TR-0782

Drit Parser: A Generalized LR Parsing Algorithm Using Dot Reverse Item

by
H. Tanaka, K. G. Suresh & H. Numazaki
(Tokyo Institute of Technology)

June, 1992

© 1992, ICOT

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
(03)3456-3191 ~ 5
Telex ICOT J32964
Minato-ku Tokyo 108 Japan

Institute for New Generation Computer Technology

Drit Parser : A Generalized LR Parsing Algorithm Using Dot Reverse Item[1]

Hozumi TANAKA K. G. SURESH Hiroaki NUMAZAKI
Department of Computer Science, Tokyo Institute of Technology
2-12-1 Ôokayama Meguro-ku Tokyo 152, Japan
Email : {tanaka, suresh, numazaki}@cs.titech.ac.jp

Abstract

We have developed a new generalized LR parser called *Drit parser*. The parsing algorithm is based on Kipps recognition algorithm but it creates the parsing results as a set of dot reverse items (*drits*). The structure of drits is symmetric to the one of Earley's items. It is possible to form parsing trees from a set of drits created after the completion of parsing. This makes Kipps recognition algorithm a practical CFG parser. In order to build a Drit parser, some modifications in the original version of Kipps recognition algorithm are required. The time and space complexity of Drit parser is in the order of n^3 and n^2 respectively, since Drit parser is based on Kipps recognition algorithm. We will conclude that Drit parser has both the advantages of Earley's and Tomita's parsing algorithm.

1 Introduction

Some compilers of programming languages have made use of the LR(k) parsing algorithm devised by Kunth [6] which enables us to parse an input sentence deterministically and efficiently. However the grammars used in this algorithm are limited to LR(k) grammars so that Context-Free Grammars (CFG) in general can not be handled.

Tomita extended the LR(k) parsing algorithm to handle CFG [11,12]. The extended algorithm is one of the generalized LR parsing algorithms. Empirical results of Tomita's and Earley's algorithm reveal that the Earley/Tomita ratio of parsing time is larger when the length of an input sentence is shorter or when an input sentence is less ambiguous [11]. In [5] the time and space complexity of Tomita's algorithm is proved to be in the order of $n^{1+\rho}$ and n^2 respectively for general CFG. Here ρ is the maximum length on right hand side (rhs) of the production rules used. Thus if $\rho > 2$, Tomita's algorithm dose not fare well theoretically compared to Earley's algorithm.

[1]This paper has been submitted to the Journal of Computational Linguistics

Kipps gave a modified version of Tomita's algorithm, which works with the time and space complexity in the order of n^3 and n^2 for any CFG. However, Kipps algorithm is only a recognition algorithm and is not a practical parsing algorithm because, it can not generate syntactic structures [8].

Tomita's algorithm creates partially parsed trees during parsing process. For a CFG with dense ambiguities, exceeding more than n^3, Tomita's algorithm makes the parsing time theoretically worse than Earley's algorithm. On the other hand the Earley's algorithm creates items which can be shared by many parsing trees. This is the reason why Earley's algorithm performs theoretically better than Tomita's in case of densely ambiguous grammars.

The fact suggests that it seems to be possible to create the Earley's items during the recognition process in Kipps recognizer. In this paper we will show that it is not appropriate to create Earley's items. However fortunately we can create another form of items called *dot reverse items* (drit), which are in symmetrically different form from Earley's items. The data structure called drit has been introduced in [10] and [9] showed the effectiveness of a parser based on drit formation. It is possible to form parsing trees from a set of drits created after the completion of parsing. This makes Kipps recognition algorithm a practical CFG parser, which in this paper we call as Drit parser. In order to build a Drit parser, some modifications in the original version of Kipps recognizer are required. The time and space complexity of Drit parser, as Kipps recognizer, is in the order of n^3 and n^2 respectively. We will conclude that Drit parser has both the advantages of Earley's and Tomita's parsing algorithms.

In section 2 we will give a formal definition of a drit. In section 3, we will describe the algorithm of Kipps recognizer which is based on Tomita's algorithm. In section 4, we will explain Drit parser and give the algorithm based on Kipps recognizer. We will also discuss the reasons for creating drits instead of Earley's items. Finally we give our conclusion in section 5.

2 Dot Reverse Item

In this section[2] we will give a formal definition of a drit, the intuitive meaning and the purpose of which will be explained in detail in section 4.1.1.

