Implementation and Evaluation of Yet Another Generalized LR Parsing Algorithm

K.G. SURESH*
HOZUMI TANAKA*

Abstract

The recognition and understanding of natural languages by computers is an important task. There are different methods for recognising natural languages syntactically and semantically. However, none of them is satisfying in terms of time complexity for ambiguous grammars and sentences. In this paper we present an implementation and evaluation of our new generalized LR parsing algorithm called Yet Another Generalized LR parsing algorithm (YAGLR) [11]. In implementing YAGLR we use the tree-structured stack (trss) instead of the graph-structured stack (GSS) as in [14]. Through our implementation we will show that our trss is as effective as that of GSS. Due to the effective merge operations, which are deeper than top nodes, and due to the nature of shared-structure of Prolog, we retain the packed nature of GSS. We also practically show that even using trss, the time and space complexity of YAGLR is n^3 and n^2 respectively, where n is the length of an input sentence. We create items called dot reverse items (drit) as our parsing result, which are symmetrically different from the items formed in [2]. We explain the advantages of creating drits and also practically prove the effectiveness of drits. The parsing trees (syntactic structures) can be formed from a set of drits created after the parsing is completed. We use the logic programming language Prolog in implementing YAGLR.

*Department of Computer Science, Tokyo Institute of Technoloy 2-12-1, Ôokayama Meguro-ku Tokyo 152, Japan.

Journal of Information Science and Technology, Vol. 1, No.4, July, 1992.
ISSN 0971-1988

1 Introduction

Some compilers of programming languages have made use of the LR(k) parsing algorithm devised by Kunth [5] which enables us to parse an input sentence deterministically and efficiently. But the grammars used in this algorithm are limited to LR(k) grammars so that Context-Free Grammars (CFG) in general cannot be handled.

In recent times, ambiguous context-free grammars are used for the syntactic and semantic processing of natural languages. Efficient syntactic and semantic parsing for context-free languages are generally characterized as complex, specialized, highly formal algorithms. There are two exceptional methods which are used for efficient parsing. The first is Earley's algorithm [2], which produces the parsing results in the form of a *parse list* consisting of a 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 coordinate 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 [15].

The second is Tomita's algorithm [13, 14], which generalizes Kunth's LR(k) parsing techniques and extends it to handle CFG. Tomita's algorithm uses the data structure called *graph-structured stack* (GSS) and construct a parse forest as the parsing result consisting of all the trees in packed form. 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 [13].

Even though Tomita's algorithm produces all the parsing trees in the packed form 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 [13], in which the disambiguation is done by asking the user. According to Johnson [3], any algorithm which uses packed forest representation will suffer from an exponential time and space worst case complexity with respect to the input length and also with respect to grammar size. Some modifications to the Tomita's algorithm were made by Kipps [4], so that the worst case time complexity became n^3, but the Kipps algorithm's space complexity is worse than Tomita's. In Earley's algorithm, parse trees are formed from the parse list created during parsing. In this paper we refer to the item created in Earley's algorithm as Earley's item. Earley's algorithm works with time and space complexity in the order of n^3 and n^2 respectively for any CFG [1]. In our new generalized LR parsing algorithm we stick to the items formed 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 [11]. In its original version we used GSS, but in the implementation we use *tree-structured stack* (trss). In this paper we explain all the actions of YAGLR on a set of trss, which we call TRSS. Due to our merge algorithm, which merges the trss deeply in an effective way, and due to the nature of the shared structure of Prolog, we retain the packed nature of GSS. Through our implementation we practically show that even using trss, the time and space

complexity of YAGLR is n^3 and n^2 respectively where n is the length of an input sentence. YAGLR creates items called *dot reverse items* (drits) as parsing results which are symmetrically different from Earley's items. These drits not only make effective merge operations possible, but also ease the removal of duplicated items.

The rest of this paper is organized in the following way. Section 2 gives a brief introduction to the generalized LR parsing algorithm. In Section 3, we introduce the drit by comparing it with Earley's item and discuss the merits of creating drits instead of Earley's items. We also prove experimentally the advantages of creating drits in Section 5. Section 4 gives an implementation of the YAGLR algorithm along with the merge algorithm for TRSS. In Section 5 we give the evaluation of YAGLR on the basis of implementation, and we practically prove through our experiment that the time complexity 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 our paper with a brief discussion on our future research directions.

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 Figs. 1 and 2 respectively [14].

