element of the remainder with form $[\mathrm{A} \rightarrow \alpha \cdot \mathrm{B}\ ,\ i, j]$, create an inactive edge $[\mathrm{A} \rightarrow \alpha\ \mathrm{B}\ \cdot, i, k+1]$ between positions i and $k+1$.

(e) for each active edge starting at position i, ending at position j and having B as the leftmost element of the remainder with form $[\mathrm{A} \rightarrow \alpha \cdot \mathrm{B}\ \beta, i, j]$, create an active edge $[\mathrm{A} \rightarrow \alpha\ \mathrm{B} \cdot \beta, i, k+1]$ between positions i and $k+1$.

If we find an edge of the form $[\mathrm{S} \rightarrow \alpha \cdot, 0, n]$, then accept else reject.

Note that step (b) and (d) generate new inactive edges which will be in turn applied with step (b)-(e). The loop will be done until no more new item is generated.

To understand the algorithm more clearly, let us consider parsing the input sentence "I saw a man in the park" using the grammar in fig. 1-1. Bottom-up chart parsing will proceed as follows.

```
k = 0, $w_1$ = 'I'
  Apply (a) to $w_1$
  (1)  [n → 'I' ·, 0, 1]
  Apply (b) to (1)
  (2)  [NP → n ·, 0, 1]
  Apply (c) to (2)
  (3)  [S → NP · VP, 0, 1]
  (4)  [NP → NP · PP, 0, 1]
k = 1, $w_2$ = 'saw'
  Apply (a) to $w_2$
  (5)  [v → saw ·, 1, 2]
  Apply (c) to (5)
  (6)  [VP → v · NP, 1, 2]
k = 2, $w_3$ = 'a'
  Apply (a) to $w_3$
  (7)  [det → a ·, 2, 3]
  Apply (c) to (7)
  (8)  [NP → det · n, 2, 3]
k = 3, $w_4$ = 'man'
  Apply (a) to $w_4$
  (9)  [n → 'man' ·, 3, 4]
  Apply (b) to (9)
  (10)  [NP → n ·, 3, 4]
  Apply (c) to (10)
  (11)  [S → NP · VP, 3, 4]
  (12)  [NP → NP · PP, 3, 4]
  Apply (d) to (8) & (9)
  (13)  [NP → det n ·, 2, 4]
  Apply (c) to (13)
```

(14) [S → NP · VP, 2, 4]
Apply (c) to (13)
(15) [NP → NP · PP, 2, 4]
Apply (d) to (6) & (13)
(16) [VP → v NP ·, 1, 4]
Apply (d) to (3) & (13)
(17) [S → NP VP · , 0, 4]
Apply (c) to (15)
(18) [S → S · PP, 0, 4]
. . . .

Continuing in the same way, the rest of the input would be analysised as a preposition phrase(PP) which is represented by the following edge.

(19) [PP → p NP ·, 4, 7]
Then this edge would be used to create the following.
Apply (d) to (15)&(19)
(20) [NP → NP PP ·, 2, 7]
Apply (d) to (18)&(19)
(21) [S → S PP ·, 0, 7]
Apply (d) to (6)&(20)
(22) [VP → v NP · , 1, 7]
Apply (d) to (3)&(22)
(23) [S → NP VP ·, 0, 7]

As two edges of the form [S → $\alpha\cdot$, 0, 7] are generated, the input is accepted as a correct sentence that has two syntactic ambiguities.

Simply stating, bottom-up chart parsing is started by constructing inactive edges corresponding to the individual words and their categories. If there are more than one category for a word, all edges corresponding to all of its categories will be constructed. Then the algorithm tries to construct larger constituents from those inactive edges by matching them with CFG rules or by merging them with active edges. For example, the inactive edge (1) is matched with the rule NP → n in the grammar to construct the active edge (3), and the inactive edge (9) is merged with the active edge (8) to construct the inactive edge (13). If at last some edges of the category S are spanned from position 0 to position n, the input would be parsed as a correct sentence.

A chart can be viewed as a graph where each vertex represents a position in a sentence and each arc linking vertices represents an edge. A visualized version of chart is shown in fig. 3-1.

