自然言語処理 86－7
（1991. 11. 15）

YAGLR[1]アルゴリズムの実現と評価

K．G．スレッシュ　　沼崎浩明　　田中　穂積

東京工業大学　工学部
〒152 東京都目黒区大岡山2-12-1

あらまし

本研究では，ＹＡＧＬＲ法と呼ぶ新しい一般化ＬＲ法を提案する．ＹＡＧＬＲ法に基づくパーザを計算機にインプリメントし，その性能を評価する．ＹＡＧＬＲ法は，実際にグラフ構造化スタックを用いる．現在ＹＡＧＬＲアルゴリズムは，木構造化スタックを用いて重複計算を回避する．ＹＡＧＬＲ法では，スタックのマージをトップ節点よりも深く進める．解析中に統語解析木は作らず，統語解析木を構成する部品(逆ドット項)を作成する．ＹＡＧＬＲ法は，スタックのマージを深く進めることと，逆ドット項を作成することによって従来のパーサと比較して統語解析時間が高速になる．さらに，メモリ空間の圧縮も可能になる．ＹＡＧＬＲパーザは論理型言語 Prolog を用いて，計算機上に実現する．木構造化スタックが消費するメモリ空間は，Prolog の shared-structure の機構によりスタックが消費するメモリ空間はグラフ構造化スタックと同じである．本論文では，ＹＡＧＬＲ法のインプリメンデーションと実験による性能評価の結果を示す．また，実験結果に基づいて，ＹＡＧＬＲの解析時間が n^3 のオーダであるという事実を示す．

A Practical Realization of YAGLR Algorithm on Prolog Using Tree-Structure Stacks

K. G. SURESH　　Hiroaki NUMAZAKI　　Hozumi TANAKA

Department of Computer Science, Faculty of Engineering,
Tokyo Institute of Technology,
2-12-1, Meguro-ku, Ôokayama, Tokyo 152
Email : suresh@cs.titech.ac.jp

Abstract

In this paper we describe an implemenation and evaluation of our new generalized LR parsing algorithm called YAGLR. In our implementaion we used tree-sturcture stack to realize YAGLR algorithm, whereas in its original version we use graph-structure stack. The merge operations of YAGLR proceeds deeper than top nodes effectively. Through reduce actions YAGLR creates items called *drit* which are symmetrically different from Earley's item. Through our implementaion we show the advantages of creating *drits* instead of Earley's items. We also show through our implementation that, the parsing time of YAGLR is in the order of n^3, where n is the length of an input sentence. Because of our merge algorithm and due to the nature of shared-strucure of Prolog, even though we use tree-structure stack, we retain the packed nature of GSS. This reduces the memory space used by YAGLR to a greater extent.

[1]YAGLR : Yet Another Generalized LR parser

1 Introduction

In recent times, ambiguous context-free grammars (CFG) are used for the syntactic and semantic processing of natural language. Efficient syntactic and semantic parsing for context-free languages are generally characterized as complex, specialized, highly formal algorithms. There are two major parsing algorithms that are used for the efficient parsing. The first is Earley's algorithm (Earley, 1970), which produce the parsing result in the form of a *parse list* consisting of set of items. This reduces the computational dependence on input sentence length from exponential to cubic cost. An attractive feature of Earley's algorithm is that it can easily be modified to parse co-ordinate structures of unlimited breadth. Such structures exist in the logical form of natural sentences. Numerous variations on Earley's method have developed into a family of chart parsing algorithms (Winograd, 1983).

The second is Tomita's algorithm (Tomita, 1986), which generalizes Kunth's (Kunth, 1965) and DeRemer's (DeRemer, 1971) computer language LR parsing techniques. Tomita's algorithm constructs parse forests, as the parsing result, which are nothing but a set of items. Tomita's algorithm uses the data structure called graph-structure stacks (GSS). 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, 1986).