(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

Fig. 1 *An English context-free 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		

Fig. 2 *LR parsing table for the grammar in Fig. 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 input sentence. There are mainly 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 the 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 the right-hand side of the rule numbered 'N', from the top of the stack and then go to the new state determined by the 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 Fig. 1 has conflicts in states 11 and 12 for column p. Each of the two conflicts contains both a shift and a reduce action, making for what is said to be a shift/reduce conflict. When our parser encounters a 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$	

Fig. 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. 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 in row 0 and column *n* (the preterminal of "I") contains sh4, it pushes *n* and covers the stack with state 4. This is what has happened in line (2).

Then, "saw" becomes the current input symbol. As the action of state 4 on v (the grammatical category of "saw") is re3, it carries out a reduce operation by using the

rule NP $\rightarrow n$. 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 is 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 kept in abeyance until all the other remaining stacks experience shift actions. At line (10) both the stacks with shift action sh6 are shifted to state 6. The remaining parsing process proceeds in this way.

3 Earley's Item and Dot Reverse Item

In YAGLR we do not use grammatical categories along with a state number as in generalized LR parsing algorithm shown in Fig. 3; instead we use a *position number* along with a state number. The position number indicates the position up to which the shift of an input sentence has been completed. Note that a sequence of state numbers alone completely determines the basic parsing process.

During reduce actions, YAGLR creates drits which are symmetrically different from Earley's items. Since a state is always accompanied by the position number, we call the pair a node in the rest of this paper (see subsection 4.1). From the trss, we can create either Earley's items or drits. In this section, we will give the basic idea of drits and discuss the merits of creating drits instead of Earley's items.

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

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

Here, $S1$, $S2$ and $S3$ indicate states, and 3, 5 and 6 position numbers. The position number i in a node means that the parser has shifted the input words up to w_i. Thus the node [{6}, $S1$] in (a) covers the input word w_6, the node [{5}, $S2$] which is between [{6}, $S1$] and [{3}, $S3$], covers the input word w_4 to w_5 and so on.

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

(b) *Earley's items*:

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

In the items in (b), number 3 inside the item is the position number, starting from which Earley's items are formed and ending with the dot position stated by the suffix of I_5 or I_6. This indicates that a part of an input sentence from position 3 to 5 ($w_4 w_5$) in the first item has been recognized as "B". In this way a part of the input sentence from position 3 to 6 ($w_4 w_5 w_6$) in the second item has been recognized as "BC" and combined as "A" by applying the 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*:

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

In the case of (c), the 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 a direction opposite to that of Earley's items. Hence we call them dot reverse items (drits), which will be defined at the end of this section. Here, in the first drit, a part of the input sentence from position number 6 down to 5 (w_6) has been recognized as "C" and a part of the input sentence from position 6 down to 3 ($w_4w_5w_6$) in the second drit has been recognized as "BC" and combined as "A" by applying the rule 'x'.

Now let us think the following case of the stack in (d) having two top nodes.

(d) . . . —[{3}, $S3$]———[{5}, $S2$]———[{6}, $S1$] Top (re, x)

: : : —[{2}, $S4$]———[{4}, $S2$]———[{6}, $S1$] Top (re, x)

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

(e) *Earley's Items*:

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

$$I_6 \ni [A \rightarrow BC \cdot\ ,\ 3]$$

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

$$I_6 \ni [A \rightarrow BC \cdot\ ,\ 2]$$

(f) *drits*:

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

$$I_3 \ni [A \rightarrow \cdot BC\ ,\ 6]$$

$$I_4 \ni [A \rightarrow B \cdot C\ ,\ 6]$$

$$I_2 \ni [A \rightarrow \cdot BC\ ,\ 6]$$

Since both top nodes of (d) are exactly the same, let us merge the stack (d) one node left to get the stack in (g).

(g) . . . —[{3}, $S3$]———[{5}, $S2$]———┐[{6}, $S1$] Top (re,x)
: : : —[{2}, $S4$]———[{4}, $S2$]———┘

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

Since the state $S2$ of two nodes immediately left of 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 to the left as shown in (h).

(h) . . . —[{3}, $S3$]———┐[{4, 5}, $S2$]———[{6}, $S1$] Top (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}, $S1$]

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

are popped. In addition to Earley's items shown in (e), the following two Earley's items are also created which we do not want to have.

