Handling of Ill-designed Grammars in Tomita's Parsing Algorithm

R. Nozohoor-Farshi

School of Computer Science
University of Windsor, Windsor, Canada N9B 3P4

ABSTRACT

In this paper, we show that some non-cyclic context-free grammars with ε-rules cannot be handled by Tomita's algorithm properly. We describe a modified version of the algorithm which remedies the problem.

1. Introduction

Tomita's parsing algorithm [8,9] is an efficient all-paths parsing method which is driven by an LR parse table with multi-valued entries. The parser employs an acyclic parse graph instead of the conventional LR parser stack. The parser starts as an ordinary LR parser, but splits up when multiple actions are encountered. Multiple parses are synchronized on their shift actions and are joined whenever they are found to be in the same state.

The parallel parsing of all possible paths makes this algorithm suitable for parsing nearly all the arbitrary context-free grammars. In fact, one may view this method as a precompiled form of Earley's algorithm [2,3]. Earley [2] proposed a form of precompiled approach to his method in the case of a restricted class of grammars which has undecidable membership. Tomita's algorithm, on the other hand, is intended for use with general grammars. Since the method uses a parse table, it achieves considerable efficiency over the Earley's non-compiled method which has to compute a set of LR items at each stage of parsing. In this respect, Tomita's algorithm can indeed be considered as a breakthrough in efficient parallel parsing in practical systems. However, there seem to be at least two types of context-free grammars that cannot be handled by this method properly. The first type are cyclic grammars. These grammars have infinite ambiguity and therefore have to be excluded from syntactic analyses. The second kind of grammars include certain context-free grammars with ε-productions. Some of these are unambiguous and some have bounded, bounded direct or unbounded degrees of ambiguity.

Grammars of the latter type may seldom be used to describe the syntax of natural language. In fact, we consider them as somewhat ill-designed. But, they may creep in easily when one is designing a natural language grammar with ε-rules. Such rules cause unexpected infinite loops in parsing. In this paper, we modify the parsing algorithm so that it can handle the second type grammars.

The modification introduces cyclic subgraphs in the original graph-structured parse stack. These subgraphs correspond to the parsing of null substrings in the input sentence. Thus, the modification incurs no cost to the grammars or the inputs that do not need this feature. We believe that adding such a feature to Tomita's algorithm is very desirable. Because, it enriches the method to be comparable to Earley's algorithm in its coverage, and yet it is in a precompiled form.

In the following sections, we discuss the two types of the grammars that cause problems in the original algorithm, and we present the modified algorithm.

2. The Two Types of Grammars

Cyclic grammars are those in which a non-terminal, like A, can derive itself (i.e., $A \overset{+}{\Longrightarrow} A$). G_1 and G_2 are examples of cyclic grammars.

G_1:	G_2:
$S \rightarrow A$	$S \rightarrow S\ S$
$A \rightarrow S$	$S \rightarrow x$
$A \rightarrow x$	$S \rightarrow \varepsilon$

In G_1, A ===> S ===> A, and in G_2, S ===> S S ===> S. Cyclic grammars produce infinite number of parse trees for a finite length input such as "x" in L(G_1) and L(G_2). They cause problem in every parsing algorithm. Therefore, they have been avoided in describing syntax of languages traditionally.

Both Earley's and Tomita's algorithms will fail to detect the cyclicity of G_1 and G_2. Given an input sentence "x", one can however obtain the minimal parses with respect to either grammar by Earley's algorithm and only with respect to G_1 by Tomita's algorithm. The second algorithm will not terminate when the grammar G_2 is used. Tomita [8] discusses the cyclic grammars and rules out their inclusion in natural language parsing. Such exclusion can be achieved through a simple test before generating a parse table (see [1] for example).

Among the second kind grammars that cannot be handled with the original algorithm are the examples G_3, G_4, G_5 and G_6 below.

G_3:	G_5:
$S \rightarrow A\ S\ b$	$S \rightarrow A\ S\ b$
$S \rightarrow x$	$S \rightarrow x$
$A \rightarrow \varepsilon$	$A \rightarrow t$
	$A \rightarrow \varepsilon$

G_4:	G_6:
$S \rightarrow M$	$S \rightarrow M\ N$
$S \rightarrow N$	$M \rightarrow A\ M\ b$
$M \rightarrow A\ M\ b$	$M \rightarrow x$
$M \rightarrow x$	$N \rightarrow b\ N\ A$
$N \rightarrow A\ N\ b$	$N \rightarrow x$
$N \rightarrow x$	$A \rightarrow \varepsilon$
$A \rightarrow \varepsilon$	

G_3 is unambiguous, G_4 has bounded ambiguity, G_5 has bounded direct ambiguity while G_6 has unbounded ambiguity (see Apendix 1 for the definition of these terms). One may note that in these grammars, unlike cyclic grammars, there are only finite number of parse trees for a given finite length input.

