PROCCDINGS OF "THE INDIAN COMPUTING CONGRESS"
HELD AT HYDERABAD DURING DECEMBER 26-29, 1991

IMPLEMENTATION AND EVALUATION OF YET ANOTHER GENERALIZED LR PARSING ALGORITHM

K.G. SURESH and HOZUMI TANAKA
Department of Computer Science, Tokyo Institute of Technology
2-12-1 Ookayama Meguro-ku Tokyo 152, Japan

Abstract

This paper presents an implementation and evaluation of our new generalized LR parsing algorithm called Yet Another Generalized LR parsing algorithm (YAGLR). In its original version, YAGLR uses graph-structured stack (GSS) whereas in the preliminary implementation we use tree-structured stack (*trss*). The merge operation of stacks in our algorithm is deeper than top nodes and is effective. Hence the parsing time and the reduction in memory space are remarkable. Due to effective merge operations, even when using *trss*, we retain packed nature of GSS and thus not causing heavy loss of memory space. Through reduce actions, YAGLR creates items called *drit* which are symmetrically different from Earley's item. The advantages in creating *drit* are realized. Through our implementation, we practically prove that for a context-free grammar with reasonable size and complexity, YAGLR's parsing time is in the order of n^3, where n is the length of an input sentence. We conclude that YAGLR has the 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]. This 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 an input sentence is less ambiguous [Tomita 86]. It has been shown by [Johnson 89] and [Kipps 89] that for some CFG, Tomita's algorithm dose not fare well compared to Earley's algorithm.

In this paper we present an implementation and evaluation of our new parsing algorithm YAGLR [Tanaka 91]. This algorithm in its original version, uses graph-structured stack (GSS). In this paper we use tree-structured stack (*trss*) to implement YAGLR. Through out this paper we explain all the actions of YAGLR on a set of *trss*. We call a set of *trss* as TRSS. The reasons for using *trss* is of two folds. First, the implementation of algorithms which we use to compare in this paper are also implemented using *trss*. Second, because of our merge algorithm, we find that even using *trss*, the parsing time and reduction in memory space are remarkable.

Tomita's algorithm creates packed forest during parsing process whereas YAGLR creates item called *dot reverse item* (*drit*) which are symmetrically different from Earley's item. These *drits* make not only effective merge operations possible, but also ease the removal of duplicated items. In the following sections, we briefly state about generalized LR parsing and the necessity of *drits*. Rest of the paper gives the implementation details with the evaluation of YAGLR based on experimental results. We also prove experimentally that for the most complex CFG given by Tomita [Tomita 86], YAGLR's parsing time is in the order of n^3.

TATA McGRAW-HILL PUBLISHING COMPANY LTD,
NEW DELHI

2 Generalized LR Parsing : An Overview

The generalized LR parsing algorithm uses stacks and an LR parsing table generated from given grammar rules. An English grammar and its LR parsing table are shown in figure 2-1 [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 2-1

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		

The parsing table consists of two fields, a parsing 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), which is the grammatical category of an input sentence. Here, $ represents end of the input 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 'shN' in some entries of the action field in LR table indicates that the generalized LR 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 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 2-1 has conflicts in state 11 and 12 for column 'p'. Each of the two conflict contains both a shift and a reduce actions, which is called a shift/reduce conflict. When our parser encounters a conflict, all reduce actions should be carried out before the shift action.

3 Dot Reverse Item

During reduce actions, YAGLR creates *drits* which are symmetrically different from Earley's items. Since a state always accompanied with position number, we call the pair as a node in the rest of this paper (see 4.1). From the *trss* we can create either Earley's items or *drits*.

One of the important factor in creating *drits* is that, it enable us to do deeper merge of TRSS and makes the structure of TRSS much simpler. In other case, if we create Earley's items, the deep merge is not possible and we have to restrict only to the merge of top nodes, and if we do deep merge then it leads to the creation of unwanted items. The reason why the creation of proper *drits* is possible comes from the fact that LR parsing is based on right-most derivation. Another important factor in using *drits* is the localization of duplication checks. The position number j inside Earley's item $I_i \ni [A \rightarrow B \cdot C\ j]$ will change within the processing of a single input word. On the other hand, the position number j inside *drits* will remain the same throughout the processing of the input word w_j and thus it enables us to limit the duplication check within the processing of a single input word. Therefore we can localize the range of duplication check of *drits*. The *drits* are elaborately discussed in [Tanaka 91].