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

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 trss than (g). In other words, creation of drits enables us to do a deeper merge of TRSS, which is one of the important advantages of creating drits instead of Earley's items. In case 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 a deep merge then it leads to the creation of improper and needless 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 and drits reflects this right-most derivation.

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 (say w_k); 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, w_k, as shown in (f) ($k = 6$ in this case). This enables us to limit the duplication check of drits within the processing of a single input word. In case Earley's items are created, localization of the duplication check is not possible because k is not a constant. Therefore we can localize the range of duplication check of drits.

The following is a formal definition of a drit.

Let $G = (N, T, P, S)$ be a CFG and let $w = w_1 w_2 \ldots w_n \in T^*$ 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 position number i is the location between w_i and w_{i+1}. The special position number '0' represents the left-hand-side position of w_1.

I_i, a set of drits, 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 An Implementation of the YAGLR Algorithm

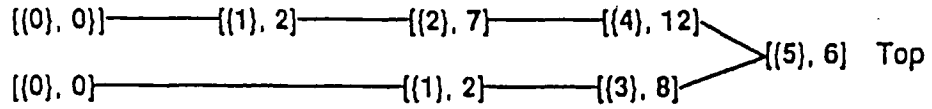
In this section we will give the structure of a TRSS along with shift and reduce actions on TRSS followed by merge operations. In our implementation, each entry in an LR parsing table is regarded as a process which will handle shift, reduce, accept and error actions.

4.1 Structure of TRSS

In parsing, the basic parsing processes are determined by a sequence of state numbers in the stack. Whether or not we use grammatical symbols (as in generalized LR parsing), a 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 is used to create drits during reduce actions. In general, there will be several top nodes in TRSS, but after merging, the remaining top nodes can never be 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 and its list structure are shown in Fig 4. In the trss in Fig. 4, [{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.



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

Fig. 4 *An example of TRSS and its list structure*

4.2 Shift Action

Let us consider a shift action 'sh, u' to a TRSS as shown in (a). It shifts (pushes) a new node onto the top of the TRSS 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) . . .—[{M}, s]————[{i}, t] Top (sh, u)
(b) . . .—[{M}, s]————[{i}, t]]—[{i + 1)}, u] Top
(c) $I_i \ni [X \rightarrow \cdot\, w_{i+1}, i + 1]$

4.3 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 and terminal symbols on its RHS. Applying this rule for the reduce action on (d), the stack (e) is obtained along with the creation of a set of drits as shown in (f).

(d) . . .—[P_k, s_k]————[P_{k+1}, s_{k+1}]——. . .——[P_{k+m}, s_{k+m}] Top
(e) . . .—[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 (e) 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 (d) is {i} which includes

only one position number of the last input word shifted so far. A set of position number P'_{k+m} remains the same as $\{i\}$ after the reduce action.

(f) *Creation of drits*:

$I_a \ni [A \rightarrow \cdot X_1X_2 \ldots X_m, i]$ .....

$I_b \ni [A \rightarrow \cdot X_1X_2 \ldots X_m, i]$ $I_e \ni [A \rightarrow X_1X_2 \ldots \cdot X_m, i]$

..... $I_f \ni [A \rightarrow X_1X_2 \ldots \cdot X_m, i]$

$I_c \ni [A \rightarrow X_1 \cdot X_2 \ldots X_m, i]$ .....

$I_d \ni [A \rightarrow X_1 \cdot X_2 \ldots X_m, i]$ $I_g \ni [A \rightarrow X_1X_2 \ldots \cdot X_m, i]$

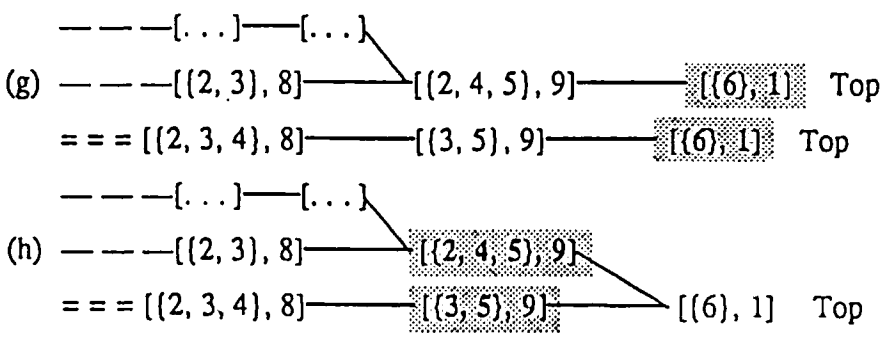
The position number i inside a drit is a position number of the top node in the stack and remains unchanged until the next shift action occurs. Note that a drit such as $[A \rightarrow X_1, \ldots, X_m \cdot , i]$ $(\in I_i)$ is not created because it does not contribute to the formation of trees.

