Proceedings of ROCLING IV
R. O. C. Computational Linguistics Conference
(中華民國八十年
第四屆計算語言學研討會論文集)

YAGLR : Yet Another Generalized LR Parser

Hozumi TANAKA K.G. SURESH

Department of Computer Science
Tokyo Institute of Technology
2-21-1 Oookayama Meguro-ku Tokyo 152, Japan
Email : tanaka@cs.titech.ac.jp

Topic area : Syntax (Parsing), Natural Language Processing

Date : 12/07/1991

(Draft)

Abstract

We have developed a new generalized LR parsing algorithm called YAGLR. The parsing algorithm uses graph-structure stacks similar to Tomita's algorithm, but the merge operations on the graph-structure stack is more deeper. Since YAGLR has an effective merge operations, the speed in parsing time and the reduction in memory space are remarkable. Because of YAGLR's effective merge operations, even if tree-structure stacks are used in YAGLR instead of graph-structure stack, this will not cause heavy loss of memory space and still the compactness and reduction of memory space is retained. The realization of tree-structure stack is more simpler than graph-structure stack which is one of the important factor for the implimentation of YAGLR. YAGLR's parsing time is in the order of $\Omega(n^3)$ for any CFG. We also present the experimental results which supports this fact. We conclude that YAGLR has advantages of both Earley's and Tomita's algorithm.

1 Introduction

Some compilers of programming languages have made use of the LR(k) parsing algorithm devised by Kunth [Kunth 65] which enables us to parse an input sentence deterministically and efficiently. But the grammars used in this algorithm is limited to LR(k) grammar so that Context Free Grammars (CFG) in general can not be handled.

Tomita extended the LR(k) parsing algorithm to handle CFG [Tomita 86,87]. The extended algorithm is one of the generalized LR parsing algorithms. Empirical results of Tomita's and Earley's algorithm reveal that the Earley/Tomita ratio of parsing time is larger when the length of an input sentence is shorter or when the input sentence is less ambiguous [Tomita 86]. It has been shown by [Johnson 89] and [Kipps 89] that for some CFG Tomita's algorithm dosen't fare well compared to Earley's algorithm.

In this paper we propose a new generalized LR parsing algorithm called YAGLR (Yet Another Generalized LR parsing) which has advantages of both Earley's and Tomita's algorithms. YAGLR also uses graph-structure stack (GSS) similar to Tomita's but the memory space used by YAGLR's GSS is more compact than that of Tomita's. Even if tree-structure stacks are used in YAGLR instead of GSS, this will not cause heavy loss of memory space and will remain almost the same as GSS. The realization of tree-structure stack is more simpler than GSS which is one of the important factor in the implimentation of YAGLR. The reason for memory compactness in YAGLR is that the merge operations of stacks in YAGLR is more deeper and effective. Further YAGLR's parsing time is in the order of $\Omega(n^3)$ for any CFG. Our experimental results also supports this fact.

Tomita's algorithm creates partially parsed trees during parsing process whereas YAGLR creates items called as *dot reverse items* (*dril*) which are different from Earley's items. These *drils* make not only effective merge operations possible, but also ease the removal of duplicated items. In section 2 we will explain about generalized LR parsing (Readers familiar with generalized LR parsing can skip this section). In section 3 we introduce *dril* by comparing with Earley's items and discuss the factor of forming *drils* instead of Earley's items. Section 4 gives the formal definition of *drils* used in YAGLR. Section 5 gives a parsing algorithm of YAGLR along with the merge procedures of GSS. In section 6, we will give some of the experimental results and its related discussions. In section 7, future research themes will be discussed.

2 Brief Introduction to Generalized LR Parsing Algorithm

The generalized LR parsing algorithm uses LR stacks and an LR parsing table generated from predetermined grammar rules. An ambigous English grammar and its LR parsing table are shown in figure 1 and 2, respectively [Tomita 87].