The following is a formal definition of a *drit*.

Let G = (N, T, P, S) be a CFG and let $w = w_1\ w_2\ ....\ w_n \in T^*$ be an input sentence in T^* which is a set of a sequence of terminal symbols. For a CFG rule $A \rightarrow X_1\ ...\ X_m$ and $0 \leq j \leq n$, $[A \rightarrow X_1\ X_2\ ...\ X_k \cdot X_{k+1}\ ...\ X_m,\ j]$ is called a *drit* for w. The dot between X_k and X_{k+1} is a metasymbol not in N and T. The 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 $S \overset{*}{\Rightarrow} \gamma\ A\ \delta$, $\beta \overset{*}{\Rightarrow} w_{i+1}w_{i+2}\dots w_j$, and $\delta \overset{*}{\Rightarrow} w_{j+1}w_{j+2}\dots w_n$ where the dot position is a suffix i of an item set I_i.

The difference of a *drit* with an 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.

4 An Implementation of YAGLR Algorithm

In this section we will give the structure of a TRSS along with shift and reduce actions on TRSS followed by merge operations. In our implementation, each entry in a LR parsing table is regarded as a process which will handle shift, reduce, accept and error actions.

4.1 Structure of TRSS

In parsing, the basic parsing process are determined by a sequence of state numbers in the stack. Whether or not we use grammatical symbols (as in generalized LR parsing) or packed forest (as in Tomita's method) or position numbers (as in our method) along with a state in the stack, they do not affect the basic parsing process. Each node of a *trss* used in YAGLR has the following structure :

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

The set of position numbers are used to create *drits* during reduce actions. In general, there will be several top nodes in TRSS, but after merging, the remaining top nodes will be at most no more than the number of distinct states. Even though we use *trss* in our implementation, because of our merge operations, we retain the packed nature of GSS. An example of *trss* is given in (a) and its list structure in (b). In the *trss* in (a), [{5},6] is the top node and other nodes below top nodes such as [{4},12], [{3}, 8], [{2},7], [{1}, 2] and [{0},0] are all called parent nodes of the top node.

(a)

$$\begin{tikzcd}
[\{0\},0] \arrow[r, dash] & [\{1\},2] \arrow[r, dash] & [\{2\},7] \arrow[r, dash] & [\{4\},12] \arrow[dr, dash] & & \\
 & & & & [\{5\},6] & \text{Top} \\
[\{0\},0] \arrow[rr, dash] & & [\{1\},2] \arrow[r, dash] & [\{3\},8] \arrow[ur, dash] & &
\end{tikzcd}$$

(b) [[{0}, 0, {1}, 2, {2}, 7, {4}, 12], [{0}, 0, {1}, 2, {3}, 8], {5}, 6] Top

4.2 Shift action

Let us consider a shift action 'sh,u' to a *trss* as shown in (c). It shifts (pushes) a new node onto top of the *trss* getting (d) and creating a *drit* in I_i as shown in (e). The position number of the shifted node in (d) is increased by one.

(c) $\ldots -[\{M\}, s] \text{———} [\{i\}, t]$ Top (sh, u)
(d) $\ldots -[\{M\}, s] \text{———} [\{i\}, t] \text{———} [\{(i+1)\}, u]$ Top
(e) $I_i \ni [X \to \cdot\, w_{i+1}, i+1]$

4.3 Reduce action

Let us consider a reduce action for a *trss* using a CFG rule $A \to X_1X_2 \ldots\ldots X_m$, having m nonterminal symbols on its RHS. Applying this rule for the reduce action on (f), the stack (g) is obtained along with the creation of a set of *drits* as shown in (h).

(f) $\ldots -[P_k, s_k] \text{———} [P_{k+1}, s_{k+1}] \text{———} \cdots\cdots \text{———} [P_{k+m}, s_{k+m}]$ Top
(g) $\ldots -[P_k, s_k] \text{———} [P'_{k+m}, t]$ Top

where $P_k = \{a, b, \ldots\}$, $P_{k+1} = \{c, d, \ldots\}, \ldots, P_{k+m-1} = \{e, f, \ldots, g\}$, $P_{k+m} = \{i\}$, $P'_{k+m} = \{i\}$. The state 't' in (g) is a new state determined by goto field of both s_k and A. Note that a set of position numbers, namely P_{k+m} at the top node of (f) is $\{i\}$ which includes only one position number of the last input word shifted so far. A set of position numbers P'_{k+m} remains the same as $\{i\}$ after the reduce action.