A property common to these grammars is that there exists a non-terminal, say S, such that S $\overset{+}{\Longrightarrow}$ α S β where α $\overset{+}{\Longrightarrow}$ ε but β $\overset{*}{\not\Longrightarrow}$ ε. For example, in G_3 or G_5, S can be rewritten as S ===> A S b ===> S b. Rules like these may be excluded from a grammar by using an appropriate test (see Appendix 2). However, one may keep or include such rules in a grammar for the following reasons.

(1) To capture some rare phenomena, for example, embedded that-sentences [[THAT [[THAT . . . [[THAT S] VP] . . .] VP]] VP] in which a number of terminal 'that's are omitted.

(2) Grammars with ε-productions are more concise and readable than the grammars without ε-rules. In fact, elimination of ε-rules from a grammar may increase the size of the grammar exponentially. Therefore, one may use rules similar to the examples G_3 to G_6 to compact the grammar and the parse table, knowing that their presence should not affect the correct parsing of valid inputs.

(3) More frequently, such rules may appear in a grammar when ε-productions are introduced without an adequate care. It is important to note that replacement of these rules (and their associated symbols) may not always be easy.

Grammars G_3 through G_6 can be parsed by Earley's algorithm with no problem. For example, consider the sentence xbbb $\in$ L(G_3). That algorithm will produce the following states.

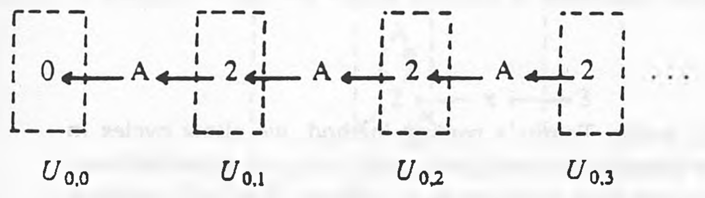
state 0		state 1		state 2		state3	
root → .S#, 0		S → x., 0		S → ASb., 0		S → ASb., 0	
S → .ASb, 0	x →	root → S.#, 0	b →	root → S.#, 0	b →	root → S.#, 0	b →
S → .x, 0		S → AS.b, 0		S → AS.b,0		S → AS.b,0	
A → ε., 0							
S → A.Sb, 0							

state 4		state 5
S → ASb., 0		
root → S.#, 0	# →	root → S#., 0
S → AS.b, 0		

However, the above grammars cause an infinite loop in Tomita's algorithm. Applying the algorithm for ε-grammars (given in [8]) to the input sentence xbbb and the parse table for G_3, the result will be an infinite graph-structured stack as shown below.

State	x	b	#	A	S
0	re3,sh3			2	1
1			acc		
2	re3,sh3			2	4
3		re2	re2		
4		sh5			
5		re1	re1		
	Action table			Goto table	

Grammar G_3:
(1) S → A S b
(2) S → x
(3) A → ε



In Tomita's algorithm the state nodes created in the parse graph are partitioned into U_0, U_1, . . . , U_n where each U_i is the set of state vertices which are created before shifting of word a_{i+1} in the input. Furthermore, in the presence of ε-productions, each U_i is partitioned into $U_{i,0}$, $U_{i,1}$, $U_{i,2}$, ... Each $U_{i,j}$ denotes the set of state vertices created while parsing the j-th null construct after the i-th input symbol a_i is shifted and before the shifting of next actual input symbol a_{i+1} takes place. Tomita assumes that the number of null constituents between every adjacent pair of input symbols is always finite. Though his assumption is correct for non-cyclic grammars, it cannot be incorporated as such in the parser since it will require arbitrary and complex lookaheads in general case. As noted earlier this strategy fails in the example grammars.

It is interesting to note that the same strategy will succeed in the case of LR grammar G_3^r which is the reverse of G_3.

G_3^r:
S → b S A
S → x
A → ε

The difference between G_3 and G_3^r is that in G_3 a null deriving constituent appears on the left part of a recursive phrase, while in G_3^r, it appears on the right side of the recursive construct. Thus, the parser for G_3 does not know how many A's it has to create before consuming the first input word "x". In the case of G_3^r, the left context provides enough information to limit the number of empty constructs to a finite size.

One may observe that though G_3 is an unambiguous grammar, it is not LR(k) for any k. Viewing differently, one may argue that such grammars can be parsed deterministically and more efficiently by non-canonical parsers. Marcus' parser [5] and bottom-up variations of it described in [6,7] can handle this grammar in a much better way, since they create the rightmost A in the parse tree first. The reader may also consult [6,7] to see the advantage of these parsers over Tomita's algorithm when grammars like G_7 are to be parsed.