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 the merge operation.

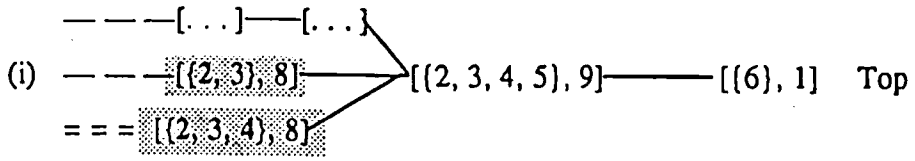
For example, by applying ($M1$) to the TRSS as shown in (g), we get the TRSS as shown in (h).



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

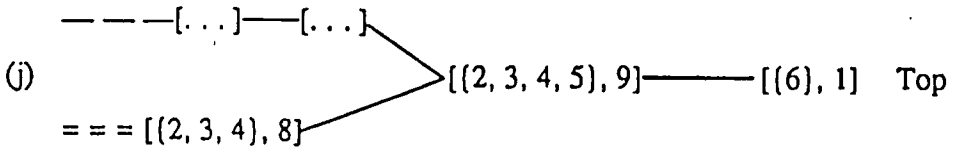
(M21) If M is neither a subset nor a superset of N, a new merged node $[M \cup N, s]$ is formed as a new parent node of X and all the parent nodes of $[M, s]$ and $[N, s]$ will become the parent nodes of the new merged node $[M \cup N, s]$.

For example, in case of (h), 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 (h) is shown in (i).



(M22) If M is a subset of N, then simply take $[N, s]$ as a merged node and merge is not 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 the parent nodes from X to $[N, s]$ are removed. Merge is not necessary beyond this level.

For example, in the case of (i), 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 (*i*) and get the TRSS as shown in (*j*).



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 involving comparison with SAX [6] and SGLR [8]. SAX is based on the bottom up version of the chart algorithm and SGLR is based on Tomita's algorithm using tree-structured stacks.

5.1 Experimental Environment

The experiments were performed on the Sun 3/260 machine and using Quintus Prolog. We used different sets of grammar in our experiment ranging from grammars

with 3 rules to 550 rules to study the parsing efficiency of our algorithm. In this paper we concentrate on four different types of grammar. Gram-1 is a grammar in [3] which is a highly densely ambiguous grammar. For this grammar and its input pattern, readers are requested to refer [3]. Gram-2 is a grammar 44 rules, gram-3 with 123 rules and gram-4 with 400 rules. Gram-2 and gram-4 are taken from [13], and gram-3 from our laboratory 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 centre all our experimental results mainly around gram-4. The results of gram-1 and gram-3 are given in [11].

The inputs for grammars 2, 3 and 4 are made more systematically. The *n*th sentence in the set is obtained by the schema, *noun verb det noun* (*prep det noun*)$^{n-1}$ [13]. The example sentence with this structure is: *I saw a girl on the bed in the apartment with a telescope*. The ambiguity of such sentences grows enormously. Sentences of this type are necessary to find the parsing efficiency *against* sentence ambiguity.

All our programs 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 little (even though we use trss). If we include the gc time, then the ratio between YAGLR and other parsers will vary to a large extent in a positive way to YAGLR. The parsing times in our implementations are without forming trees for SAX, SGLR parsing while YAGLR parsing creates drits.