(h) Creation of *drits* :

$I_a \ni [A \to \cdot\, X_1X_2 \ldots X_m, i]$
$I_b \ni [A \to \cdot\, X_1X_2 \ldots X_m, i]$
....
$I_c \ni [A \to X_1 \cdot X_2 \ldots X_m, i]$
$I_d \ni [A \to X_1 \cdot X_2 \ldots X_m, i]$

....
$I_e \ni [A \to X_1X_2 \ldots \cdot X_m, i]$
$I_f \ni [A \to X_1X_2 \ldots \cdot X_m, i]$
....
$I_g \ni [A \to X_1X_2 \ldots \cdot X_m, i]$

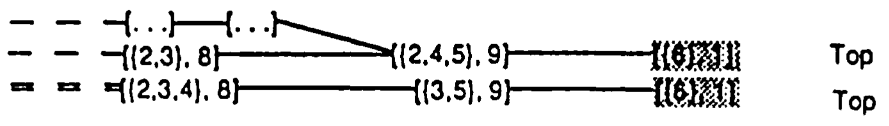
The position number *i* inside a *drit* is a position number of the top node in the stack and is remained unchanged until the next shift action occurs. Note that the *drit* such as $[A \to X_1, \ldots, X_m \cdot, i]$ $(\in I_i)$ is not created because they do not contribute to the formation of trees.

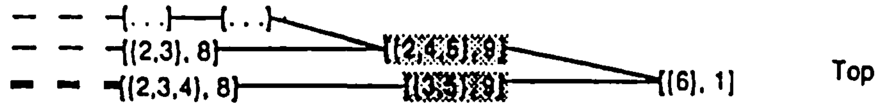
4.4 Merge of Nodes

In our merge operation, the nodes which have the same states can only be merged. Our merge operation begins from the top nodes with the same state and then proceeds one level down towards the parent nodes. To merge two nodes with the same state, we apply the following operations (M1) and (M2).

(M1) The two top nodes $[\{i\}, s]$ and $[\{i\}, s]$ are merged into one node as $[\{i\}, s]$ which inherits all the parent nodes of the two top nodes before merge.

For example, by applying (M1) to the TRSS as shown in (i), we get the TRSS as shown in (j).

(i)


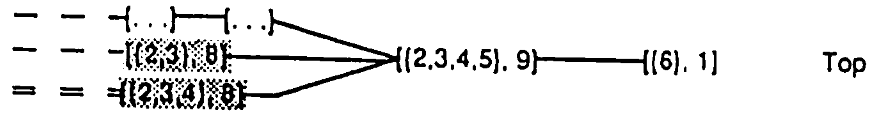
(j)


(M2) For the two parent nodes [M, s] and [N, s] of X (X is a merged node), apply either (M21) or (M22).

(M21) If M is neither a subset nor a superset of N, a new merged node [M ∪ N, s] is formed as a new parent node of X and all the parent nodes of [M, s] and [N, s] will become the parent nodes of the new merged node [M ∪ N, s].

For example, in case of (j), X is [{6}, 1] and its parent nodes to be merged are [{2,4,5}, 9] and [{3,5}, 9]. The resultant TRSS after applying (M21) to (j) is shown in (k).

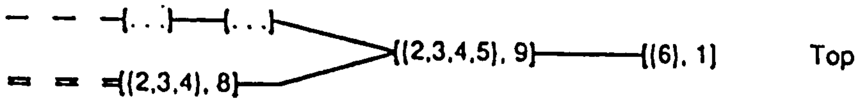
(k)



(M22) If M is a subset of N, then simply take [N, s] as a merged node and no more merge is necessary beyond this level. The parent nodes of X from [M, s] is removed. If M is a superset of N, then take [M, s] as a merged node and parent nodes from X to [N, s] are removed. No more merge is necessary beyond this level.

For example, in case of (k), X is [{2,3,4,5}, 9] and the parent nodes to be merged are [{2,3}, 8] and [{2,3,4}, 8]. We now apply (M22) to the above TRSS (k) and get the TRSS as shown in (l).