(1)	S	→	NP,VP
(2)	S	→	S,PP
(3)	NP	→	n
(4)	NP	→	det,n
(5)	NP	→	NP,PP
(6)	PP	→	p,NP
(7)	VP	→	v,NP

Figure 1

State	det	n	p	v	$	S	NP	PP	VP
0	sh1	sh2				3	4		
1		sh5							
2			re3	re3	re3				
3			sh6		acc			7	
4			sh6	sh8				10	9
5			re4	re4	re4				
6	sh1	sh2					11		
7			re2		re2				
8	sh1	sh2					12		
9			re1		re1				
10			re5	re5	re5				
11			sh6/re6	re6	re6			10	
12			sh6/re7		re7			10	

Figure 2

The parsing table consists of two fields, a parsing action field *action* and a goto field *goto*. The parsing actions are determined by state (the row of the table) and a look-ahead preterminal (the coloumn of the table) of an input sentence. Here, $ represents the end of the sentence. There are two kinds of stack operations: shift and reduce. Some entries in the LR table contain more than two operations and are thus in conflict. In such cases, a parser must conduct more than two operations simultaneously.

The 'sh N' in some entries of the LR table indicates that the generalized LR parser has to push a look-ahead preterminal on the LR stack and goto 'state N'. The symbol 're N' denotes that the parser has to pop the number of elements (corresponding to right hand side of the rule numbered 'N') from the top of the stack and then goto the new state determined by goto field. The symbol 'acc' means that the parser has successfully completed parsing. If an entry contains no operation, the parser will detect an error.

The LR table in figure 2 has conflicts in state 11 and 12 for coloumn 'p'. Each of the two conflict contains both a shift and a reduce operation and is called a shift/reduce conflict. When our parser enconuters the conflict, all shift actions shoule be carried out after all reduce actions are completed.

On input "i saw a girl with a telescope", the sequence of stack and input contents is shown in Fig.3. For example, at line (1) the parser is in state 0 with "i" the first input symbol. The action in row 0 and column n (the grammatical catagory of "i") of the action field of Fig.2 is sh2, meaning shift and cover the stack with state 2. That is what was happened in line (2): the first grammatical category n and the state symbol 2 have both been pushed onto the stack.

Then, "saw" becomes the current input symbol, and the action of state 2 on v (the grammatical catagory of "saw") is to reduce by NP → n. One state symbol and one grammar symbol are poped from the stack and 0 again becomes the top of the stack. Since the goto of state 0 on NP is 4, NP and 4 are pushed onto the stack. We now have the configuration in line (3). Each of the remaining moves are determined similarly until the shift of "girl".

In line (8), we get a conflict with sh6/re7, because for the word "with" whose preterminal is p and the state on the top is 12. Here we carry out re7 at first and we see that the stack on which sh6 to be performed is waited until all the other remaining stacks experiences shift actions. At line (10) we do the shift action and merge the stacks because both stacks have the same top state. The remaining parsing proceeds in this way.

No.	Stack	Input	Actions
(1)	0	I saw the girl with a tele$	shift
(2)	0 n 2	saw the girl with a tele$	reduce by NP→n
(3)	0 NP 4	saw the girl with a tele$	shift
(4)	0 NP 4 v 8	the girl with a tele$	shift
(5)	0 NP 4 v 8 det 1	girl with a tele$	shift
(6)	0 NP 4 v 8 det 1 n 5	with a tele$	reduce by NP→det,n
(7)	0 NP 4 v 8 NP 12	with a tele$	reduce by VP→v,NP shift
(8)	0 NP 4 [v 8 NP 12 VP 9]	with a tele$	*shift reduce by S→NP,VP
(9)	0 [NP 4 v 8 NP 12 S 3]	with a tele$	*shift shift
(10)	0 [NP 4 v 8 NP 12 S 3] p 6	a tele$	
- -	- - - - - -	- - - - - -	- - - - - -

Figure 3. An Example of Generalized LR Parsing

3 Earley's Item and Dot Reverse Item

In YAGLR method, we are not using terminal and non-teriminal symbols along with state number as in Generalized LR parsing algorithm shown in fig.3 . But instead we use position numbers along with state number. The position number indicates the position upto which the shift of an input sentence has been completed. Since all the actions are carried out using state number, the actions of every nodes in the stack will be the same if we use either position number or grammar catagory along with state number. In Tomita's method, packed forest representation is used instead of grammatical catagories and partially parsed trees are produced.

During reduce actions, YAGLR creates *drits*. These *drits* are different from Earley's items. Instead of a partially parsed tree, we attach a set of position numbers to each node and we can create either Earley's items or *drits* during reduce actions. In this section, we would like to consider the factors of forming *drits* instead of Earley's items.

Let us consider the following stack with a reduce action 're,x' and an input sentence $w_1 w_2 \cdots w_n$.