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 $X \rightarrow Y_1\ ...\ Y_m$ and $0 \leq j \leq n$, $[X \rightarrow Y_1\ Y_2\ ...\ Y_k \cdot Y_{k+1}\ ...\ Y_m, j]$ is called a drit for w. The dot between Y_k and Y_{k+1} is a metasymbol not in N and T. The suffix i in the input sentence is called the *position number* in the following sections. The special position number '0' and 'n' represents the left and right hand side position of w_1 and w_n respectively.

[2]The reader can skip this section in the first reading

R_i, a set of drit is defined as follows. For i and j ($0 \leq i \leq j \leq n$), $[X \rightarrow \alpha \cdot \beta, j]$ $\in R_i$ iff $S \stackrel{*}{\Rightarrow} \gamma\ X\ \delta$, $\beta \stackrel{*}{\Rightarrow} w_{i+1}w_{i+2}\ldots w_j$, and there exists h ($0 \leq h \leq i$) such that $\gamma \stackrel{*}{\Rightarrow} w_1 w_2 \ldots w_h$ and $\alpha \stackrel{*}{\Rightarrow} w_{h+1}w_{h+2}\ldots w_i$.

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. In other words, in case of drit, the position immediately after β is given as j in drit's data structure, and the portion of the input sentence between j and i ($j > i$) is accepted as β.

3 An Overlook of Tomita and Kipps Recognition Algorithms

In this section after a brief explanation of Tomita's algorithm as a recognizer, we will explain Kipps recognizer. We assume the familiarity of generalized LR parsing algorithm. Because of the simplicity we will not consider the ϵ rule until 4.1.2

3.1 Tomita's Algorithm As a Recognizer

Following the notation given by Kipps, a graph-structured stack (GSS) is a single directed acyclic graph. Each stack element corresponds to a vertex of a graph which may have more than one parent. The root of the graph is a common bottom-of-stack. Each leaf of the graph represents a distinct top-of-stack. The leaves of the graph grow in stages. Each stage U_i corresponds to the i-th word w_i of the input sentence. Upon scanning w_{i+1}, the recognizer carries out mainly the following two actions in stage U_i. (1) add an additional leaf to U_i (reduce action) or (2) add a leaf to U_{i+1} (shift action). What kinds of actions are carried out is determined by the leaves in U_i, LR table, and the preterminal of scanning word w_{i+1}. Only after all the leaves in the stage U_i has been processed, the recognizer proceeds to the stage U_{i+1} scanning the next word w_{i+2}.

Each vertex of the GSS in U_i is a triple $< i, s, L >$, where i is an integer denoting the i-th word scanned, s is the parse state and L is a set of parent vertices. The recognizer works as follows. A GSS is initialized by pushing $< 0, s0, \{\} >$, which becomes the root of the GSS with w_1 as a look-ahead word (scanning word), whose preterminal will be used to determine the parsing actions in the LR table.

A LR table consists of two fields, an ACTION field and a GOTO field. The parsing actions are determined by state (the row of the table) and a look-ahead preterminal (the column of the table) of an input sentence. End of the sentence is represented by '-|'. There are four types of actions: shift, reduce, accept and error. Some entries in the LR table contain more than two actions and are thus in conflict. In such cases, a

recognizer must conduct more than two actions in parallel.

1. Reduce : The recognizer pops the number of vertices (corresponding to the rhs of the production rule specified by the reduce action) from the top of the stack and then creates a new leaf in U_i which becomes active and the state of which will be determined by the GOTO field.

2. Shift : A new leaf corresponding to a look-ahead word w_{i+1} is pushed in U_{i+1}. The state of the leaf is determined by the shift action. Note that the newly created leaf in U_{i+1} is not active until there is no active leaf remained unprocessing in U_i and the look-ahead word becomes w_{i+2}.

3. Error : The leaf with error action will be truncated.

4. Accept : Recognition process will end with success.

In case of 1 and 2, a new leaf is added in U_i and U_{i+1} respectively and edges are formed from the new leaf to its parents. If there exists a leaf with the same state as that of a newly created leaf in U_i in case of 1, or U_{i+1} in case of 2, then they will be merged into one. The leaf after merge will have several parents. Merging of same leaves avoids the duplicated processing of input sentence and also it makes sure the number of vertices in each stage to be within the number of total states in LR table. Hence the order of the number of vertices in each stage becomes constant.