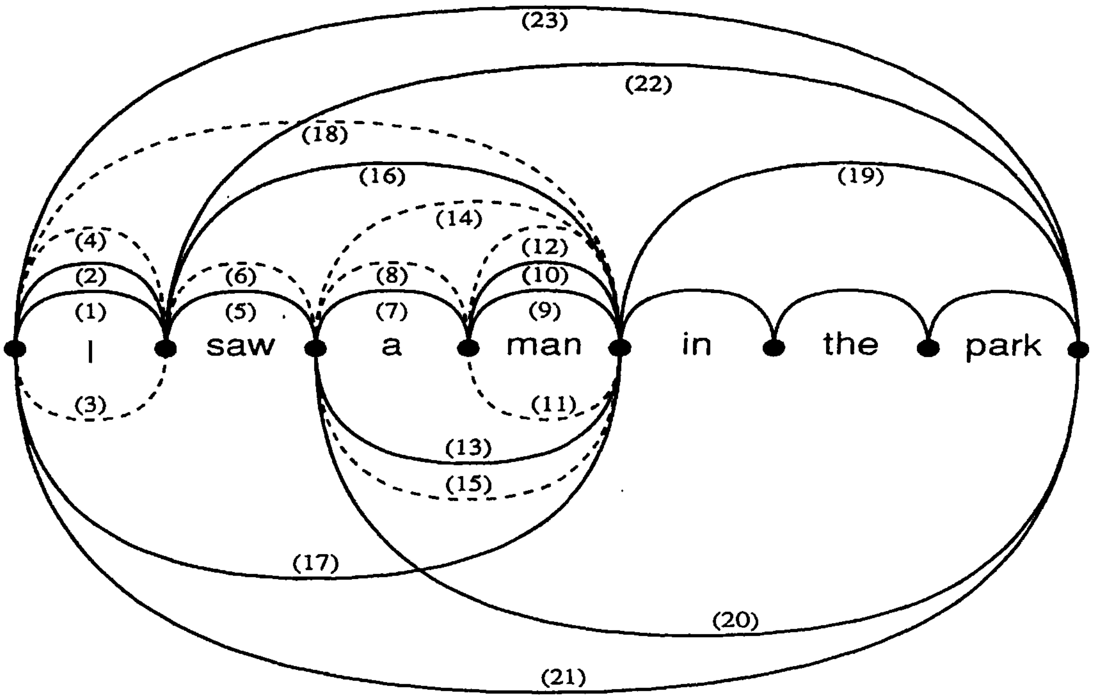


Figure 3-1 : The chart of parsing "'I' saw a man in the park"

The bottom-up and top-down strategies have different advantages. Since a bottom-up strategy starts by looking at the input and then building up the larger structures, the structures that cannot be projected down to the input words will never be built. On the other hand, because a top-down strategy begins by looking for a sentence and then searching for its substructures, the structures that cannot be projected up to the structure of a sentence would not be constructed. Mixing both strategies in a proper way can achieve more efficient parsing[Winograd 83].

4 GLR parsing algorithm

The *LR(k) parser* [Knuth, 65], which is one of the most efficient *shift/reduce parser* based on rightmost derivations. It can parse deterministically and efficiently any input sentences generated by a LR(k) grammar which is a subset of CFG. Tomita extended the LR(k) parser to handle a general CFG not limited to Chomsky normal form [Tomita 86]. The extended algorithm is called *Tomita's algorithm* which is known as one of the most efficient *generalized LR (GLR) parsers*. Empirically, Tomita's algorithm is faster than Earley's algorithm. In this section we will give a brief introduction of Tomita's algorithm with an example and at the end we will mention some of the problems.

GLR parsing algorithm uses a stack and an *LR table* constructed from general CFG to guide the parsing process. Because of one word look-ahead, GLR parsing algorithm inherits the merits of original LR parsing algorithm. Furthermore GLR parsing algorithm will be able to avoid applications of similar Predictor operations as in Earley's algorithm mentioned in the end of section 2. In short, all the necessary Predictor operations are compiled into a LR table in advance. This will be clearly understood if the reader is familiar with the method to generate a LR table from given CFG. However, in this paper, due to the shortage of space, we skip LR table construction algorithm. The reader who wants to know the LR table construction algorith may refer to [Aho, 86]. An example of a LR table constructed from fig.1-1 is shown in fig.4-1 which consists of two fields, an *Action field* and a *Goto field*.

State	Action field					Goto field			
	det	n	v	p	$	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 4-1 : A LR table for the grammar in fig.1-1

GLR parsing algorithm works with a stack called *graph-structured stack (GSS)* and a LR table by pushing or popping a pair of vertices each of which corresponds to a partially parsed tree and a state.

A GSS is initialized by pushing a state 0 in stage U_0, which becomes the root vertex of the GSS with the first word w_1 as a look-ahead word (scanning word), whose preterminal will be used to determine the parsing actions in the LR table. An input sentence is parsed stage by stage for each word from left to right thus changing the GSS.

The readers not familiar with GLR parsing may skip the following definitions and proceed to a GLR parsing example given later in this section.

Upon scanning the i+1st word w_{i+1} as a look-ahead word, the parsing algorithm carries out the following four actions in stage U_i after pushing i-th word.

1. *Reduce* : The parser pops twice the number of vertices (corresponding to the right hand side (rhs) of the production rule specified by the reduce action) from the top of the stack and then pushes two vertices in the stage U_i, which are a partially parsed tree (created by the rule specified by the reduce action) and the state determined by the Goto field of LR table.

2. *Shift* : A pair of vertices corresponding to a look-ahead word w_{i+1} and a state is pushed in U_{i+1} with a look-ahead terminal w_{i+2}. The state pushed on the top of the stack is determined by the shift action. Note that the newly created state vertex in U_{i+1} is not active until there is no active vertex [3]remained unprocessing in U_i.

3. *Error* : The vertex with error action will be truncated.

4. *Accept* : Parsing process will end with success.

Only after every vertex in the stage U_i has been processed, the parser proceeds to the stage U_{i+1} scanning the next word w_{i+2}. What kind of actions (shift; reduce; accept; error) are to be carried out is determined by the top vertices in U_i, LR table, and the preterminal of the scanning word w_{i+1}.