(l)



4.5 Merge Algorithm of TRSS

Since we defined the merge operations considering two nodes, we now give the merge algorithm of TRSS as follows. Our merge is performed in depth-first method by considering two *trss* at a time.

```
procedure merge (TRSS);
  begin
   Initialize TmpStk to [];
   while TRSS ≠ empty  do
     repeat
       pickup and retract a  trss (call target_trss) from TRSS;
        if at least one  trss with target_trss's same top node exits in TRSS
        then
           begin
             repeat
                pickup and retract a  trss (call s_trss) from TRSS having same top node of target_trss;
                apply (M1) to target_trss and s_trss to get a merged top node;
                for the parent nodes of merged top node apply (M2);
                name the resultant of the merges of target_trss and s_trss as m_trss;
                target_trss := m_trss;
             until no more  trss with same top nodes as target_trss in TRSS exist;
           end
       put the target_trss into the TmpStk;
     until TRSS becomes empty;
   TRSS := TmpStk;
  end
```

In applying (M2), if (M21) is applied then our merge proceeds one level down towards the parent nodes by calling (M2) recursively. However in case of applying (M22), we do not need to proceed our merge further.

4.6 Procedure of YAGLR

Let us give a complete algorithm of YAGLR.

```
1. Set the initial state of a set of trsses (TRSS) as :   (Bottom)   [{0}, 0]   (Top)
2. Initialize the TempStack to [ ]
3. If TRSS is empty then goto 5;
   Pick up and retract one trss from TRSS (TRSS := TRSS - trss);
   for this trss
      Assign the actions determined by LR table;
         case actions of
            'accept': end with "success" for the trss and goto 3;
            'error' : end with "failure" for the trss and goto 3;
            'shift' : push the trss into TempStack and goto 3;
            'reduce': goto 4;
            'shift/reduce':
                        push the trss with the shift action into TempStack and
                        goto 4 carrying the trss with the reduce action(s)
         end;
4. do the reduce action(s) and push the newly formed trss(es)' into TRSS and merge the TRSS;
   goto 3.
5. If TempStack := [ ] then return;
   Perform shift action for every trss in TempStack and push the resultant into TRSS ;
   merge the TRSS;
   goto 2.
```

5 Evaluation of YAGLR

In this section we present the evaluation of YAGLR based on 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.

5.1 Experimental Environment

The experiments were done on Sun 3/260 machine and using Quintus PROLOG. We used different sets of grammars in our experiment ranging from the grammars with 3 rules to 550 rules to study the parsing efficiency of our algorithm. In this paper we concentrate on four different type of grammars. Gram-1 is a grammar in [Johnson 89] which is a highly densely ambiguous grammar. For this grammar and its input pattern readers are requested to refer [Johnson 89, pp.205]. Gram-2 is a grammar with 44 rules, gram-3 with 123 rules and gram-4 with 400 rules. Gram-2 and gram-4 are taken form [Tomita 86], and gram-3 is from our lab in Tokyo Institute of Technology. Gram-4 becomes highly ambiguous and could therefore be considered as one of the toughest natural language grammars in practice. So we center all our experimental results mainly around gram-4. The results of gram-1 and gram-3 were give in [Tanaka 91].

The input for the grammars 2, 3 and 4 is made more systematically. The n-th sentence in the set is obtained by the schema, *noun verb det noun* $(prep\ det\ noun)^{n-1}$ [Tomita 86]. The example sentence with this structure is: *I saw a girl on the bed in the apartment with a telescope* ...... The ambiguity of this type of sentences grows enormously. This type of sentences are necessary to find the parsing efficiency against sentence ambiguity.

All our program are written in PROLOG and are complied using Quintus PROLOG. Since we are interested in the ratio of parsing time, it will be same either interpreted or compiled. The parsing time is determined by CPU time minus the time consumed for garbage collection (gc). We find that the gc consumed during the execution of our algorithm is very less (even though we use *trss*). If we include the gc time, then the ratio between YAGLR and other parsers will vary a large extent - in a positive way to YAGLR. The parsing time in our implementations are without forming trees for SAX, SGLR parsing while YAGLR parsing creates *drits*.