Let us focus more on reduce action in a stage U_i. A reduce action pops the number of vertices (say q) equal to the number of nonterminal and preterminal symbols in rhs of the rule used in the reduce action. Then the ancestor vertices at a distance of q will tentatively become the top of the stack, and using the GOTO field of LR table, a new leaf is pushed in the stage U_i for every ancestors. At the same time new edges are formed from the leaves to the ancestors and the vertices in the distant stages becomes the parents of the leaves. In consequence, a vertex $< i, s, L >$ in U_i has at most c*i parents where c is the number of total states and is constant. We can conclude that the number of ancestors of each vertex is in the order of i.

The ancestors at a distance of q from a leaf in the stage U_i will be obtained by traversing every path from the leaf to them. As the number of parents is in the order of i, the number of paths between the leaf and the ancestors at a distance of q becomes at most i^q. Therefore, the time of Tomita's recognizer is in the order of $n^{1+\rho}$ ($= \sum_{i=1}^{n+1} i^\rho$, where ρ is the number of nonterminal and preterminal symbols in the rhs of the longest production). If $\rho > 2$, the order crosses over n^3. For the grammars in Chomsky normal form $\rho = 2$ and hence the order of recognition time becomes n^3.

3.2 Kipps Recognizer

The reason for Tomita's recognizer crossing over n^3 order of parsing time is due to the time consumed in getting the ancestors. Although the number of ancestors at a

distance of q from a leaf in the stage U_i is in the order of i, the time needed to extract them is in the order of i^q. The reason is that, to get the ancestors, an edge once traversed can be again traversed repeatedly. If the duplicated traversal of the edge is prevented, the time of getting the ancestors can be reduced.

In order to prevent the duplicated traversal of edges, Kipps changed the data structure of the vertex as $< i, s, A >$. Here i represents the position number of the shifted word, s the state and A is the ancestors table which is a set of tuples such that $\{< k, L_k > | k = 1, 2, \cdots, \rho\}$ where L_k is a set of ancestors at a distance of k from the vertex $< i, s, A >$. For example, the set of parents L in section 3.1 equals to L_1. From the above discussion, we know that the ancestors table is formed by at most ρ tuples and the number of ancestors in L_k is in the order of i. Once an entry in an ancestors table is filled, the time to reget that entry is constant thereafter. In other words, time to get the ancestors becomes constant. The ancestors table can be formed in a constructive way using the ancestors tables formed in the past.

Following is the definition of a function ANCESTORS, which is a table look-up function that generates table entries the first time they are requested [5].

```
ANCESTORS( v =  <i, s, A>, k )
  if   k = 0
       return ( { v } )
(1) else  if ∃  <k, L_k> ∈ A
            return( L_k)
(2)       else
            let  L_k :=  ∪ v ∈ L_1 | < 1, L_1 > ∈ A    ANCESTORS( v, k-1)
            let  A :=  A ∪ { <k, L_k >}
            return ( L_k)
```

The L_1 in the element $< 1, L_1 >$ of the ancestors table A of the vertex is a set of parent vertices and note that L_1 is filled up when the vertex is formed. The portion of (1) and (2) in the body of ANCESTORS function have the following roles. (1): If $< k, L_k > (k > 0)$ is in A then those vertices are returned. (2): Otherwise for all parent vertices v in L_1, ANCESTORS(v, k-1) is called recursively to generate all the vertices at a distance of k from v. Before returning those vertices they are recorded in the ancestors table of v. Once the ancestors table is filled then the vertices at a distance of k from v can be obtained directly from the ancestors table in a constant time since all the vertices have been recorded in A, and is not needed to call ANCESTORS recursively. In order to make effective computation during reduce action, we modify this ANCESTORS function to be just a table look-up function as shown in section 4.1.2.

For ANCESTORS($< i, s, A >, k$), time bound in the worst case is in the order of i^2 [5]. On considering the constant time to call the ANCESTORS once it has been

called, for the input sentence length n, Kipps recognition algorithm takes the time in the order of n^3. We suggest our redears to refer [5] for detailed understanding of Kipps recognition algorithm.

4 Drit Parsing Algorithm

The Drit parsing algorithm is almost the same as Kipps recognition algorithm with some modifications in shift and reduce actions. The modifications are related to the formation of drits during shift and reduce actions. The GSS used in our Drit parser is the same as the one used in Kipps recognition algorithm. In section 4.1, after discussing the reasons for creating drits instead of Earley's items, we will modify the Kipps ANCESTORS function, and then give a complete algorithm of reduce action in Drit parser. Section 4.2 gives a shift algorithm along with a top level algorithm of Drit parser.