In case of 1 and 2, two vertices are pushed in U_i and U_{i+1} respectively and edges are formed from the new vertex to its parents. If there exists a top vertex with the same state as that of a newly created top vertex in U_i in case of 1, or U_{i+1} in case of 2, then they will be merged into one. The vertex after merge will have several parents. Merging vertices with the same state prevents the duplicated processing of the input sentence in the later stage.

As mentioned before, a LR parsing table consists of two fields, an Action field and a Goto field. The parsing actions are determined by state (the row of the table) and a look-ahead preterminal (the column of the table). Here, $ represents the end of an input sentence. Some entries in the LR table contain more than one operation and are thus in conflict. In such cases, GLR parser must conduct more than one operation simultaneously.

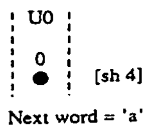
The 'shN' in some entries of the Action field in LR table indicates that the GLR parser has to push a look-ahead preterminal on the LR stack and shift to 'state N'. The symbol 'reN' indicates that the parser has to pop twice the number of vertices (corresponding to right hand side of the rule numbered 'N') from the top of the stack and then go to the new state determined by Goto field. The symbol 'acc' completes parsing successfully. If an entry contains no operation, the parser will detect an error. The LR table in fig.4-1 has conflicts in state 11 and 12 for column 'p'. Each of the two conflicts contains both a shift and a reduce action, which is called a shift/reduce conflict. When GLR parser encounters a conflict, both actions are carried out simultaneously, but the vertex created by the shift action remains inactive until no active vertex remains to be performed.

Let us consider the grammar in fig.1-1 and an input sentence "I saw a man in the park". GLR parsing will proceed as follows. ○ and □ represents a state and a partially parsed tree respectively. In the example, partially parsed trees are designated by unique pointer. The black circles represent an active state.

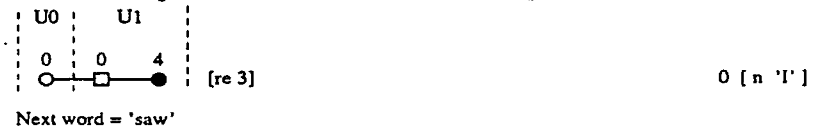
At the beginning, the GSS contains only one vertex with state 0 in the stage U_0

[3]"Active vertex" means there are some action(s) on the top of the stack.

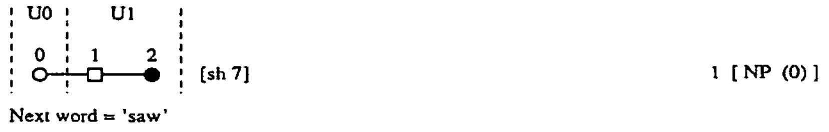
(called state vertex) and no partially parsed trees . By looking at the action table, the next action "shift 4 [sh 4]" is determined from the LR table given in fig. 4-1.



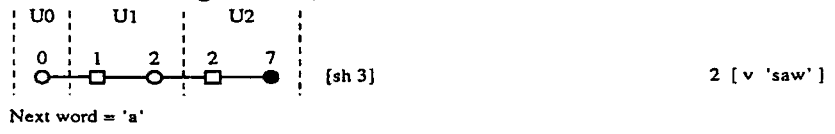
On shifting the word "I", a partially parsed tree ([n, "I"]) corresponding to the shift action is created and is pushed onto the GSS (called pointer vertex) along with the state vertex with state 4. The parser enters into the stage U_1. The next action "reduce 3 [re 3]" is determined from the action part of the LR table, because the state at the top of the stack is 4 and the preterminal of the next word is "v".



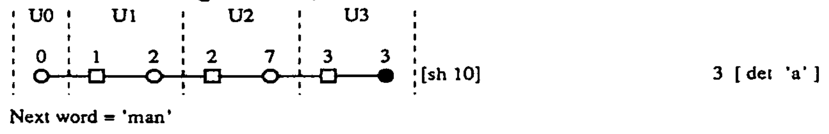
During a reduce action, two vertices from the top of GSS are poped and temporarily the state of the top of GSS becomes 0, which will determines the next state by referring Goto field. A new partially parsed tree pointed by 1 is created in the parse forest. Then the partially parsed tree is pushed onto the GSS along with the state vertex determined by Goto field of 0 and NP.