(a) $\cdots$-[{3}, S3]—[{5}, S2]—[{6}, S1] (Top) re,x

Here, S1, S2, S3 indicates states and 3, 5, 6 indicates position numbers. The position number i is location between w_i and w_{i+1}. The node [{6}, S1] in (a) covers the input word w_6, the node [{5}, S2] covers the input word w_4 to w_5 and so on.

Now, assume that the rule 'x' in the reduce action is A $\rightarrow$ B C, then two nodes from the top must be popped and the Earley's items shown in (b) are formed.

(b) Earley's items :

$I_5 \ni [A \rightarrow B \cdot C, 3]$
$I_6 \ni [A \rightarrow B\ C \cdot, 3]$

In the items in (b), number 3 inside the item is the position number, starting from which Earley's items are formed. This indicates that the input sentence from position 3 to 5 in the first item has been recognized as "B". In this way the input sentence from position 3 to 6 in the second item has been recognized as "B C" and combined as "A" by applying rule 'x'.

Let us see what will happen if we form the items starting from position number 6 and ending with 3 in the reverse order using the rule 'x'.

(c) *drits* :

$I_5 \ni [A \rightarrow B \cdot C, 6]$
$I_3 \ni [A \rightarrow \cdot B\ C, 6]$

In case of (c), number 6 inside the item is the position number appeared in the top node of the stack (a)[1]. These items are formed by considering the dot positions from right to left which is in reverse direction compared to Earley's items. Hence we call them as dot reverse items (*drits*), the definition of which will be given in the next section. Here, in the first *drit*, the input sentence from position number 6 down to 5 (w_6) has been recognized as "C" and the input sentence from position 6 down to 3 ($w_4 w_5 w_6$) in the second *drit* has been recognized as "B C" and combined as "A" by applying rule 'x'.

Now let us think of the following case having stack (d) whose top nodes are merged, which is same as Tomita's merge.

(d) $\cdots$-[{3}, S3]—[{5}, S2]—[{6}, S1]┐ re,x
==-[{2}, S4]—[{4}, S2]—[{6}, S1]┘

Using the same rule 'x', through the reduce actions on stack (d), Earley's items and *drits* are formed in (e) and (f) respectively.

(e) Earley's Items :	(f) *drits* :
$I_5 \ni [A \rightarrow B \cdot C, 3]$	$I_5 \ni [A \rightarrow B \cdot C, 6]$
$I_6 \ni [A \rightarrow B\ C \cdot, 3]$	$I_3 \ni [A \rightarrow \cdot B\ C, 6]$
$I_4 \ni [A \rightarrow B \cdot C, 2]$	$I_4 \ni [A \rightarrow B \cdot C, 6]$

[1] This position number 6 will remain the same until the next shift action.

$I_6 \ni [A \rightarrow B\ C \cdot, 2]$ $\qquad$ $I_2 \ni [A \rightarrow \cdot B\ C, 6]$

Since both top nodes of (d) has the same state, let us merge the stack (d) one node deeper to get the stack shown in (g).

```
(g) ····[{3}, S3]—[{5}, S2]┐[{6}, S1] re,x
     ==-[{2}, S4]—[{4}, S2]┘
```

Performing the reduce action 're,x' on (g), the two nodes are popped from the top. The items same as that in (e) and (f) are created from (g).

Since the state S2 of two nodes below the top node [{6}, S1] in (g) are the same, now let us see what will happen if we proceed the merge of stack (g) one more node down as shown in (h).

```
(h) ····[{3}, S3]┐[{4,5}, S2]—[{6}, S1] re,x
     ==-[{2}, S4]┘
```

In (h) we merged the nodes of state S2 with the union of position numbers. On carrying out the reduce action 're,x' on (h), the two nodes [{4,5}, S2] and [{6}, S1] are popped. In addition to Earley's items shown in (e), the following two Earley's items are also created which we don't want to have.

$$I_4 \ni [A \rightarrow B \cdot C, 3]$$
$$I_5 \ni [A \rightarrow B\ C \cdot, 2]$$

However, if we form *drits* for the above merged stack (h) it is again the same as in (f). This means that the creation of *proper drits* is possible from much deeper merged GSS than (g). In other words, the new merge algorithm enables us to make deeper merge of GSS and makes the structure of GSS much simpler. This is one of the important advantages of creating *drits* instead of Earley's items. The reason why the creation of proper *drits* is possible comes from the fact that LR parsing is based on the right-most derivation. Needless to say, Tomita's merge algorithm does not permit such a deep merge operation. The details of our merge algorithm is given in the following sections along with justifications.