4.1 Reduce Action

Let us assume that all the production rules in a CFG are numbered. Let the p-th rule which is used for a reduce action be,

$$D_p \rightarrow C_{p1}\ C_{p2} \cdots C_{pq}$$

and the stack top (leaf) experiencing this reduce action be $< i, s, A >$. Then, the leaf corresponds C_{pq}, its parents correspond C_{pq-1} and in the same way, the vertices at a distance of k from the leaf correspond C_{pq-k}.

As well as Tomita's algorithm, applying this p-th rule, we can produce the partially parsed trees corresponding to the p-th rule. Instead of partially parsed trees, Drit parser creates drits which are parts of the parse trees. The reader may think that at this time Earley's items can also be created, but the problems in that are explained in 4.1.1. In 4.1.2 we give the definition of the reduce action performed by Drit parser.

4.1.1 Purpose of Dot Reverse Items

Using the ancestors table in the leaf alone, we can create proper drits during reduce actions. However sometimes it is not possible to create proper Earely's items from the ancestors table of the leaf. We will explain this fact through some concrete examples. The fact that we can create proper drits from the ancestors table of the leaf alone, plays an important role in computing the complexity of Drit parser. The reason will be explained in 4.4.

Let the input sentence be $w_1 w_2 \cdots w_n$. For the clear understanding of the concept of position number, we first consider the following stack in which we show only the position numbers and hide other informations such as state and ancestors table.

(a) $\cdots$—<3,..,..>—<5,..,..>—<6,..,..> top reduce by X → Y Z

Here, $< 6, .., .. >$ is the top of the stack, which is a leaf. The position number inside the vertex means that, the input sentence up to that position number has been processed (shifted). Accordingly, the position number 6, in this case means that the input sentence up to the word w_6 has been processed. In a similar way the vertex $< 5, .., .. >$ indicates the processing of the input sentence up to w_5. From this fact we can realize that the vertex $< 6, .., .. >$ covers a part of the input sentence w_6, the vertex $< 5, .., .. >$ in between $< 3, .., .. >$ and $< 6, .., .. >$ covers w_4 w_5.

When applying a reduce action "reduce by X → Y Z" to the stack (a), the vertex $< 6, .., .. >$ matching to Z and the vertex $< 5, .., .. >$ matching to Y are popped and the drits as shown in (b) are created.

(b) Drits: $R_5 \ni [\, X \rightarrow Y \cdot Z,\ 6],$ $R_3 \ni [\, X \rightarrow \cdot\ Y\ Z,\ 6]$

In case of (b), number 6 inside each item is the position number appeared in the leaf of the stack (a). In the first drit, a part of the input sentence from position number 6 down to 5 (w_6) has been recognized as "Z" and in the second drit, a part of the input sentence from position 6 down to 3 ($w_4 w_5 w_6$) has been recognized as "Y Z" and combined as "X" by applying the rule used for the reduce action. The number 6 inside each item is the position number, starting from which drits are created and ending with the dot position stated by the suffix of R_3 and R_5.

Similarly, from (a) we can also create Earley's items as shown in (c).

(c) Earley's items: $E_5 \ni [\, X \rightarrow Y \cdot Z,\ 3],$ $E_6 \ni [\, X \rightarrow Y\ Z \cdot\ ,\ 3]$

In each item in (c), number 3 inside the item is the position number, starting from which Earley's items are created and ending with the dot position stated by the suffix of E_5 and E_6. This indicates that a part of the input sentence from position 3 to 5 ($w_4 w_5$) in the first item has been recognized as "Y". 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 "Y Z" and combined as "X" by applying the reduction rule.

Next, consider a merged stack as in (d) and using only the ancestors table A of the leaf, let us create drits and Earley's items.

(d) $\cdots$—<3,..,..>—<5,..,..>—<6,s,A> top reduce by X → Y Z
$\cdots$—<2,..,..>—<4,..,..>-
here $A = \{<1, \{v5, v4\}>, <2, \{v3, v2\}>, \cdots\}$
where $v5 = <5,..,..>$, $v4 = <4,..,..>$, $v3 = <3,..,..>$,
$v2 = <2,..,..>$, $\cdots$.