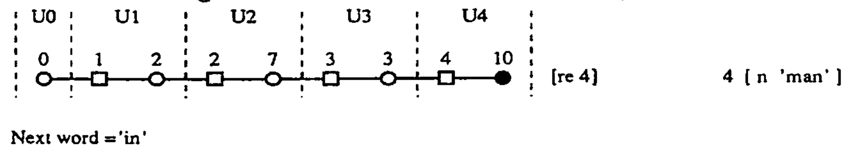
After executing shift 7, the GSS will become,



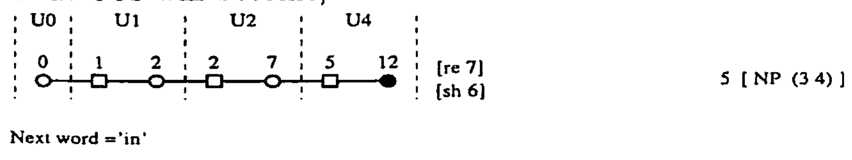
After executing shift 3, the GSS will become,



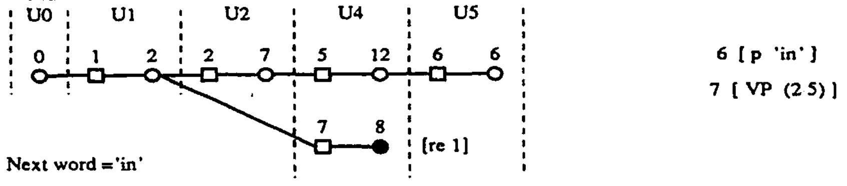
After executing shift 10 the GSS will become,

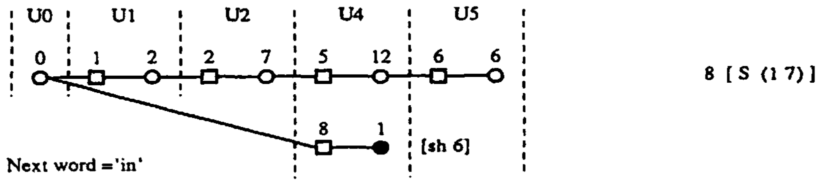


The next action reduce 4 is carried in a similar way as explained before and the resultant GSS will become,



At this point, a conflict with reduce 7 and shift 6 occurs and both should be executed. In case the shift action is executed first, then a pointer vertex and a state vertex corresponding to that shift action will be pushed onto the GSS of stage U_5 and it will be inactive until there is no reduce action remains to be performed in stage U_4. Accordingly, in the GSS below, the vertex with state 8 is still active and the action reduce 1 is executed. The resultant GSS is shown below.

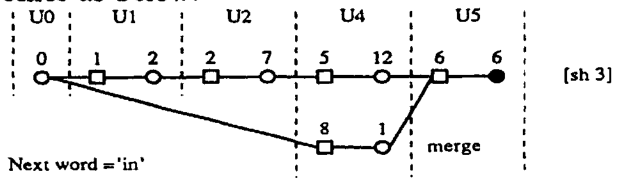




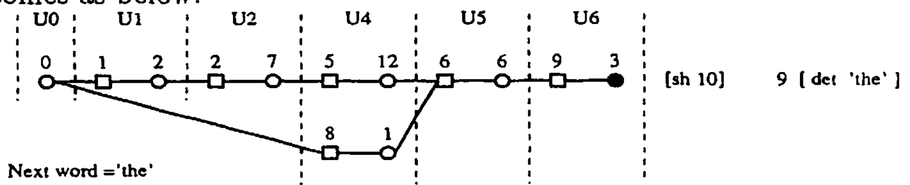
The shift action, shift 6 associated with the active vertex is executed.

Actually a partially parsed tree and a state vertex corresponding to the shift action will be pushed onto the GSS. But, before pushing, the merge action checks whether there exists a same inactive state vertex which might have been pushed by the same shift action.

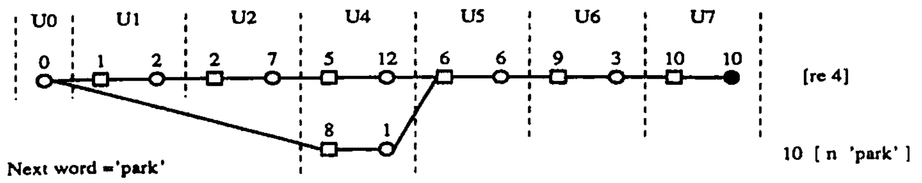
In the GSS above, there already exists a same state vertex in U_5 and because both are shifting same word 'in' as the same preterminal 'p' and the resultant GSS becomes as below.



Now since there is no reduce action remains to be performed in stage U_4, the state vertex 6 in U_5 becomes active. After carrying the next action shift 3, GSS becomes as below.

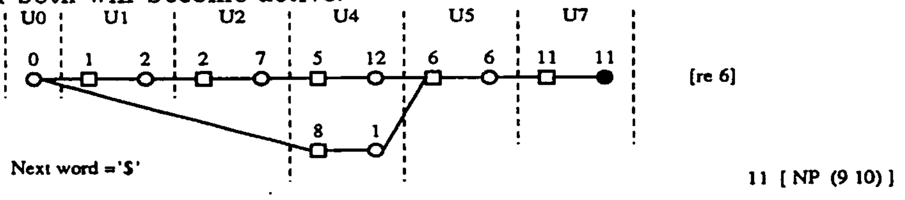


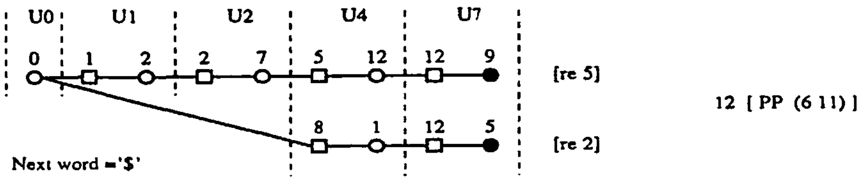
After executing shift 10 the GSS will become,