Another important fact in using *drits* is the localization of duplication checks. The position number inside Earley's items will change within the processing of a single input word itself, as we see in (e). On the other hand, the position number inside *drits* will remain the same throughout the processing of a single input word and thus it enables us to limit the duplication check within the processing of a single input word. Therefore we can localize and reduce the range of duplication check of *drits*.

4 Dot Reverse Item

While Tomita's algorithm creates partialy parsed trees during reduce actions, YAGLR creates the *drits* which differ from Earley's items. A *drit* is defined as follows.

Let G = (N, T, P, S) be a CFG and let $w = w_1\ w_2\ \ldots.\ w_n \in T^*$ be an input sentence in T^* which is a set of a sequence of terminal symbols. For a CFG rule $A \rightarrow X_1\ \ldots\ X_m$ and $0 \leq j \leq n$, $[A \rightarrow X_1\ X_2\ \ldots\ X_k \cdot X_{k+1}\ \ldots\ X_m, j]$ is called a *drit* for w. The dot between X_k and X_{k+1} is a metasymbol not in N and T. The suffix i in the input sentence is called the *position number* in the following sections. The special position number '0' represents the left hand side position of w_1.

I_i, a set of *drit* is defined as follows. For i and j $(0 \leq i \leq j \leq n)$, $[A \rightarrow \alpha \cdot \beta, j] \in I_i$ iff $S \overset{*}{\Rightarrow} \gamma\ A\ \delta$, $\beta \overset{*}{\Rightarrow} w_{i+1} w_{i+2} \ldots w_j$, and $\delta \overset{*}{\Rightarrow} w_{j+1} w_{j+2} \ldots w_n$ where the dot position is a suffix i of an item set I_i. The sequence of sets $I_0, I_1, \ldots, I_n$ is the parse list for the input sentence w.

The difference of a *drit* with Earley's item lies in the interpretation of j. It is evident from the above definition that, in the *drit*, the analysis has been completed for β which is on the *right* hand side of the dot symbol. On the contrary, in case of Earley's item, the analysis has been completed for α which is on the *left* hand side of the dot symbol.

5 The Method of YAGLR

In this chapter, we will explain the structure of a graph-structured stack (GSS) and merge actions followed by shift and reduce actions. Finally we give the procedure of YAGLR.

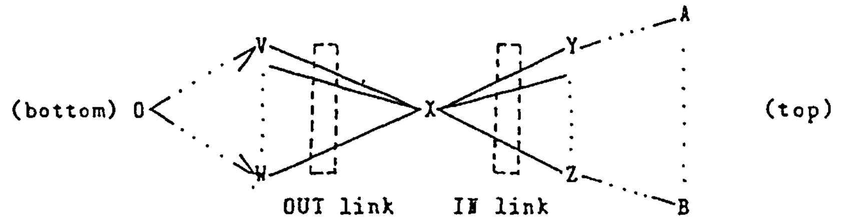
5.1 GSS,Merge, Shift and Reduce Action

5.1.1 Graph-Structured Stack (GSS)

Each node of GSS used in YAGLR has the following structure:

[<a set of position numbers>, <state>].

The set of position numbers are used to produce *drits* during reduce actions. Traversing a path of GSS from top to bottom, we have two adjoining nodes A and B (refer figure below). Concerning to this path, let the node which is nearer to top of the stack be A. Then, A is the parent node of B and B is the child node of A. The node X positioned other than top and bottom of the stack stretches links to the children nodes and the parent nodes. Concerning to node X, the former's links (links to children nodes) are called as OUT links and the later's links (links from parent nodes) are called as IN links. Again concerning to node X, OUT links V, ..., W are all in the same level. The bottom of GSS is mentiond as O in the figure below. In general, there will be several top nodes in GSS, but after merging, the remaining top nodes will be at most no more than the number of distinct states.



5.1.2 Merge of Nodes

In YAGLR, the merge proceeds from top to bottom of GSS. The nodes which are having the same state can only be merged and the merge procedure differes depending on the location of the merging node. To merge two nodes with the same state, we apply the following procedures (M1) and (M2). Applying (M1) and (M2) recursively the merge can be extended to handle more than two nodes having the same states.