Even though in (Tomita, 1986) it was climed that Tomita's algorithm produces all the parsing trees during the parsing process, they are nothing but a set of items with pointers. A method for disambiguation of trees from the parse forest is proposed by (Tomita, 1986), in which the disambiguation is done by asking the user. In (Johnson, 1989) and (Kipps, 1989) it was stated that any algorithm which uses packed forest representation will take exponential time for certain sets of CFGs. In Earley's algorithm, parse trees are formed from the parse list created during parsing. Earley's algorithm works with the time complexity, in the order of n^3 for any CFG (Aho,Ulman 1972). In our new generalized LR parsing algorithm we stick on to the formation of items as the result of parsing. But our items are symmetrically different from Earley's.

In this paper we present an implementation and evaluation of our new generalized LR parsing algorithm called YAGLR (Tanaka, 1991). In the implementation of this algorithm we used tree-structure stacks (*trss*). In its original version we used graph-structure stacks (GSS). In this paper we explain all the actions of YAGLR on a set of *trss*, which we call as TRSS. Because of our merge algorithm, we find that even using *trss*, the parsing time and reduction in memory space are remarkable. Instead of packed forest as in Tomita's algorithm, YAGLR creates items called *dot reverse item* (*drit*). These *drits* not only make effective merge operations possible, but also ease the removal of duplicated items.

In the following sections we state briefly about the *drits* and we give the implementation details of our algorithm along with the experimental results. We also prove experimentally that, the parsing time of YAGLR is in the order of n^3 for a grammar with reasonable size and time complexity in practical natural language processing. We conclude this paper with a brief discussion on our future research directions.

2 A Brief Introduction to Generalized LR Parsing

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, 1987).

(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 : A simple and ambiguous English grammar

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 2-2 : LR table for the grammar in fig.2-1

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 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 LR table indicates that the generalized LR parser has to push a look-ahead preterminal on to 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 2-2 has conflicts in state 11 and 12 for column 'p'. Each of the two conflicts contain both a shift and a reduce actions, which is called a shift/reduce conflict. When our parser encounters the conflict, all reduce actions should be carried out before the shift action.

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

Figure 2-3. An Example of Generalized LR Parsing

On input "I saw a girl with the telescope", the sequence of stack and input contents is shown in fig.2-3. For example, at line (1) the parser is in state 0 with "I" the first input symbol. As the action field of fig.2-2 in row 0 and column 'n' (the preterminal of "I") contains 'sh4', it pushes 'n' and cover the stack with state 4. That is what was happened in line (2).

Then, "saw" becomes the current input symbol. As the action of state 4 on 'v' (the preterminal of "saw") is 're3', it carries out a reduce action by using the rule NP → n. One state symbol and one grammar symbol are popped from the stack and 0 again becomes the top of the stack. Since the goto field of state 0 on NP is 2, NP and 2 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 (7), we get a conflict with 'sh6/re7', where we carry out 're7' at first and 'sh6' is waited until all the other remaining stacks experiences shift actions. At line (10) both the stacks with shift action 'sh,6' are shifted to state 6. The remaining parsing proceeds in this way.

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 section 4). 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 deep merge on 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 w_k. On the other hand, the position number j inside *drits* will remain the same throughout the processing of the input word w_k, ($w_k = w_j$) 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, 1991).

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 $S \stackrel{*}{\Rightarrow} \gamma\ A\ \delta$, $\beta \stackrel{*}{\Rightarrow} w_{i+1}w_{i+2}\ldots w_j$, and $\delta \stackrel{*}{\Rightarrow} w_{j+1}w_{j+2}\ldots w_n$ where the dot position is a suffix i of an item set I_i.