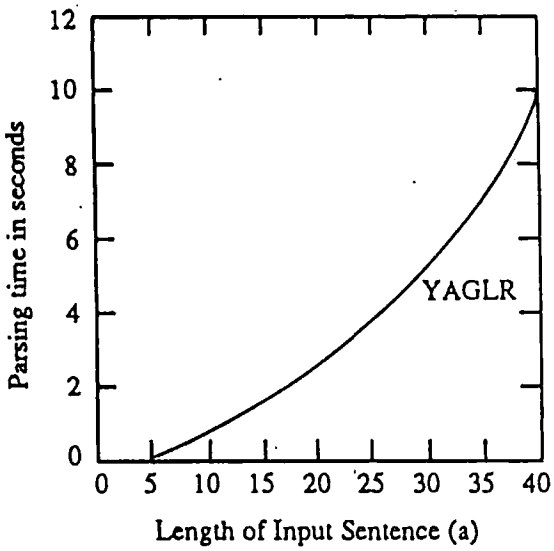
5.2 Experimental Evaluation of YAGLR

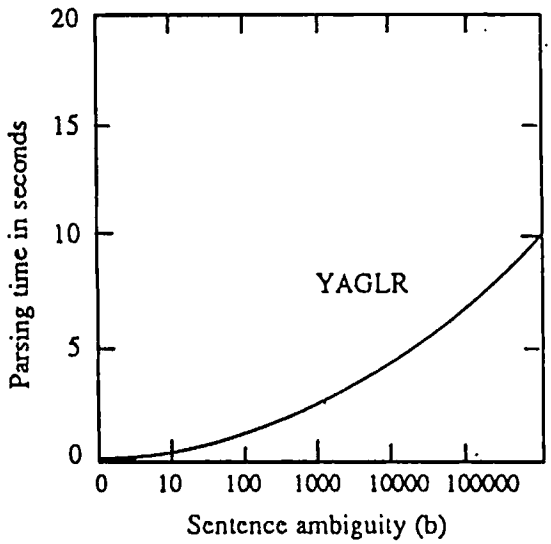
Here, we give our preliminary results on the implementation of YAGLR. Figures 5(a) and (b) show the parsing time of YAGLR for gram-4 against length of the input sentence and against sentence ambiguity respectively. We find that YAGLR parses the sentence faster, as the ambiguity of the sentence increases. 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(c) shows the number of drits created by YAGLR against ambiguity. Here, all the drits created during parsing are indicated by a dashed curve, which includes duplicated drits. After the shift of an input word w_i, our parser makes duplication checks of the drits created in between w_{i-1} and w_i. The other curve shows the number of non-duplicated items created among the duplicated items. Our parsing time shown in Figs. 5(a) and (b) includes the time consumed for removing the duplicated items. If the sentence is ambiguous, the creation of duplicated drits is unavoidable. It should be noted that, if we do not do the duplication check, the YAGLR parser will run faster.

5.3 Comparison with Other Methods

In this subsection, we would like to compare the performance of YAGLR with that


Parsing time in seconds
12
10
8
6
4
2
0
YAGLR
0 5 10 15 20 25 30 35 40
Length of Input Sentence (a)


Parsing time in seconds
20
15
10
5
0
YAGLR
0 10 100 1000 10000 100000
Sentence ambiguity (b)

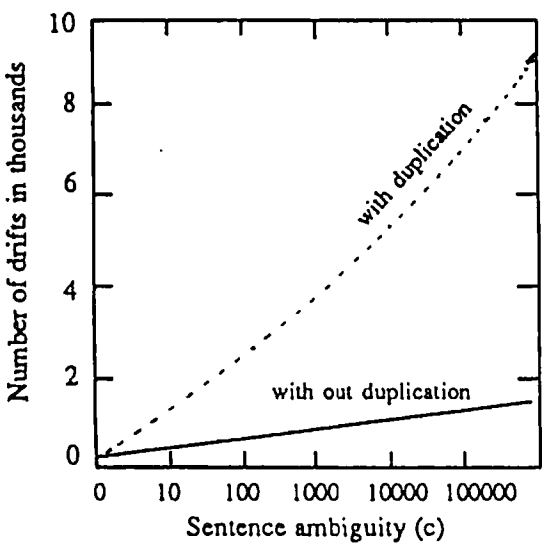

Number of drifts in thousands
10
8
6
4
2
0
with duplication
with out duplication
0 10 100 1000 10000 100000
Sentence ambiguity (c)

Fig. 5 (a, (b) and (c)

of other parsers. In Figs. 6(a) and (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, be it taken against sentence length or ambiguity. The higher the ratio of parsing time of SGLR/YAGLR or SAX/YAGLR, the lower is the parsing time of YAGLR. Here we see that SGLR/YAGLR ratio and SAX/YAGLR ratio are high for a sentence with considerable length, as the ambiguity increases.

Figure 6(c) shows a comparison of Earley's items created using Earley's algorithm and drits created by YAGLR for gram-4. All the duplicated items are removed during parsing and Fig. 6(c) showa the graph of non-duplicated items. From this figure, we can realize the advantages of creating drits rather than Earley's items.