(M1) The two top nodes [{i}, s] and [{i}, s] are merged into one node as [{i}, s]. The two top nodes before merge are now become merged into one and their OUT links becomes the OUT links of newly created merged node.

(M2) Let the parent node be X. For the two children nodes [M, s] and [N, s] of X, a new merged node [M ∪ N, s] is formed. The OUT links of [M ∪ N, s] are decided by either (M 21) or (M 22).

(M21) If M (or N) is subset of N (or M) then, the node [N, s] (or [M, s]) will become the merged node and hence, the OUT links of merged node will become the OUT links of the node [N, s] (or [M, s]). In this case, the states of the children nodes attached to OUT links are different from each other and so no more merge is possible.

(M22) If M (or N) is not subset of N (or M), then all OUT links of both [M, s] and [N, s] will become the OUT links of the merged node.

After the explanation of a reduce action, we will give the justification of the above two merge procedures, (M1) and (M2). Since we defined the merge actions considering two nodes, we now give the merge algorithm of GSS as follows.

5.1.3 Merge Algorithm of GSS

(1) If all the top nodes of stack has distinct states then merge process ends. Otherwise, to merge the top nodes of the stack with same states, apply (M1) repeatedly. Then, for every set of children nodes of all the merged top nodes, perform (2).
(2) In a given set of nodes, if there is no node having same state then the merge action ends. Otherwise, apply (M2). In performing (M2) for the two nodes, if (M21) is applied then the childern nodes of the currently merged node has dictinct states. It means that it is not necessary to proceed the merge beyond this level. This is an important fact from the efficiency point of view of our merge operations. If (M22) is applied merge operations will proceed one step deeper.

An Example of Merge on GSS

```
                              /--[···]---[···] (top)
(a) ··· -[{2,3}, 8]<--[{2,4,5}, 9]--[{6}, 1]  (top)
(b) === -[{2,3,4}, 8]--[{3,5}, 9]---[{6}, 1] (top)
```

↓ applying (M1)

```
                              /--[···]---[···] (top)
(c) ··· -[{2,3}, 8]<--[{2,4,5}, 9\
                                   >--[{6}, 1] (top)
    === -[{2,3,4}, 8]--[{3,5}, 9/
```

↓ applying (M22)   { the

```
                              /--[···]---[···] (top)
(d) ··· -[{2,3}, 8]<\
                     >--[{2,4,5}, 9]---[{6}, 1] (top)
    === -[{2,3,4}, 8/
```

↓ applying (M21)

```
                              /--[···]---[···] (top)
(e) ··· -[{2,3}, 8]<\
                     >--[{2,4,5}, 9]---[{6}, 1] (top)
    === -[{2,3,4}, 8/
```

Let us merge the above stacks (a) and (b) using our merge algorithm. Since the top states of both the stacks are the same which is 1, use (M1) and get the partialy merged stack (c). The children nodes of the merged top node in (c) has the same state 9 and one more step deeper merge operation is carried out using (M22) and we get (d). As newley created merged node has two children nodes with the same state 8, carry out merge operations using (M21) since we have the set of position number {2,3} is a subset of {2,3,4}. As all the children nodes below the node [{2,3,4}, 8] are retained as the resultant children node of the merged node, we get (e) (in this case it is the same as (d)) and the merge operation ends here.

5.1.4 Shift action

Let us explain a shift action, "sh, u" to a path of GSS as shown in (a). It shifts (pushes) a new node onto the path of GSS getting (b) and creating a *drit* in I_i as shown in (c). The position number of the shifted node in (b) is increased by one.

(a) $\cdots$ -[M,s]—[{i},t] (top) "sh,u"

(b) $\cdots$-[M,s]—[{i},t]-[{(i+1)},u] (top)

(c) $I_i \ni$ [X $\rightarrow \cdot\ w_{i+1}$, i+1].