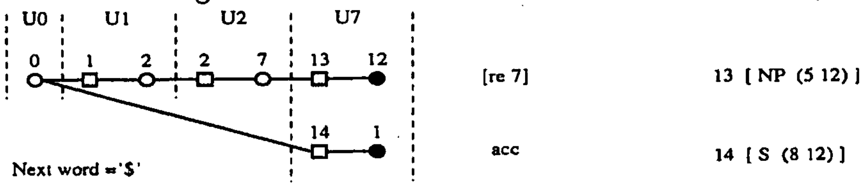
After executing reduce 4 the GSS will become,

In executing reduce 6, after popping the vertices, we have two top state vertices and both will become active.

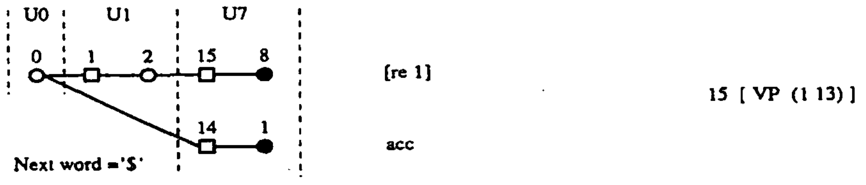


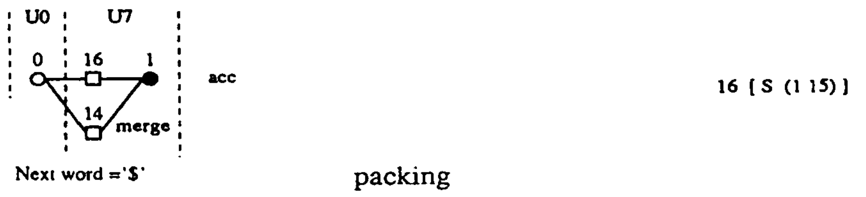


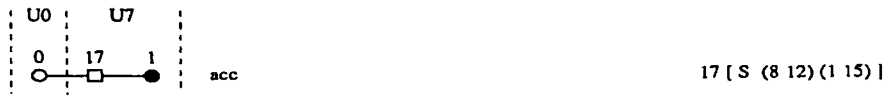
After executing reduce 5 and reduce 2 the GSS will become,



The reader will be able to trace the following GLR parsing process.







The above trace shows all the actions carried out by the GLR parser, such as shift, reduce, merge, pack to create *packed forest.* Note that in case of having more than one preterminal for a word, all vertices corresponding to all of its preterminals will be shifted.

The Tomita's GLR parsing algorithm mentioned above is called *simplified LR (SLR)* parsing algorithm. There are other GLR parsing algorithms called *canonical LR (CLR)* and *LALR* parsing algorithm. CLR parsing algorithm is slightly efficient over SLR but, consumes more space for the LR table. LALR parsing algorithm uses more compact LR table than the one of CLR. Although these two parsing algorithms use a slightly different LR table, GLR parsing algorithm explained above remains the same in principle. Interested reader should refer to [Aho, 86] for CLR and LALR parsing algorithms.

We would like to consider some improvements over Tomita's GLR parsing algorithm. The time complexity of Tomita's GLR parsing algorithm becomes in the order of $n^{\rho+1}$ where ρ is the length of the longest production in the grammar, and n is the length of the input sentence. If the given grammar is in the Chomsky normal form, then the time complexity will become in the order of n^3. According to [Shann, 91], the efficiency of Tomita's GLR over chart parser comes from the packed forest representation, and according to Kipps [Kipps, 89], inefficiency of the GLR parser is in its duplicated traversals of GSS during reduce actions. Kipps invented a GLR algorithm with the time complexity of n^3 by preventing this inefficiency, but his algorithm is considered to be only a recognizer. Tanaka and Suresh find out a family of GLR parsing algorithms called *AGLR*, which are based on Kipps recognizer but can produce parsing results. The time complexity of their algorithm is in the order of n^3 and be proven to be empirically faster than Tomita's algorithm [Tanaka, 92]. Careful experiments reveal that Chart parsing is about 1.5 times slower than Tomita's parsing [Shann, 91], and Tomitas' parsing is about 1.5 times slower than AGLR [Tanaka, 92].

GLR parser is also used in speech recognition researches [Tomita, 86], [Kita, 89]. In this case, phonemes correspond to preterminals of LR table and a look-ahead preterminal/phoneme becomes an expected phoneme to be examined next in the results obtained from acoustic data. In order to do so, LR table will be retrieved in reverse way of parsing. The state of the top of GSS determines a row of LR table in which there are several actions. From these actions we can get corresponding preterminals/phonemes which are used to predict next phoneme, and thus enable to limit the search space.

All of these algorithms mentioned above are based on bread-first parsing algorithms and are easy to be implemented in parallel. But in this paper we would like to consider about the time complexity of them only in sequential version as summerized in figure 4-2, where n is the length of an input sentence, and ρ (≥ 2) is the length of the longest rhs of production rules used in the CFG.