5.2 Experimental Evaluation of YAGLR

Here, we give our preliminary results.on the implementation of YAGLR. The figure 5-2(a) and 5-2(b) shows the parsing time of YAGLR for gram-4 against length of the input sentence and against sentence ambiguity respectively. We find that YAGLR parses the sentence more faster, as the ambiguity of the sentence increases. In other words, as the ambiguity increases, the parsing time of YAGLR decreases rapidly regardless to the size of the grammar or length of the input sentence.

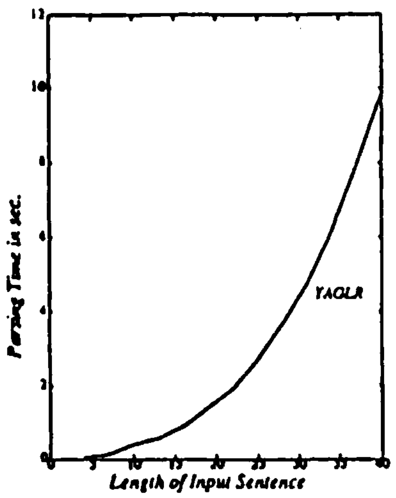


Figure 5-2(a)

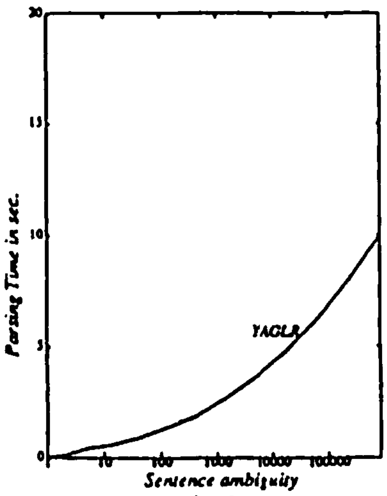


Figure 5-2(b)

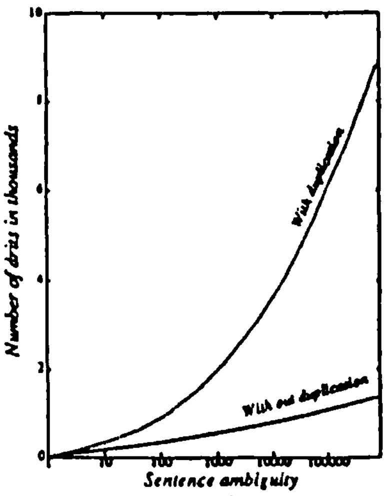


Figure 5-2(c)

The figure 5-2(c) shows the number of *drits* created by YAGLR against the ambiguity. Here, the total *drits* produced during parsing is indicated by a dashed curve, which includes duplicated *drits*. After the shift of an input word w_i, our parser makes duplication check to the *drits* produced in between w_{i-1} and w_i. The other curve shows the number of non-duplicated items created among duplicated items. Our parsing time shown in fig.5-2(a) and (b) includes the time consumed for removing the duplicated items. If the sentence is ambiguous, the creation of duplicated *drits* are unavoidable. It should be noted that, if we do not do the duplication check, YAGLR parser will run more faster.

5.3 Comparison with other methods

In this subsection, we would like to compare the performance of YAGLR with other parsers. In figure 5-3(a) and 5-3(b), we give the ratio of parsing time of SGLR/YAGLR and SAX/YAGLR against sentence length and sentence ambiguity respectively for gram-4. The ratio will be the same by taking it either against sentence length or ambiguity. The high, the ratio of parsing time of SGLR/YAGLR or SAX/YAGLR, the low, the parsing time of YAGLR. Here, we see that, SGLR/YAGLR ratio and SAX/YAGLR ratio is high for a sentence with considerable length, as the ambiguity increases.