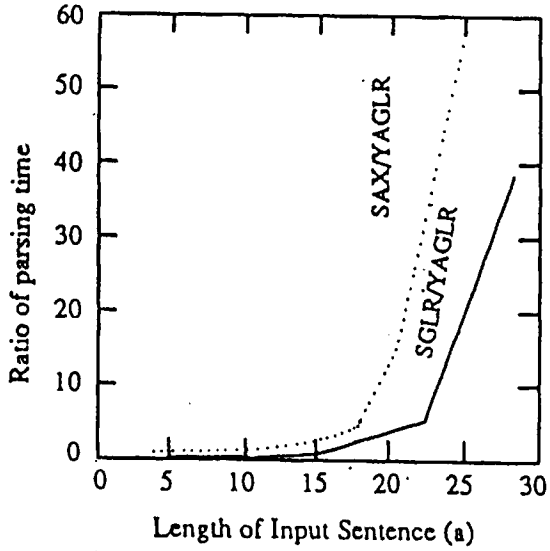
There are some grammars for which the number of non-duplicate Earley's items created using the YAGLR algorithm is less than that of drits. But the total number of items created including duplicated items is far less in the case of drits. The parsing time includes the creation of the total number of items which includes duplicated items. The more the duplicated items, the greater is the time consumed for creating and removing. Also, as we discussed briefly in Section 3, 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.

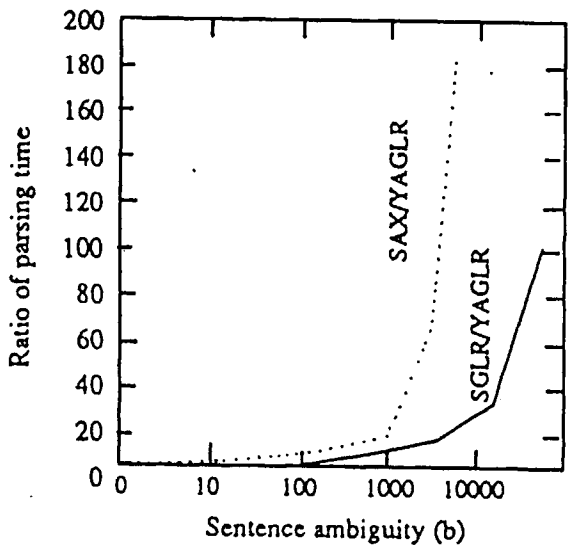
Some raw empirical data obtained from experimental results using gram-1 and gram-4 are given in the table in Fig. 6(d). In the table, I/P denotes length of the input sentence, n denotes sentence number according to the schema described in subsection 5.1 and *Trees* denotes the number of ambiguities. These data entirely depend 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.

Figures 6(e) and (f) show results of memory space consumed by YAGLR for the parsing of gram-2 and gram-4 respectively. YAGLR consumes very little memory space due to its effective merge operations. Note that in YAGLR we create drits, whereas in our experiments SAX and SGLR do not generate partially parsed information in any form. This is the reason why YAGLR needs more space up to a 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 the input sentence increases, the reduction in memory space is remarkable regardless of the size and ambiguity of the grammar.

5.4 Experimental Computational Complexity of YAGLR

For gram-1, we have theoretically proved the complexity of YAGLR to be of the order of n^3 [12]. But we are yet to prove this in the case of general CFG. In this subsection we give our experimental proof for the complexity of YAGLR. Figure 7 shows the order of parsing time of YAGLR for gram-1 and gram-4. On taking log scale for both X and Y axes 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 the X and Y axes in Fig. 5.4 shows the sample line with slope 3. In Fig. 5.4 we find that the time


Ratio of parsing time
Length of Input Sentence (a)
SAX/YAGLR
SGLR/YAGLR
0
10
20
30
40
50
60
0
5
10
15
20
25
30


Ratio of parsing time
Sentence ambiguity (b)
SAX/YAGLR
SGLR/YAGLR
0
20
40
60
80
100
120
140
160
180
200
0
10
100
1000
10000