	Earley's Algorithm	Chart Algorithm	GLR Algorithm	
			Tomita	AGLR
Parsing	n^3	n^3	$n^{\rho+1}$	n^3
Tree generation	n^2	n^2	n	n

Figure 4-2 : Complexity Table

The problem of GLR parsing algorithm is to consume a lot of memory space if the size of the CFG becomes large. It consumes $c^{|G|}$ where c is a constant and $|G|$ is the size of the grammar. However, empirically it will not be a big problem in the future, because of the recent rapid developments in memory integration technology.

5 Scoring parsing result by probabilistic CFG

Generally speaking, CFG parsing algorithms explained above might produce many parsing results, out of which one wants to extract most plausible parsing results for semantic processing. Recently, *probabilistic CFG (PCFG)* has been paid much attention to ranking parsing results using scores based on statistical information.

PCFG is a set of rewriting rules with probability such as $< \mathrm{A} \rightarrow \alpha, \mathrm{p} >$, where p is a probability associated with the rule. In PCFG, the following constraint must be hold: The summation of probabilities of all productions with A on its lhs is equal to 1.

Figure 5-1 shows an example of PCFG.

$$\begin{array}{llll}
(1) & \mathrm{S} & \rightarrow & \mathrm{NP\ VP,\ p1} \\
(2) & \mathrm{S} & \rightarrow & \mathrm{S\ PP,\ p2} \\
(3) & \mathrm{NP} & \rightarrow & \mathrm{n,\ p3} \\
(4) & \mathrm{NP} & \rightarrow & \mathrm{det\ n,\ p4} \\
(5) & \mathrm{NP} & \rightarrow & \mathrm{NP\ PP,\ p5} \\
(6) & \mathrm{PP} & \rightarrow & \mathrm{p\ NP,\ p6} \\
(7) & \mathrm{VP} & \rightarrow & \mathrm{v\ NP,\ p7}
\end{array}$$

Figure 5-1: A sample of English PCFG

By the constraints of PCFG, p1 + p2 = 1, p3 + p4 + p5 = 1, p6 = 1, p7 = 1 in fig. 5-1.

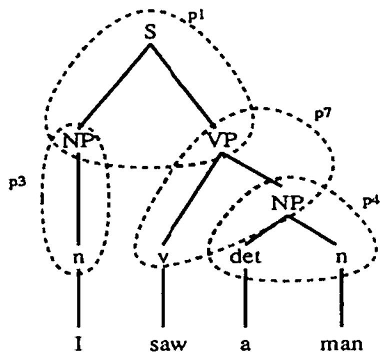


Figure 5-2 : Probability of a parsing tree: p1 × p3 × p7 × p4.

Resulting probability of the parsing tree in fig.5-2 becomes, p1 × p3 × p7 × p4. For each parsing results, we can calculate scores in a similar way and the tree with the highest score will be the most probable one.

In order to assign probability to each rule of PCFG, we have to use a set of plausible parsing trees obtained from a text corpus. Then we count the occurrences of each rule in the plausible parsing trees. After the normalization, we can attach a probability to each rule of PCFG. However, it is very difficult to get a set of plausible parsing results from a large volume of text corpus. Lari and Young [Lari, 90] give us an estimation method of the probability from raw text corpus, which is called Inside-Outside method. Note that the Inside-Outside method does not exclude the use of text corpus with plausible parsing results from which Pereira et al., efficiently generate PCFG [Pereira, 92].

On using GLR to calculate the resulting probability of a parsing tree given in fig. 5-2, it forces us to delay the calculation of probability until a reduce action is executed. However, Wright [Wright, 90] proposed a method of GLR parsing of PCFG which uses a *probabilistic LR parsing table* in which each action is associated with a probability. Using the action probabilities in the probabilistic LR parsing table, the probability of a partially parsed tree is calculated whenever an action is performed. Thus even in a shift action the probability is calculated. The resulting probability becomes the same as the one mentioned before in the beginning of this section. The key idea proposed by Wright is to give a method of distributing a probability in a PCFG rule to shift and/or reduce actions in a LR table.

However, we have a problem because they use only PCFG whose probability attached remains the same during parsing process. Consider the CFG rule as NP $\rightarrow$ pronoun. This rule seems to be more frequently used in the subject position than in the object position in English. Thus this rule should have higher probability in the subject position than object position. But in PCFG this context sensitiveness is not able to express [Magerman 91]. More researches will be necessary.

6 Conclusion & Discussion

There are still many interesting issues in the field of natural language parsing not mentioned above. In this section, we will discuss some of them briefly.

One of the most active issues is parallel parsing. Exploiting parallel characteristic of independent tasks, a parallel parser using a collection of processors can achieve substantial speed-up over a traditional serial parser. There are a variety of methods developed from different points of view. Some of them explore existing (serial) algorithms and try to adapt them to parallel parsing on certain hardwares or programming languages [Chiang & Fu 84][Matsumoto 87,89][Tanaka and Numazaki 89]. On the other hands, other methods design new hardwares to accommodate certain algorithms. There are also some researchers whose their main interest in parallel processing is not to improve the performance of parsers but to simulate human language processing with connectionist networks [Fanty 85] [Jain and Waible 91]. More references to papers on parallel parsing can be found in [Nijholt 91].