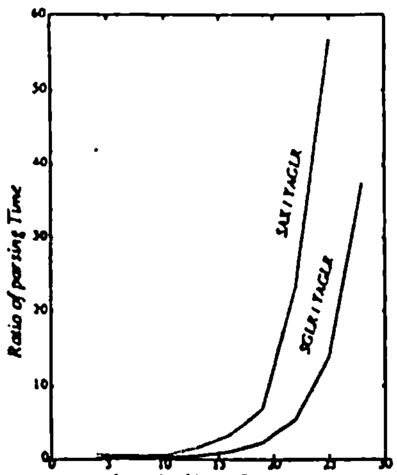


Figure 5-3(a)

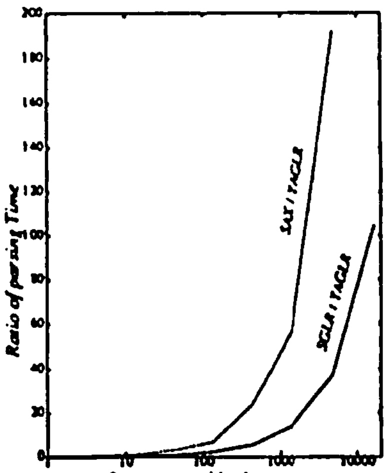


Figure 5-3(b)

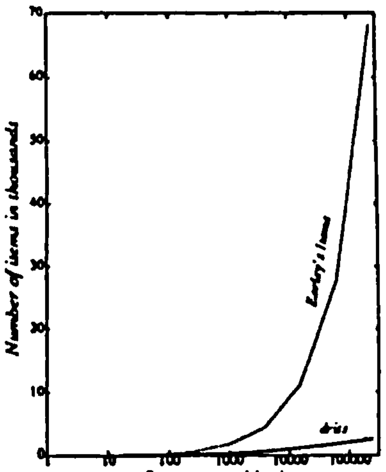


Figure 5-3(c)

The figure 5-3(c) shows a comparison of Earley's items and *drits*. For this comparison we consider gram-1 and its input pattern. The reason for using gram-1 is that, in case of gram-1, the total number of non-duplicated items created by either *drits* or Earley's items should be equal. So it will be better considering this grammar to compare the number of *drits* and Earley's items created including duplicated ones. From the fig.5-3(c) we can realize the advantages of creating *drits* ove Earley's items.

There are grammars, for which the number of non-duplicate Earley's items are less than that of *drits*. But, the total number of items created including duplicated items are far less in case of *drits*. The parsing time includes the creation of total number of items which includes duplicated items. The more the duplicated items, the more the time consumed for creating and removing. Also, as we disussed briefly in section 2, on creating Earley's items using our algorithm, leads to the creation of unwanted items [Tanaka 91]. Hence it is safe to conclude that *drits* are better than Earley's items.

Gram-1					Gram-4				
I/P	*TIME in msec.*			*Trees*	*n*	*TIME in msec.*			*Trees*
	SAX	SGLR	YAGLR			SAX	SGLR	YAGLR	
5	34	50	67	20	1	50	17	67	1
6	67	83	117	70	2	117	84	167	2
7	233	250	183	256	3	267	150	400	5
8	800	833	367	969	4	967	350	600	14
9	2867	3117	517	3762	5	3067	1000	934	42
10	10750	12650	866	14894	6	9700	3200	1417	132
11	41616	49716	1383	59904	7	32217	10683	1917	429
12	262250	222235	2017	244088	8	113135	37800	2700	1430
					9	398832	137000	3667	4862
					10	-	498731	4750	16796

Figure 5-3(d)

Some raw empirical datas got from the experimental results using gram-1 and gram-4 are given in the table in fig. 5-3(d). In the table, I/P denotes length of the input sentence, *n* denotes sentence number according to the schema described in 5.1 and *Trees* denotes the number of ambiguities. These datas entierly depends on the machine system and the programming language used. But we hope that, the ratio of parsing time will be the same for any system under a particular programming environment.

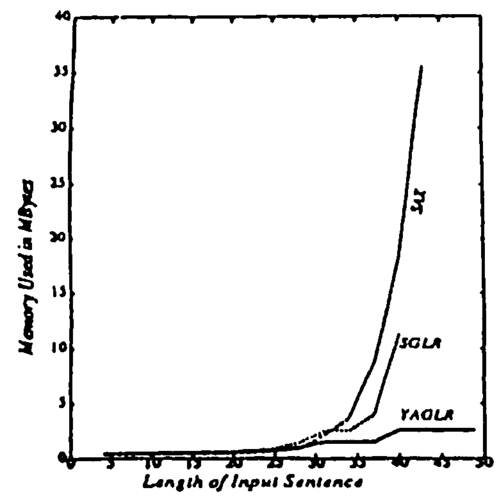


Figure. 5-3(e)