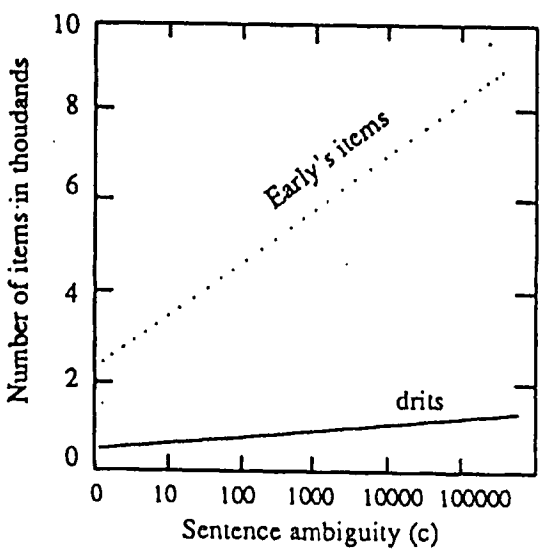

Number of items in thoudands
Sentence ambiguity (c)
Early's items
drits
0
2
4
6
8
10
0
10
100
1000
10000
100000

Fig. 6 (a), (b) and (c)

Gram-1				
VP	*TIME* (ms)			*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	2,867	3,117	517	3,762
10	10,750	12,650	866	14,894
11	41,616	49,716	1,383	59,904
12	262,250	222,235	2,017	24,4088

Gram-4				
n	*TIME* (ms)			*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	3,067	1,000	934	42
6	9,700	3,200	1,417	132
7	32,217	10,683	1,917	429
8	113,135	37,800	2,700	1,430
9	398,832	137,000	3,667	4,862
10	—	498,731	4,750	16,796

Fig. 6(d) *Comparison of empirical result*

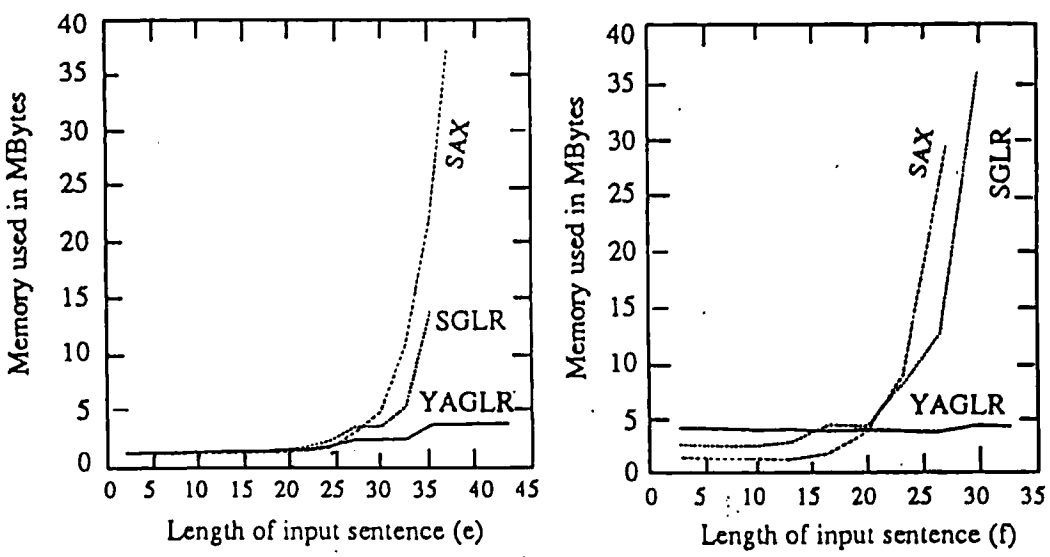


Fig. 6 (e) and (f)

curve of gram-4 is parallel to 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 parallel to the sample line and that it is nearly of the order of n^4. However, as already stated, we have proved theoretically that the complexity of YAGLR for gram-1 is of the order of n^3. From the curve of YAGLR in Figs 5.3(e) and (f) we can conclude that the space complexity of YAGLR is of the order of n^2.

6 Conclusion

We have illustrated the basic idea of the YAGLR parsing algorithm and its implementation, and have also provided an evaluation. It should be noted that, after completing the parsing operation, 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 time and the memory space consumed by YAGLR are very little.

We experimentally proved that our YAGLR parsing algorithm parses a given input sentence much faster. The more the ambiguities in the input sentence and in the grammar, the greater is the parsing speed of YAGLR. We practically proved that, for optionally chosen CFGs with reasonable size and complexity, the time and space complexity of YAGLR are of the order of n^3 and n^2 respectively.