5.1.5 Reduce action

Let us consider a reduce action for a path of GSS using a CFG rule having m nonterminal symbols on its RHS:

$$A \rightarrow X_1 X_2 .... X_m$$

After the reduce action for (a), (b) is obtained along with the formation of a set of *drits* as shown in (c).

$$\begin{array}{l}
\text{(a)} \cdots -[P_k, s_k]-[P_{k+1}, s_{k+1}]- \cdots -[P_{k+m}, s_{k+m}](\text{top}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \downarrow \\
\text{(b)} \cdots -[P_k, s_k]-[P'_{k+m}, t](\text{top})
\end{array}$$

Where the state 't' in (b) is a new state determined by GOTO table of both s_k and A (the LHS of a CFG rule used in the reduce action) and

P_k = {a, b,. . .}, P_{k+1} = {c ,d,. . .}, . . . , P_{k+m-1} = {e, f,. . .,g}, P_{k+m} = {i}, P'_{k+m} = {i}.

Note that a set of position number P_{k+m} at the top node of (a) is {i} which includes only one position number of the last input word shif so far and a set of position number P'_{k+m} after the reduce action is {i}. Note that the position number in both P_{k+m} and P'_{k+m} are same.

(c) Formation of *drits* :

$$\begin{array}{lll}
I_a & \ni & [A \rightarrow \cdot X_1 X_2 ... X_m ,i] \\
I_b & \ni & [A \rightarrow \cdot X_1 X_2 ... X_m ,i] \\
 & & .... \\
I_c & \ni & [A \rightarrow X_1 \cdot X_2 ... X_m ,i] \\
I_d & \ni & [A \rightarrow X_1 \cdot X_2 ... X_m ,i] \\
 & & .... \\
I_e & \ni & [A \rightarrow X_1 X_2 ... \cdot X_m ,i] \\
I_f & \ni & [A \rightarrow X_1 X_2 ... \cdot X_m ,i] \\
 & & .... \\
I_g & \ni & [A \rightarrow X_1 X_2 ... \cdot X_m ,i]
\end{array}$$

The position number i inside a *drit* is a position number of the top node in the stack and is remained unchanged until the next shift action occurs. The *drit* such as [A $\rightarrow X_1$, ... , $X_m \cdot$,i] ($\in I_i$) is not produced because they do not contribute the formation of trees.

5.1.6 Justification of the merge of nodes

The justification of (M1) in section 5.1.2 is evident without any arguments. The justification of (M2) is as follows.

1. In YAGLR, a shift action will wait until all reduce actions has been carried out. Because of this, during all the reduce actions the top nodes of all GSS will have the same position number. Therefore all position numbers inside any *drits*, formed by reduce actions before the next shift, should be the same.

2. A suffix of a *drit* set such as i in I_i stands for the position number of a dot in the *drit*. The suffix i is determined by position numbers in the poping nodes by a reduce action as explained in 5.1.5. As two nodes merged are in the same distance from the top node, all position numbers

in these nodes are included in the resultant merged node which includes a union of two sets of position numbers in the two nodes before merge.

3. When M is not a subset of N, it can be clear that the children nodes of both top nodes should be inherited to the merged node and be candidate nodes of the next merge operations. On the other hand, if M is a subset of N, the merged node [M ∪ N, s] is equal to [N, s] which inherits all the subtrees immediately below [N, s].

5.2 Procedure of YAGLR

Let us give a complete algorithm of YAGLR.

1. Set the initial state of GSS as :
   (Bottom) [{0}, 0] (Top)
2. For every node in the top of the stack, select the actions determined by the LR table and do them as described in (3).
3. For actions other than shift: Do (3.1) to (3.4) repeatedly for every top node of the stack and goto (4).
   (3-1) "accept": End with "success"
   (3-2) "error" : End with "failure"
   (3-3) "reduce": Do the reduce action and then with the newley created top nodes goto (2).
   (3-4) "shift/reduce": Do all the reduce actions by (3-3) and then goto 4 with shift actions.
4. Carry out all shift actions, merge GSS and then goto (2)

6 Experimental Results

In this section we give the preliminary experimental results comparing with SAX [Matsumoto, 88] and SGLR [Numazaki, 91]. SAX is based on the bottom up version of Chart algorithm and SGLR is based on Tomita's algorithm using tree-structured stacks. All the experiments are done on Sun 3/260 machine and using Quintus PROLOG.