Parsing free word order languages is another interesting issue. Several languages, e.g. Walpiri, Japanese, German and Thai, exhibit significant word order variation. Using CFG to describe such languages is cumbersome because a numerous number of rules would be needed to enumerate all possible configurations of a constituent. Generalized Phrase Structure Grammars(GPSG) [Gazdar et al. 85] provides a solution to this problem by decomposing the grammar rules into Immediate Dominance (ID) rules and Linear Precedence(LP) rules. Using ID/LP grammars, free word order languages can be easily and concisely described. However, how to combine the separate constraints together in parsing becomes a problem. One obvious way is to compile an ID/LP grammar into a CFG which can then be parsed with any existing CFG parsing algorithm. However, this approach yields a huge object grammar which can degrade the parsing performace. Shieber [Shieber 84] presents an ID/LP *direct* parsing algorithm by modifying Earley's algorithm to use ID/LP grammars directly. Meknavin et al. [Meknavin et al. 92] improves Shieber's algorithm by compiling ID and LP rules into generalized discrimination networks and Hasse diagrams, and using bottom-up chart algorithm instead of Earley's. Another top-down approach of direct ID/LP parsing is presented in [Abramson and Dahl 92].

As true natural language usage can be filled with many types of errors, in order to achieve a practical interactive natural language understanding system we have to provide methods for the system to recover from those errors reasonably. The system should attempt to understand what users intend to say like the way human do when they communicate to each other. Many algorithms for parsing ill-formed input are based on using chart [Mellish 89],[Kato 91],[Meknavin 93]. Some LR-based algorithms are employed in speech recognition systems[Saito and Tomita 88]. Many other approaches which interested readers should refer to can be found in [Allen(ed.) 83].

All parsing algorithms mentioned in this paper are concerned with only CFG. As described earlier, however, there are some languages which cannot be described by CFG, called context-sensitive languages. The attempt to handle such languages have led to inventions of various grammar formalisms. Since coping with the class of fully context-sensitive grammars is difficult and may not be required in practical natural language processing because of their extraneous power, some are concerned with only so called "mildly context-sensitive grammars"(tree adjoining grammars(TAGs) [Joshi 85] et. al.). Lexical functional grammars(LFGs) [Kaplan and Bresnan 83], on the other hand, have much more power than CFGs and TAGs. Designing parsing algorithms for all of these grammars is another interesting field of researches[Crain and Fodor 85].

Finally, we would like to mention about the relation between logic programming language and natural langauge parsing. Pereira et.al, discussed about the advantages of using logic programming language such as Prolog for natural language processing. Interested reader can refer to [Pereira, 80], [Matsumoto, 83], [Pereira, 87] for top-down and bottom-up parsing in logic programming. Handling of long dependency is another difficult task in parsing, but most interested works have been based on the logic programming. Interested reader should refer to [Pereira, 81], [Dahl, 84], [Crain, 85], [Tokunaga, 88] and [Alshawi, 92].

References

1. Abramson H. and Dahl V. (1992) *On Top-down ID-LP Parsing with Logic Grammars*, submitted for publication.

2. Aho A.V. and Ullman J.D. (1972) *The Theory of Parsing, Translation and Compiling*, Vol.1, Prentice-Hall.

3. Alshawi H. ed (1992) *Core Language Engine*, The MIT Press.

4. Allen J. F. (1983) *Special Issue on Ill-formed Input*, American Journal of Computational Linguistics, Vol.9, No.3-4.

5. Aho A.V., Ravi Sethi. and Ullman J.D. (1986) *Compilers, Principle, Techniques, and Tools*, Addision Wesely.

6. Chiang Y. T. and Fu K. S. (1984) *Parsing Algorithms and VLSI Implementation for Syntactic Pattern Recognition*, IEEE transactions on pattern analysis and machine intelligence, PAMI-6, 3, pp. 302-314.

7. Crain S. and Fodor J. D. (1985) *How Can Grammars Help Parsers?* , in Dowty, D. R. et. al.(Ed.), Natural language parsing, pp. 94-128, Cambridge University Press.

8. Dhal V. (1984) *On Gapping Grammars*, Proc. of 2nd International conf. on Logic programming, pp.77-88, Uppsala, Sweden.

9. Dowty D. R. et. al. (1985) *Natural Language Parsing*, Cambridge University Press.

10. Earley J. (1970) *An Efficient Context-free Parsing Algorithm*, Comm. of the ACM, 13(2) pp.95-102.

11. Fanty, M. A. (1985) *Context-free Parsing in Connectionist Networks* (Tech. Rep. No. TR 714), Computer Science Department, University of Rochester.

12. Fujisaki T, et al. (1991) *A Probabilistic Parsing Method for Sentence Disambiguation*, Current issues in parsing technology, Tomita, M.Ed, pp-139-152, Kluwer Academic publishers.

13. Gazdar G., Klein E., Pullum G.K. and Sag I.A. (1985) *Generalized Phrase Structure Grammar*, Basil Blackwell pub.

14. Jain, A. N. and Waible, A. H. (1991) *Parsing with Connectionist Networks*, In Tomita, M.(Ed.), Current issues in parsing technologies, Kluwer Academic Publishers.