The 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 Realization of tree-structure stack

In YAGLR method, we do not use grammatical categories along with a state number as in generalized LR parsing algorithm shown in fig.2-3. In Tomita's method, packed forest representation is used instead of grammatical symbols. On the other hand, we use position numbers along with a state number. The position number indicates the position upto which the shift of an input sentence has been completed. Note that a sequence of state numbers in the stack completely determines the basic parsing process. In other words, whether or not we use grammatical symbols or packed forests or position numbers along with a state in the stack, they do not affect our 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 and due to the shared-structure of Prolog, 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) [{0}, 0]—[{1}, 2]—[{2}, 7]—[{4}, 12]
[{0}, 0]—[{1},2]—[{3}, 8]
[{4}, 12] and [{3}, 8] → [{5}, 6]

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

5 An Implementation of YAGLR Algorithm

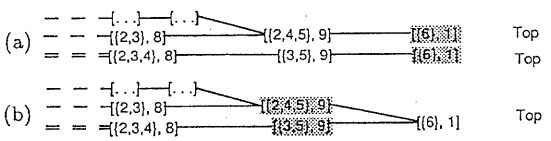
In this section we will first describe about the merge operation of YAGLR algorithm followed by the shift and the reduce actions on TRSS. Finally we give the merge algorithm and the procedure of YAGLR. In our implementation, each entry in a LR parsing table is regarded as a process which will handle shift, reduce, accept and error actions.

5.1 Merge Operation

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.

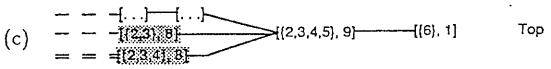
For example, by applying (M1) to the TRSS as shown in (a), we get the TRSS as shown in (b).



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

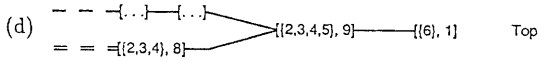
(M21) If M is neither a subset nor a superset of N, a new merged node [M ∪ N, s] is formed as a new parent node of X and all the parent nodes of [M, s] and [N, s] will become the parent nodes of the new merged node [M ∪ N, s].

For example, in case of (b), 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 (b) is shown in (c).



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

For example, in case of (c), 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 (c) and get the TRSS as shown in (d).



5.2 Shift Action

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

(e) . . . -[{M}, s]——[{i}, t] Top (sh, u)
(f) . . . -[{M}, s]——[{i}, t]——[{(i+1)}, u] Top
(g) $I_i \ni [X \rightarrow \cdot\, w_{i+1}, i+1]$

In our implementation, when a process handling shift action is called by a $trss_s$ in TRSS, the shifted node is pushed into the $trss_s$. The parsing will continue with the rest of *trss* in TRSS. Once all the *trss* in TRSS completes shift action, all the shifted nodes (which are now top nodes) with similar states will be merged using (M1). With the result of (M1), the parser will continue its action for the next input word w_{i+1}.

5.3 Reduce action

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

(h) . . . -[P_k, s_k]——[P_{k+1},s_{k+1}]- . . . -[P_{k+m}, s_{k+m}] Top
(i) . . . -[P_k, s_k]——[P'_{k+m}, t] Top

where $P_k = \{a, b, \ldots\}$, $P_{k+1} = \{c, d, \ldots\}$, . . . , $P_{k+m-1} = \{e, f, \ldots, g\}$, $P_{k+m} = \{i\}$, $P'_{k+m} = \{i\}$.
The state 't' in (i) is a new state determined from the goto field of s_k and A. Note that a set of position numbers, namely P_{k+m} at the top node of (h) 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.

(j) Creation of *drits* :

$I_a \ni [A \rightarrow \cdot\, X_1X_2 \ldots X_m, i]$
$I_b \ni [A \rightarrow \cdot\, X_1X_2 \ldots X_m, i]$
.....
$I_c \ni [A \rightarrow X_1 \cdot\, X_2 \ldots X_m, i]$
$I_d \ni [A \rightarrow X_1 \cdot\, X_2 \ldots X_m, i]$
.....
$I_e \ni [A \rightarrow X_1X_2 \ldots \cdot\, X_m, i]$
$I_f \ni [A \rightarrow X_1X_2 \ldots \cdot\, X_m, i]$
.....
$I_g \ni [A \rightarrow X_1X_2 \ldots \cdot\, X_m, i]$

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 $[A \rightarrow X_1, \ldots, X_m \cdot, i]$ ($\in I_i$) is not created because they do not contribute to the formation of trees.