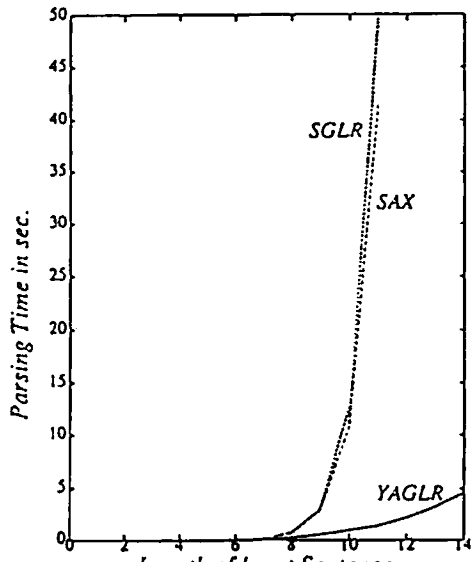


Figure. 5.1

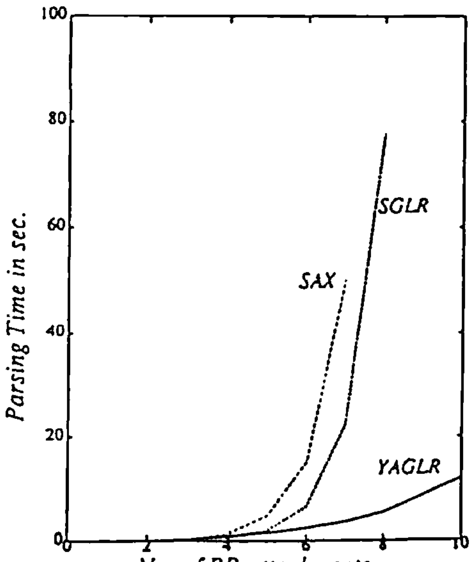


Figure. 5.2

In figure 6.1 we give the results for the grammar in [Johnson,89] for which Tomita's and even SAX or SGLR doesn't fare well. In figure 6.2 we give the result for a larger grammar with 123 grammar rules. For fig.6.1 we can easily prove that the parsing time is in the order of $\Omega(n^3)$ [Tanaka, 91] and also from fig.6.2, for general CFG, we can infer that the time compexity of YAGLR is in the order of $\Omega(n^3)$. The results in fig 6.1 and 6.2 shows the parsing time without forming trees for all the three SAX, SGLR and YAGLR parsers. Figure 6.3 shows the results of our preliminary experiment on the memory space consumed by YAGLR for the parsing of fig.6.1 and fig.6.4 shows that of fig.6.2. From these comparisons the performance of YAGLR is clearly understandable.

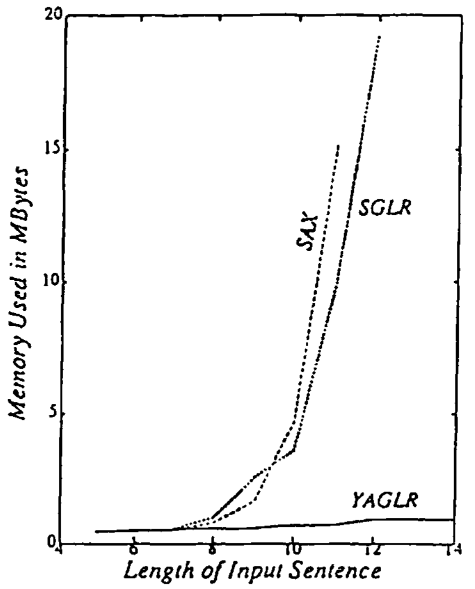


Figure. 5.3

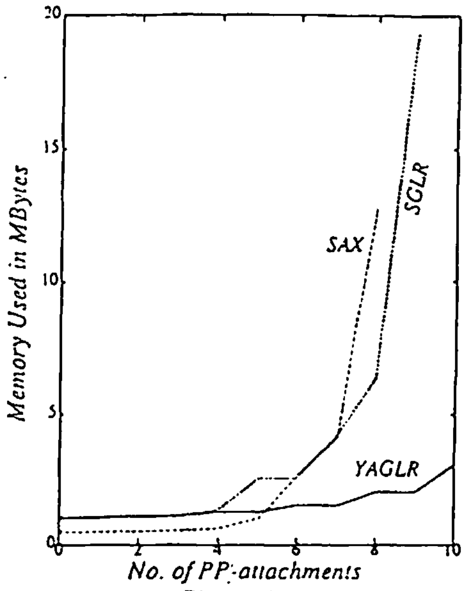


Figure. 5.4

The memory space consumed by YAGLR is very less since YAGLR does effective and efficient merge operations on GSS. Note that in YAGLR we produce the partially parsed informations (ie, *drits*), whereas in our experiment, SAX and SGLR are not producing any form of partially parsed informations. It is the reason why YAGLR needs more space in case of PP-attachments less than 5. Still we are testing our algorithm with different grammars and we will present them in our preparing paper.