We perform the reduce action "reduce by X → Y Z" on (d). Using this rule for reducing we have to pop two vertices from the leaf $< 6, s, A >$. The vertices at a distance of 1 from the leaf are $v5$ and $v4$, whose position numbers are 5 and 4 respectively. Again, the vertices at a distance of 2 from the leaf are $v3$ and $v2$, whose position numbers are 3 and 2 respectively. It is important to note that all the position

numbers of the vertices at distance of 1 and distance of 2 from the leaf can be obtained by only looking at the ancestors table A in the leaf alone. Thus using the ancestors table A's information alone we can create drits as shown in (e).

(e) Drits:

$R_5 \ni [\ X \rightarrow Y \cdot Z,\ 6]$, $\quad R_4 \ni [\ X \rightarrow Y \cdot Z,\ 6]$,
$R_3 \ni [\ X \rightarrow \cdot\ Y\ Z,\ 6]$, $\quad R_2 \ni [\ X \rightarrow \cdot\ Y\ Z,\ 6]$

These are certainly the same drits created by traversing each path from the leaf down to root.

Similarly, using the ancestors table A's information alone we create Earley's items as shown in (f).

(f) Earley's items:

$E_5 \ni [\ X \rightarrow Y \cdot Z,\ 3]$, $\quad E_5 \ni [\ X \rightarrow Y \cdot Z,\ 2]$,
$E_4 \ni [\ X \rightarrow Y \cdot Z,\ 3]$, $\quad E_4 \ni [\ X \rightarrow Y \cdot Z,\ 2]$,
$E_6 \ni [\ X \rightarrow Y\ Z \cdot ,\ 3]$, $\quad E_6 \ni [\ X \rightarrow Y\ Z \cdot ,\ 2]$

In the Earley's items in (f), when compared to the Earley's items obtained by traversing each path from the leaf down to root, we get two additional items as shown.

$E_5 \ni [\ X \rightarrow Y \cdot Z,\ 2]$, $\quad E_4 \ni [\ X \rightarrow Y \cdot Z,\ 3]$

From the information in the ancestors table of the leaf alone, there is no way to exclude both the combination of the position numbers 2 and 5, and 3 and 4 which are not allowed in case of creating Earley's items.

In conclusion, by using the information in the ancestors table of the leaf alone, we are able to create necessary and proper drits, whereas in case of creating Earley's items unnecessary and unproper items are being created. The reason for this difference comes from the fact that LR parsing is based on the right-most derivation and drits reflects this derivation.

Another important purpose in using drits is the localization of duplication checks. The position number inside the drits in (e) will remain the same throughout the processing of the stage U_6. This enables us to limit the duplication check of drits created within the stage U_6.

4.1.2 Algorithm for Creating drits

In this section, we give an algorithm for creating drits during the reduce action followed by the definition of reduce action. In the previous section we showed that from the ancestors table A in the leaf alone it is possible to create drits. Now let us consider the p-th production rule used during the reduce action as

$$D_p \rightarrow C_{p1}\ C_{p2} \cdots C_{pq-k}\ C_{pq-k+1} \cdots C_{pq}$$

In this case, we can create drits from the algorithm given below in stage U_i.

for k from 1 to q
  for $\forall\ j'$ $s.t.$ $< j',\ s',\ A' > \in$ ANCESTORS($v,\ k$)
    let $R_{j'} := R_{j'} \cup \{[\ D_p \rightarrow C_{p1}\ C_{p2} \cdots C_{pq-k} \cdot C_{pq-k+1} \cdots C_{pq},\ i]\}$

Note that when $k = q$, the drit such as $[D_p \rightarrow \cdot\ C_{p1}\ C_{p2} \cdots C_{pq},\ i]$ will be created. The definition of the reduce action which includes the above algorithm is given after the introduction of NEW function and modified ANCESTORS function.