In our implementation, let us consider a process handling the reduce action $r_1/r_2/.../r_p$ (where p is the total number of productions), is called by a $trss_r$ in TRSS. All the reduce actions for the $trss_r$ will be carried out to give $trss_{r_1}$, $trss_{r_2}$, . . . $trss_{r_p}$, which are the resultant *trsses* after reduce and goto actions. These resultant *trsses* will be merged with the rest of *trss* in TRSS using (M1) and

(M2). Finally, with the resultant TRSS after merge, the parser will continue its next action.

5.4 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 that of target_trss;
              apply (M1) to the top node of target_trss and
              s_trss to get a merged top node;
              for the parent nodes of merged top node
              apply (M2);
              name the result of the merges (M1) and (M2)
              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.

5.5 YAGLR Algorithm

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.

6 Evaluation of YAGLR

In this section we describe the evaluation of YAGLR based on the preliminary experimental results comparing with SAX (Matsumoto, 1988) and SGLR (Numazaki, 1991). SAX is based on the bottom up version of Chart algorithm and SGLR is based on Tomita's algorithm using tree-structured stacks.

6.1 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 three different type of grammars. Gram-1 is a grammar in (Johnson 1989) which is a highly densely ambiguous grammar. This grammar and its input pattern are given in the appendix. Gram-2 is a grammar with 44 rules, and gram-3 with 400 rules. Gram-2 and Gram-3 are taken from (Tomita, 1986). However, original version of Gram-3 is from our lab in Tokyo Institute of Technology. Gram-3 becomes highly ambiguous and could therefore be considered as one of the toughest natural language grammars in practice. In this paper we center all our experimental results mainly around gram-3. The evaluation of other types of grammars along with gram-1 are give in (Tanaka, 1991), (Suresh, 1991).

The input for the grammars 2 and 3 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, 1986). 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 the 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*.

6.2 Experimental Evaluation of YAGLR

Here, we give our preliminary results on the implementation of YAGLR. The figure 6-2(a) and 6-2(b) shows the parsing time of YAGLR for gram-3 against length of

the input sentence and against sentence ambiguity respectively.

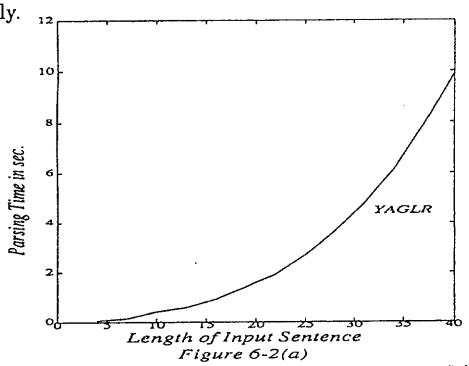


Figure 6-2(a)

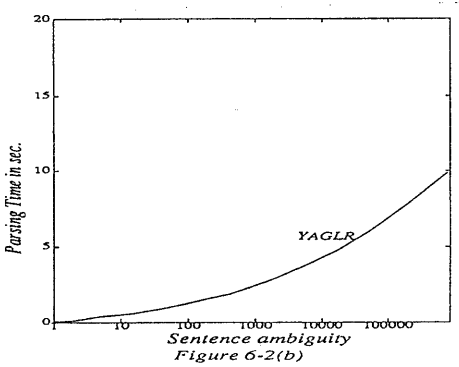


Figure 6-2(b)

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.

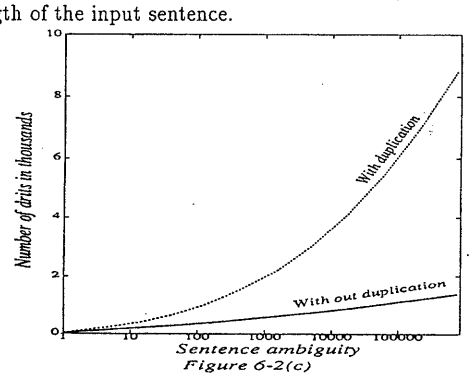


Figure 6-2(c)

The figure 6-2(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 and removes all the duplicated items. The other curve shows the number of non-duplicated items created among duplicated items. The parsing time shown in fig.6-2(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 much faster.