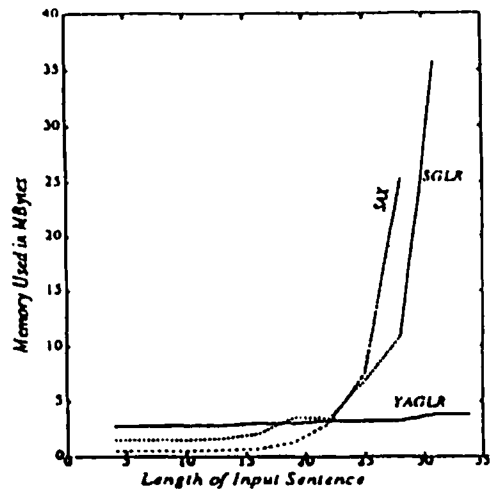


Figure. 5-3(f)

The figure 5-3(e) and 5-3(f) shows results of memory space consumed by YAGLR for the parsing of gram-2 and gram-4 respectively. YAGLR consumes very less memory space due to its effective merge operations. Note that in YAGLR we produce *drits*, whereas in our experiments, SAX and SGLR are not producing any form of partially parsed informations. It is the reason why YAGLR needs more space up to the sentence of length 18 for gram-4. The amount of memory space needed depends on the size and ambiguity of the grammar we use. In case of gram-2, which has only 44 rules, the memory space consumed by the sentence of length up to 18 is comparable. However, when the length of input sentence gets long, the reduction in memory space is remarkable regardless to the size and ambiguity of the grammar.

5.4 Experimental Computational Complexity of YAGLR

For gram-1 we proved theoretically, the complexity of YAGLR as in the order of n^3 [Tanaka 91 (in Japanese)]. But we are yet to prove in case of general CFG. In this subsection we give our experimental proof for the complexity of YAGLR. The figure 5-4 shows the order of parsing time of YAGLR for gram-1 and gram-4. On taking log scale for both X and Y axis we find that for the parsing time to

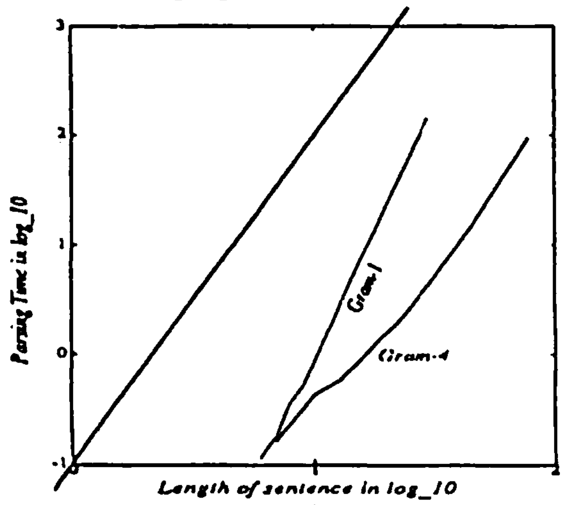


Figure. 5-4

be in the order of n^3, the slope of the time curve must be ≤ 3. Thus the line passing through X and Y axis in fig. 5-4 shows the sample line with slope 3. In the fig.5-4 we find the time curve of gram-4 is in parallel with the sample line and so the time complexity of gram-4 is in the order of n^3. In case of the time curve of gram-1, we find that it is not in parallel with the sample line and it is nearly in the order of n^4. However, we proved theoretically that the complexity of YAGLR for gram-1 is in the order of n^3.

6 Conclusion

We have shown the basic idea of YAGLR parsing algorithm and its implementation with evaluation. It should be noted that after completing parse, the syntactic trees are formed from *drits* obtained during the parsing process. Even though we used TRSS in our implementation, we find that the parsing speed and the memory space consumed by YAGLR is very less. There are certain grammars for which if we use TRSS the copying of stacks creates inefficiency. For this reason we would like to implement our algorithm in GSS as in our original version. Through our implementation, we practically proved that for a CFG with reasonable size and complexity, YAGLR's parsing time is in the order of n^3. Our future works includes showing the time complexity of YAGLR for any CFG, developing a parallel algorithm for YAGLR method and also for the tree generation 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 : *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 : Yet Another Generalized LR Parser*, Proceedings of RO-CLING IV, pp.21-31 (1991)

[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).