In order to create all the possible drits, it is necessary to modify Kipps ANCESTORS function. If we use Kipps ANCESTORS function, after requesting an entry by calling ANCESTORS(v,q), only this entry is guaranteed to be filled in and the other entries between *1* and *q-1* are not always filled in. However as explained above, all the entries up to q are necessary to create drits during reduce actions. For the purpose of filling all the entries up to q, we introduce a function NEW and we make the ANCESTORS function as just a table look-up function without generating any table entries when they are requested. When a new leaf is created the function NEW is called to carry over the ancestors table of its parent. The definition of NEW function is shown below.

```
NEW( v = <i, s, A>, u = <j, t, B>)    % u is a parent of v.
(1)     for k from 2 to ρ
(2)          if ∃ < k-1, L'_{k-1} > ∈ B
                  if ∃ < k, L_k > ∈ A
                       let L_k := L_k ∪ L'_{k-1}
                  else
                       let A := A ∪ { <k, L'_{k-1} >}
             else
                  return(v)
        return(v)
```

The above NEW function fills all necessary entries of the ancestors table A of the leaf v by getting the entries of the ancestors table of its parent u. For getting all the possible drits it is enough to fill ρ entries of the ancestors table, which are set by the iteration given in line (1). Here ρ is the rhs of the longest production rule used. The line (2) checks whether an entry to be carried exists in the parent's ancestors table. If exist, then that entry is carried to the leaf v.

As we defined the NEW function we now give the modified ANCESTORS function, which becomes just a table look-up function because all the entries of an ancestors table A have already been filled in when ANCESTORS function is called.

```
ANCESTORS( v = <i, s, A>, k )
  if  k = 0
       return ( {v} )
  else if ∃ <k, L_k > ∈ A
            return( L_k)
```

It is clear that the above modifications will not affect the time complexity, since the modified ANCESTORS function can be performed in a constant time and the NEW function can be performed at the maximum in the order of i^2.

The definition of reduce action is shown below. Here the portion in the box refers to the creation of drits and filling of the ancestors table which are different from the original Kipps recognition algorithm.

REDUCE (v, p)

for k from 1 to q
  for $\forall\ j'\ s.t.\ < j',\ s',\ A' > \in$ ANCESTORS(v, k)
    let $R_{j'} := R_{j'} \cup \{[\ D_p \rightarrow C_{p1}\ C_{p2} \cdots C_{pq\ k} \cdot C_{pq-k+1} \cdots C_{pq},\ i]\}$

(1) for $\forall\ v1' = < j',\ s',\ A1' >\ s.t.\ v1' \in$ ANCESTORS(v, q)
  let $s'' :=$ GOTO(s', D_p)
(2) if $\exists\ v'' = < i-1,\ s'',\ A'' >\ s.t.\ v'' \in U_{i-1} \wedge < 1,\ L'' > \in A''$
(3) if $v1' \in L''$
  do nothing (ambiguous)
  else
(4) if $\exists\ v2' = < j',\ s',\ A2' >\ s.t.\ v2' \in L''$
  let $vc'' := < i-1,\ s'',\ Ac'' >\ s.t.\ Ac'' = \{< 1,\ \{v1'\} >\}$
  let $vc'' :=$ NEW(vc'', $v1'$)
  for $\forall$ *'re p'* $\in$ ACTIONS(s'', w_i), REDUCE(vc'', p)
  else
  let $L'' := L'' \cup \{v1'\}$, let $v'' :=$ NEW(v'', $v1'$)
(5) if $v'' \in$ P
  let $vc'' := < i-1,\ s'',\ Ac'' >\ s.t.\ Ac'' = \{< 1,\ \{v1'\} >\}$
  let $vc'' :=$ NEW(vc'', $v1'$)
  for $\forall$ *'re p'* $\in$ ACTIONS(s'', w_i), REDUCE(vc'', p)
(6) else
  let $v'' := < i-1,\ s'',\ A >\ s.t.\ A = \{< 1,\ \{v1'\} >\}$
  let $v'' :=$ NEW(v'' , $v1'$)
  let $U_{i-1} := U_{i-1} \cup \{v''\}$

As shown above, Drit parser uses the reduce action almost the same as Kipps. Following the way of Kipps explanations, (1) iterates through all the ancestor vertices at a distance of q from v, setting s'' to the new state indicated in the GOTO table under D_p given the ancestor's state s'. (2) checks whether such a vertex v'' with the same state s'' already exists then (3) checks that a shift from the current ancestor $v1'$ has already been made. In this case do nothing. (4) checks whether $v1'$ is a clone vertex created by a ϵ rule.[2] If $v1'$ is a clone, v'' is again cloned into vc'' and all reduce

[2]On using ϵ rule for reduce action, the ANCESTORS will return a clone vertex as the ancestor of itself. More detailed discussions on clone vertex is given in [5].

actions executed on v'' are executed on the new clone vc''. Otherwise (5) checks if v'' has already been processed, if so, then it missed any production through $v1'$, so v'' is cloned into vc'' and all reduce actions executed on v'' are now executed on vc''. Whenever $v1'$ becomes a parent of v'', the ancestors table in v'' will be updated by using the ancestors tables in $v1'$, which is performed by calling the function NEW. (6) adds a vertex with state s'' to the stage U_{i-1}.