6.3 Comparison of YAGLR

In this subsection, we would like to compare the performance of YAGLR with other parsers. In figure 6-3(a) and 6-3(b), we give the ratio of parsing time of SGLR/YAGLR and SAX/YAGLR against sentence length and sentence ambiguity respectively, for gram-2. 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.

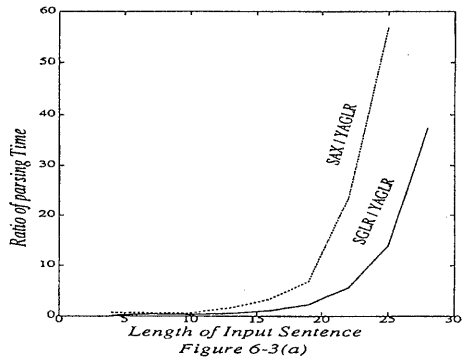


Figure 6-3(a)

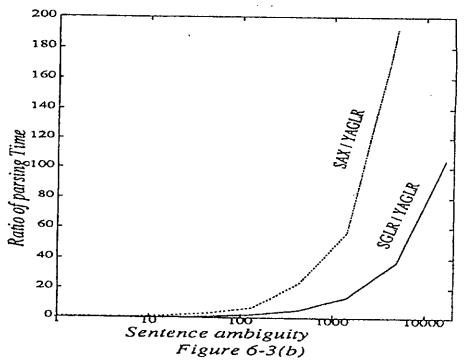


Figure 6-3(b)

The figure 6-3(c) shows a comparison of Earley's items created using Earley's algorithm and *drits* created by YAGLR for gram-3. All the duplicated items are removed during parsing and the graph in fig.6-3(c) shows that of non-duplicated items. From the fig.6-3(c) we can realize the advantages of creating *drits* over Earley's items.

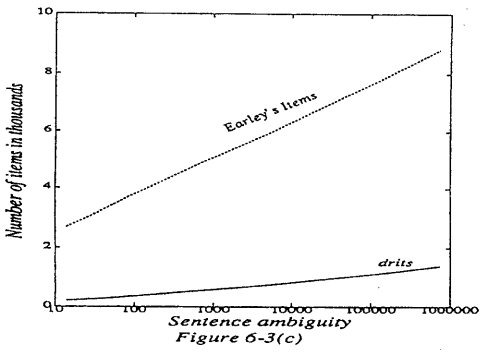


Figure 6-3(c)

There are some grammars, for which the number of non-duplicate Earley's items created using YAGLR algorithm 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. Hence it is safe to conclude that *drits* are better than Earley's items.

I/P	*TIME in msec.*			*Trees*
	SAX	SGLR	YAGLR	
5	34	50	67	20
6	67	83	117	70
7	233	250	183	256
8	800	833	367	969
9	2867	3117	517	3762
10	10750	12650	866	14894
11	41616	49716	1383	59904
12	262250	222235	2017	244088

Figure 6-3(d)

n	*TIME in msec.*			*Trees*
	SAX	SGLR	YAGLR	
1	50	17	67	1
2	117	84	167	2
3	267	150	400	5
4	967	350	600	14
5	3067	1000	934	42
6	9700	3200	1417	132
7	32217	10683	1917	429
8	113135	37800	2700	1430
9	398832	137000	3667	4862
10	-	498731	4750	16796

Figure 6-3(e)

Some raw empirical datas got from the experimental results using gram-1 and gram-2 are given in the table in fig. 6-3(d) and 6-3(e) respectively. In the table, I/P denotes length of the input sentence, n denotes sentence number according to the schema described in 6.1 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.