15. Joshi A. K. (1985) *Tree adjoining Grammars: How Much Context-sensitivity Is Required to Provide Reasonable Structural Descriptions?*, in Dowty, D. R. et. al.(Ed.), Natural language parsing, pp. 206-250, Cambridge University Press.

16. Kaplan R. and Bresnan J. W. (1983) *Lexical Functional Grammar - A Formal System for Grammatical Representation*, in Joan Bresnan(ed.), The mental representation of grammatical relations, Cambridge, Mass, MIT Press.

17. Kato T. (1991) *Yet Another Chart-Based Technique for Parsing Ill-formed Input*, Technical report on natural language processing 83-10, Information processing society of Japan, pp. 71-78. (in Japanese)

18. Kay M. (1980) *Algorithm Schemata and Data Structures in Syntactic Processing*, CSL-80-12 Xerox Palo Alto research center.

19. Kita K, Kawabata T. and Saito H. (1989) *HMM Continuous Speech Recognition Using Predictive LR Parsing*, IEEE, Proceedings of ICASSP-89, S13.3.

20. Knuth D.E. (1965) *On the Translation of Languages Left to Right*, Information and control, 8(6), pp.607-639.

21. Lari K. and Young S.J. (1990) *The Estimation of Stochastic Context-free Grammars Using the Inside-Outside Algorithm*, Computer speech and languages, 4, 35-56.

22. Magerman D.M. and Marcur, M. (1991) *Pearl: A Probabilistic Chart Parser*, Prof. of EACL'91, pp. 15-20.

23. Matsumoto Y.et.al. (1983) *BUP-A Bottom-up Parser Embedded in Prolog*, New generation computing, 1, 2, pp. 145-158.

24. Matsumoto Y. (1988) *Natural Language Parsing Systems Based on Logic Programming*, Dotor thesis of Kyoto university, Kyoto, Japan.

25. Meknavin S. (1993) *A Method of ID/LP Parsing Using Generalized Discrimination Networks*, Doctoral thesis, Dept. of computer sc., Tokyo institute of technology.

26. Meknavin S. Okumura M. and Tanaka H. (1992) *A Chart-based Method of ID/LP Parsing with Generalized Discrimination Networks*, COLING'92, Vol.1, pp.401-407.

27. Mellish C. S. (1989) *Some Chart-Based Techniques for Parsing Ill-Formed Input*, in 27th ACL, pp. 102-109.

28. Nijholt A. (1991) *Overview of Parallel Parsing Strategies*, in Tomita, M.(Ed.), Current issues in parsing technologies, Kluwer Academic publishers.

29. Pereira F. C. N. and Warren. D. H. D. (1980) *Definite Clause Grammar*, Artificial intelligence, 13, pp. 231-278.

30. Pereira F. C. N. (1981) *Extraposition Grammar*, ACL, 7, 4, pp.243-256.

31. Pereira F. C. N and Shieber S. M. (1987) *Prolog and Natural Language Analysis*, CSLI, Stanford university.

32. Pereira F. C. N. (1992) *Inside-Outside Reestimation from Partially Bracketed Corpora*, Proc. of ACL'92, pp.128-135.

33. Saito H. and Tomita M. (1988) *Parsing Noisy Sentences*, Proc. of COLING'88, pp. 561-566.

34. Shann P. (1991) *Experiments with GLR and Chart Parsing*, in Tomita, M ed., Generalized LR parsing, Kluwer Academic publishers, pp. 17-34.

35. Shieber S. M. (1984) *Direct Parsing of ID/LP Grammars*, Linguistics and Philisophy, Vol. 7, pp. 135-154.

36. Tanaka H., Suresh K.G. and Yamada K. (1992) *A Family of Generalized LR Parsing Algorithms Using Ancestors Table*, in Technical report, 92TR-0019, Department of Computer Science, Tokyo institute of technology.

37. Tanaka H. and Numazaki H. (1989) *Parallel Generalized LR Parsing Based on Logic Programming*, International workshop on parsing technologies, pp. 329-338.

38. Tokunaga T. Iwayama M. Tanaka H. and Kamiwaki T. (1988) *LangLAB: A Natural Language Analysis System*, Proc. of COLING'88, pp. 655-660.

39. Tomita M. (1986) *An Efficient Parsing Algorithm for Natural Language*, Kluwer, Boston, Mass.

40. Tomita M. (1986) *An Efficient Word Order Lattice Parsing Algorithm for Continuous Speech Recognition*, in proceedings of IEEE-IECE-ASJ international conference on acoustics, speech, and signal processing.

41. Wetherell C.S. (1980) *Probabilistic Languages: A Review and Some Open Questions*, Computing surveys, Vol.12, No.4, pp.361-379.

42. Winograd T. (1983) *Language as a Coginitive Process*, Vol. 1, Addison Wesley.

43. Wright J.H. (1990) *LR Parsing of Probabilistic Grammars with Input Uncertainty for Speech Recognition*, Computer speech and language, 4, pp. 297-323.