4.2 Shift Action

Let us consider that the parser is going to enter in the stage U_i from the stage U_{i-1} by shifting a look-ahead word w_i. If we assume C be the preterminal of the word w_i, during the shift action a drit [C $\rightarrow \cdot\ w_i$, i] is created in R_{i-1}.

$$R_{i-1} := R_{i-1} \cup \{[\ C \rightarrow \cdot\ w_i,\ i]\}$$

The reason for including the newly created drit in the set R_{i-1} is that, at the time just before shifting of the word w_i, the active leaves have the position number i-1. Only after shifting the word w_i for all the leaves, the top position number will be incremented by one and the processing enters the new stage U_i. We give the definition of the shift action in Drit parser as shown below. The portion in the box refers to the creation of drits and filling of the ancestor table which are different from the original Kipps recognition algorithm.

SHIFT(v, s)
  $R_{i-1} := R_{i-1} \cup \{[C \rightarrow \cdot\ w_i,\ i]\}$
  if $v1 = <i,\ s,\ A>$ $s.t.$ $v1 \in U_i \wedge <1,\ L> \in A$
    let $L := L \cup \{v\}$
  else
    let $v1 := <i,\ s,\ A>$ $s.t.$ $A = \{<1,\ \{v\}>\}$
    let $v1 :=$ NEW($v1$,v)
    let $U_i := U_i \cup \{v1\}$

We finally give the top level algorithm of Drit parser as follows.

PARSE($w_1, w_2, \cdots, w_{n+1}$)
  let $w_{n+1} :=$ -| % end of a sentence
  let $U_i := \{\}$ ($0 \le i \le n$)
  let $U_0 := \{ <0,\ s0,\ \{\}>\}$
  for i from 1 to $n+1$
    let $P := \{\}$
    for $v = <i\text{-}1,\ s,\ A>$ $s.t.$ $v \in U_{i-1}$
      let $P := P \cup \{v\}$
      if $\exists$ '$sh\ s'$' $\in$ ACTIONS(s, w_i), SHIFT(v, s')
        for $\forall$ '$re\ p'$' $\in$ ACTIONS(s, w_i), REDUCE(v, p)
        if 'acc' $\in$ ACTIONS(s, w_i), accept
    if $U_i = \{\}$, reject

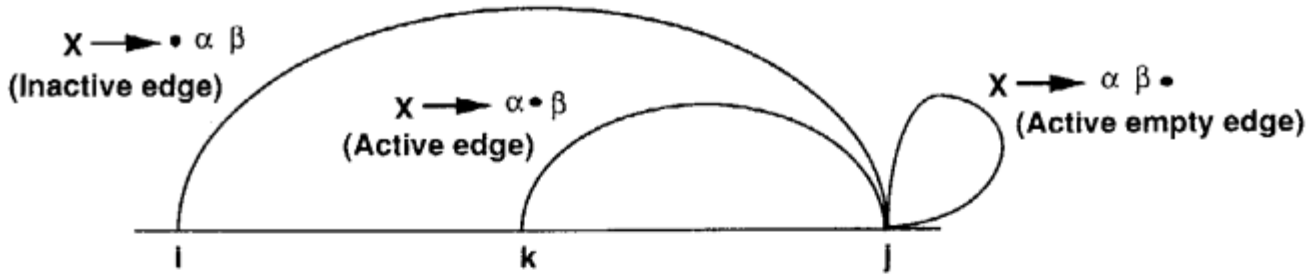
4.3 Computational Complexity of Drit Parser

Let us consider the time complexity in creating drits in the stage U_i. This can be found out by taking into account of, first, the time of the evaluation of ANCESTORS and NEW functions given in 4.1.2, and secondly, the number of vertices picked out from the ancestors table. In our Drit parser the ANCESTORS and NEW functions jointly can be performed in the order of i^2 since the NEW function can be performed in the order of i^2 and ANCESTORS in a constant time. As the number of vertices picked out from the ancestors table is in the order of i, the time complexity for Drit parser, for an input sentence of length n is in the order of n^3, same as that of Kipps recognizer.