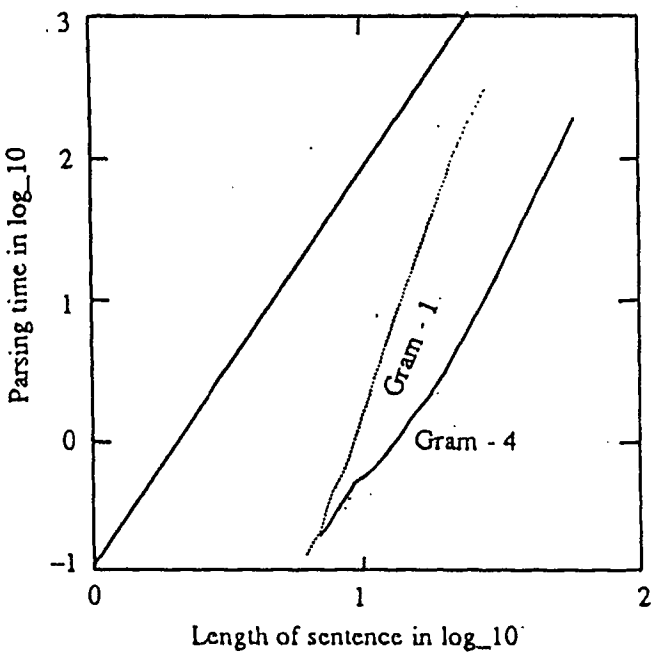


Fig. 7

Since YAGLR is based on the LR parsing algorithm, it is not needed to add the predictor items of Earley's. This reduces the total number of drits created. We also used trss effectively through our merge actions. Even using trss we find that the space consumed by YAGLR is less.

Our future work includes a theoretical proof for the time and space complexity of YAGLR for general CFG, and developing a parallel algorithm for YAGLR method and also for tree generation from drits.

Acknowledgement

The authors would like to thank Natsuki Yuasa, who created certain sets of grammars for the critical testing of the YAGLR parsing algorithm We also thank the anonymous referees for selecting our paper. The first author would like to thank Prof. Hozumi Tanaka for his kind help and fruitful discussions while carrying out this research. A preliminary version of this paper appears as [9], and this is a revised version.

REFERENCES

1. Aho A.V. and Ullman J.D., *The Theory of Parsing and Compiling*, Prentice-Hall, New Jersey, 1972.
2. Earley J., An efficient augmented-context-free parsing algorithm, *Communication of ACM*, vol. 13, no. 1–2, pp 95–102, 1970.
3. Johnson M., The Computational Complexity of Tomita's Algorithm, *Int. Parsing Workshop*, Carnegie-Mellon University, pp 203–208, 1989.
4. Kipps J.R., Analysis of Tomita's Algorithm for General Context-Free Parsing *Int. Parsing Workshop*, Carnegie-Mellon University, pp, 193–202, 1989.
5. Kunth D.E., On the Translation of Languages Left to Right, *Information and control*, vol. 8, no. 6, pp 607–639, 1965.
6. Matsumoto Y., Natural Language Parsing Systems Based on Logic Programming, *Ph.D. Thesis*, Kyoto University, Kyoto, Japan, 1988.
7. Numazaki H and Tanaka, H., A New Parallel Algorithm for Generalized LR Parsing, *COLING'* 90, vol. 2, pp 305–310, 1990.
8. Numazaki, H. and Tanaka, H., *SGLR: A Sequential Generalized LR Parser in Prolog*, Journal of Information Processing Society of Japan, vol. 32, no. 3, pp 396–403, 1991.
9. Suresh K.G. and Tanaka H., Implementation and evaluation of yet another generalized LR parsing algorithm, *Proc. Indian Computing Congress*, pp 506–515, Tata McGraw-Hill, New Delhi, 1991.
10. Tanaka H. and Numazaki H., *Parallel generalized LR parser based on logic programming*, *1st Australia-Japan Joint Symp. Natural Language Processing*, pp 201–211, 1989.
11. Tanaka H. and Suresh K.G., YAGLR: Yet another generalized LR parser, *Proc. ROCLING IV*, Republic of China, pp 21–31, 1991.
12. 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).
13. Tomita M., Efficient Parsing for Natural Language, Kluwer Academic Press, Boston, Mass., 1986.
14. Tomita M., An efficient augmented-context-free parsing algorithm, *Computational Linguistics*, vol. 13, no 1–2 pp 31–46, 1987.
15. Winograd, T., Languages as a Congnitive Process, vol. 1: syntax, Addison-Wesley, 1983.