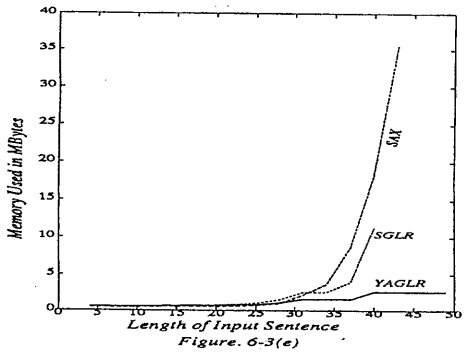


Figure. 6-3(e)

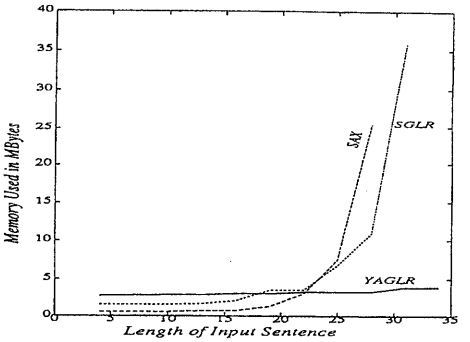


Figure. 6-3(f)

The figure 6-3(f) and 6-3(g) shows results of memory space consumed by YAGLR for the parsing of gram-2 and gram-3 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. The amount of memory space needed depends on the size and ambiguity of the grammar and the input we use. 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.

6.4 Experimental Computational Complexity of YAGLR

For gram-1 we proved theoretically, the complexity of YAGLR as in the order of n^3 (Tanaka, 1991 [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 6-4 shows the order of parsing time of YAGLR for gram-1 and gram-3.

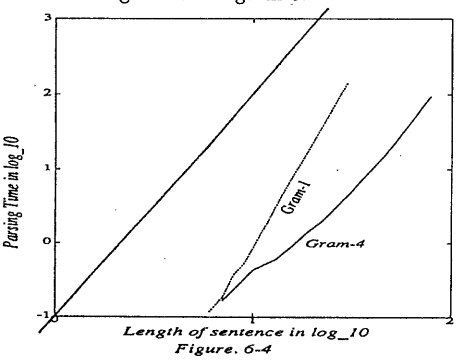


Figure. 6-4

On taking log scale for both X and Y axis we find that for the parsing time to 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. 6-4 shows the sample line with slope 3. In the fig.6-4 we find the time curve of gram-3 is in parallel with the sample line and so we can conclude that, the time complexity of gram-3 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.

7 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 of YAGLR increases 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, 1972] Aho,A.V. and Ulman,J.D.: *The Theory of Parsing, Translation, and compiling,* Printice-hall, New Jersey, 1972.

[DeRemer, 1971] DeRemer, F. : *Simple LR(k) grammars* Communications of the ACM, 14(7): 453-460, 1971.

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

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

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

[Mark, 1991] Mark Perlin. : *LR Recursive Transition Networks for Earley and Tomita Parsing,* ACL proceedings, 29th Annual Meeting, pp.98-105, 1991.

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

[Nijholt, 1991] Nijholt,A : *Overview of Parallel Parsing Strategies,* Current Issues In Parsing Technology, Chapter 14, 1991.

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

[Numazaki, 1990] Numazaki,H. and Tanaka.H : *SGLR : A Sequential Generalized LR Parser in Prolog* Information Processing Society of Japan Vol.32 No.3, 1991 (in Japanese).

[Suresh, 1991] Suresh,K.G and Tanaka,H : *Implementation and Evaluation of Yet Another Generalized LR Parsing Algorithm,* Indian Computing Congress, ICC'91 (to appear)

[Tanaka, 1989] 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, 1991] Tanaka,H. and Suresh,K.G: *YAGLR : Yet Another Generalized LR Parser,* Proceedings of ROCLING IV, pp.21-31, 1991.

[Tanaka, 1991] 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, 1986] Tomita,M: *Efficient Parsing for Natural Language,* Kluwer, Boston, Mass, 1986.

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

[Winograd, 1983] Winograd, T. : *Language as a Cognitive Process, Volume 1 : Syntax,* Addison-Wesley, 1983.

Appendix

The CFG given in (Johnson, 1989) is shown below.

(1) S → a
(2) S → SS
(3) S → S^{m+2}

The input pattern for this type of grammar is a^{n+2}, where n > m.