7 Conclusion

We have described the basic idea of YAGLR. The followings are main features discussed.

1. For optioally choosen CFG, the parsing time of YAGLR for an input of length n seems to be in the order of n^3.

2. In the reduce action, we creat *drits* which are symmetrical to Earley's item and the reasons for creating *drits* has been explained.

3. Since YAGLR is based on LR parsing algorithm, the total number of *drits* formed is less than that of Earley's items.

4. In YAGLR we used GSS similar to that of Tomita's and merging of GSS is more deeper and effective. Because of our effective merge algorithm we can even use tree-structured stacks instead of GSS and on using tree-structured stack the memory space used is very less which is realized by our experiments.

5. Experimentally, we proved that our YAGLR parsing algorithm parses the input much faster. When the ambiguity of the given input becomes more the parsing speed of YAGLR increases more and more compared to other algorithms.

A few of interesting problems have remained unsolved with YAGLR.

1. To prove the time complexity of YAGLR for general CFG is $\Omega(n^3)$.
2. To estimate the memory space needed for parsing.
3. To develop parallel algorjthm for YAGLR method.
4. To develop a parallel algorithm for the tree generation from *drits*.

In Kipps [Kipps 89], he made a simple correction in Tomita's algorithm and showed that for general CFG, the parsing time can be in the order of $\Omega(n^3)$. But his algorithm requires space more than Tomita's. In YAGLR, the space required seems to be minimized because of our merge algorithm. For a general CFG, it needs to do a detailed estimation of the memory space used by YAGLR.

With regard to the parallel algorithm for YAGLR, it is important to make waiting time as less as possible when a shift-reduce conflict occurs [Numazaki,90].

At present the tree forming algorithm of YAGLR is corresponding to the one from Earley's items. In this algorithm for the formation of one tree, the time needed is in the order of n^2 [Aho, 72]. Since we want to reduce the time consumed for forming trees, we would like to do research on the parallel algorithm for the tree formation from *drits*.

References

[Aho 72] Aho,A.V. and Ulman,J.D.: *The Theory of Parsing, Translation, and compiling*, Printice-hall, New Jersey (1972).

[Earley 70] Earley,J.: *An Efficient Augmented-Context-Free Parsing Algorithm*, comm. of ACM, 13, 1-2, pp.95-102 (1970).

[Johnson 89] Johnson,M.: *The Computational Complexity of Tomita's Algorithm*, International parsing workshop'89, Carnegie-Mellon University, pp.203-208 (1989).

[Kipps 89] Kipps,J, R.: *Analysis of Tomita's Algorithm for General Context-Free Parsing*, International parsing workshop'89, Carnegie-Mellon University, pp.193-202 (1989).

[Matsumoto 88] Matsumoto,Y. : *Natural Language Parsing Systems based on Logic Programming*, Dotor Thesis of Kyoto University, Kyoto, Japan, 1988.

[Numazaki 90] Numazaki,H. and Tanaka.H : *A New Parallel Algorithm for Generalized LR Parsing*, COLING'90 , Vol.2, pp.305-310 (1990).

[Numazaki 90] Numazaki,H. and Tanaka.H : *SGLR : A Sequential Generalized LR Parser in Prolog* Information Processing Society of Japan Vol.32 No.3, 1991.

[Tanaka 89] Tanaka,H. and Numazaki,H.: *Parallel Generalized LR Parser Based on Logic Programming*, 1st Australian-Japan Joint Symposium on Natural Language processing, pp.201-211 (1989).

[Tanaka 91] Tanaka,H. and Suresh,K.G: *YAGLR Method: Yet Another Generalized LR Parser*, SIG. NLP 83-11, Information Processing Society of Japan, pp.79-88 (1991) (In Japanese).

[Tomita 86] Tomita,M: *Efficient Parsing for Natural Language*, Kluwer, Boston, Mass(1986).

[Tomita 87] Tomita,M: *An Efficient Augmented-Context-Free Parsing Algorithm*, Computational Linguistics, 13, pp.31-46(1987).