To find the space complexity of Drit parser, we have to consider the memory space consumed by GSS and by the number of drits created. It is obvious that the space consumed by GSS is in the order of n^2. The number of drits is proportional to the number of vertices picked out from the ANCESTORS. For the stage U_i, this is in the order of i. This can be repeated for n times and hence the order becomes n^2. Thus the space used by Drit parser is in the order of n^2.

4.4 A Comparison With Chart Parser

In this section we correlate drits with the charts of Chart parser [4] as below.



Usually the chart is read from left to right, but the above chart should be read from right to left. For the active edge, inactive edge and active empty edge of the chart, we give corresponding drits as follows.
Corresponding drits :

- For the active edge $\Rightarrow R_k \ni [X \rightarrow \alpha \cdot \beta, j]$ $(k < j)$.
- For the inactive edge $\Rightarrow R_i \ni [X \rightarrow \cdot\alpha\beta, j]$ $(i < j)$.
- For the active empty edge $\Rightarrow R_j \ni [X \rightarrow \alpha\beta\cdot, j]$.

The reader can understand two important advantages of Drit parser compared to Chart parser.

1. Drit parser does not create any active empty edges[3] which do not contribute to form parse trees.

2. The parsing process in Drit parser is regulated by LR table and hence unnecessary parsing operations are avoided.

Due to the above advantages, our Drit parser will make more effective parsing compared to Chart parser.

5 Conclusion

For certain CFG it was found that, the time complexity of Tomita's generalized LR parsing algorithm is more than that of Earley's algorithm [3, 5]. Kipps gave a recognition algorithm in which he made small modifications in Tomita's algorithm. The time complexity of the modified recognizer is the same as that of Earley's (n^3 where n is the length of the input sentence) for any CFG [5]. However, Kipps algorithm only recognizes the input sentence as grammatically acceptable or not and it does not produce any parse trees. For this reason, Kipps algorithm can not be taken as a practical parser [8].

In this paper we proposed a new generalized LR parsing algorithm called Drit parser using a data structure called drit. Drit parser creates drits during the parsing process. For this purpose we made some modifications in Kipps recognition algorithm, so that it will become a practical parser by giving all the possible parses. Since Drit parser is based on Kipps recognition algorithm, the time and the space complexity of Drit parser is in the order of n^3 and n^2 respectively. Thus, Drit parser maintains the advantages of Earley's and Tomita's algorithms.

We give the followings as our future research works.

- The practical evaluation of Drit parser.
- Developing a parallel algorithm.
- A parallel algorithm for tree generation from drits.

Finally, the authors like to give the correlation with YAGLR, which is a new generalized LR parsing algorithm proposed by the authors [10]. Actually drits were introduced in YAGLR parser. It is evident from [9] that, the experimental time complexity of YAGLR is in the order of n^3 and further on using tree-structured stack instead of GSS, the merge of tree-structured stack is more deep which makes the memory space used by YAGLR less. From the empirical results it is known that if the ambiguity of the sentence is extremely high, YAGLR is much faster than the tree-structured stack version of Tomtia's algorithm. However, for generalized CFG, the time and space

[3]The active empty edges refers to the predictor items created by Earley's algorithm [2].

complexity of YAGLR are not yet to be proved. In any case, Drit parser should be practically evaluated with YAGLR.

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, Comm. of ACM, 13, 1-2, 95-102 (1970).

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

[4] Kay, M. :
Algorithm Schemata and Data Structures in Syntactic Processing, Readings in Natural Language Processing, Morgan Kaufmann Publishers, Inc. pp.35-70

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

[6] Kunth, D.E. :
On the Translation of Languages Left to Right, Information and control, 8(6), pp.607-639 (1965).

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

[8] Schabes, Y. :
Polynomial Time and Space Shift-Reduce Parsing of Arbitrary Context-free Grammars, Proc. of 29th ACL, 106-115 (1991).

[9] Suresh, K.G. and Tanaka, H. :
Implementation and Evaluation of Yet Another Generalized LR Parsing Algorithm, Proc.of the Indian Computing Congress, Tata McGraw-Hill, 506-515 (1991).

[10] Tanaka, H and Suresh, K.G. :
YAGLR : Yet Another Generalized LR Parser, Proceedings of ROCLING IV, Republic of China, 21-31 (1991).

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

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