G_7:
S → a S a
S → B S b
S → C S c
B → a
C → a
S → x

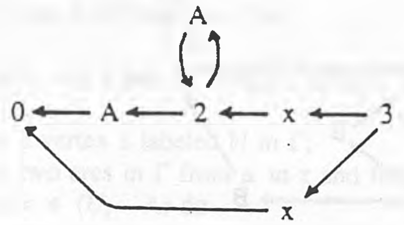
However, we should emphasis that the whole thrust and advantage of Tomita's parser lies in obtaining multiple parses with respect to ambiguous grammars such as those in examples G_4 to G_6.

In the following section, we modify Tomita's algorithm in a way that the second type grammars can be handled within this framework. In doing so, we believe that we are introducing a version of Tomita' algorithm which is a partially-precompiled equivalent of Earley's parser and can be applied to all non-cyclic context-free grammars.

3. Modified Algorithm

To accommodate grammars like G_3 to G_6 within Tomita's parsing method, we allow cycles in the graph-structured parse stack. These cycles are introduced in the parse graph in a very restricted way. Each cyclic subgraph represents a regular expression that corresponds to parsing of a null substring between two adjacent input symbols. Unlike Tomita's algorithm for ε-grammars [8], we do not partition each U_i any further. So, the set of state vertices of each cyclic subgraph entirely lies within a single U_i. Obviously, cycles are created within U_i only if parsing of the input sentence requires them. Since the parse graph is now cyclic, we do reductions along arbitrary paths (i.e., paths that are not simple and may contain repetitive vertices or arcs). Such paths are usually termed *(directed) walks* in graph theory.

Our approach though is intuitive, it has its roots in LR theory. In LR parsing, the finite automaton (from which a parse table is extracted) represents the set of all viable prefixes of the grammar in closed form. The parse stack, on the other hand, represents an actual viable prefix (of a right sentential form) in open form. The actual viable prefix is built from the input symbols which are consumed by the LR parser. It is necessary to hold the actual viable prefix in the stack so that the parser can be provided with the exact left context. However, in the modified all-paths parser we do not need to keep the null-deriving segments of the left context in open form. For example, in parsing sentences like xb. . .b ∈ $L(G_3)$, ε and A. . .A are the viable prefixes when the parser scans the first input symbol "x". Since each A derives a null string and we do not know exactly how many of them we should assume, we represent the left context in the closed form $\varepsilon + AA^*$. The corresponding parse graph will appear as the figure in below when "x" is just shifted. The parser will pick as many A's as it needs from this regular expression when the remainder of the sentence is seen.

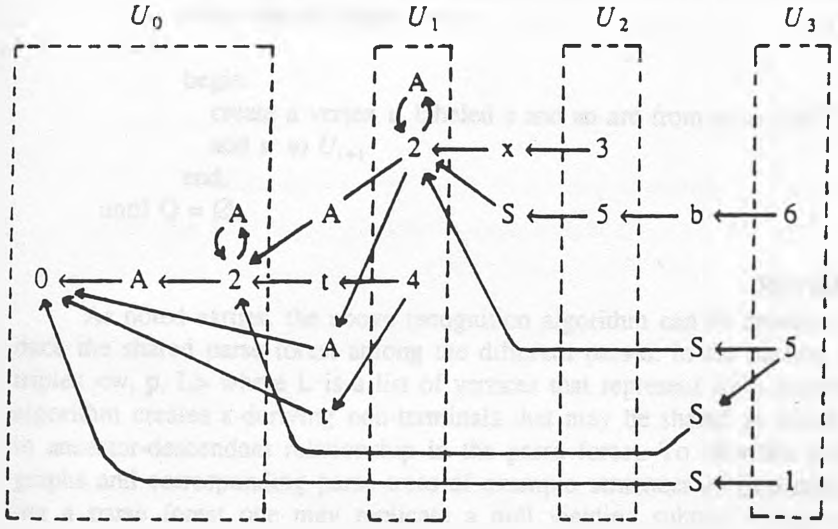


Similarly, consider the example grammar G_5 and the parse table for it as shown below. One will obtain the following snapshot of the parse graph after the parser consumes the prefix txb of the sentence txb. . .b, and all the appropriate reductions are done.

state	t	x	b	#	A	S
0	sh4,re4	sh3,re4			2	1
1				acc		
2	sh4,re4	sh3,re4			2	5
3			re2	re2		
4	re3	re3				
5			sh6			
6			re1	re1		

Action table Goto table

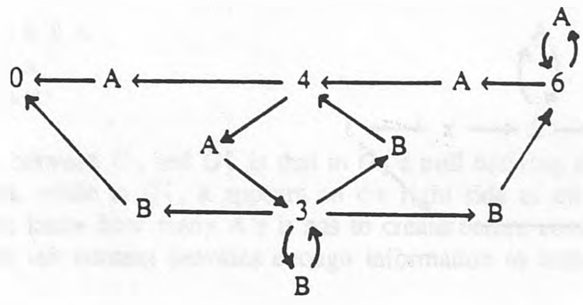
Grammar G_5
(1) $S \rightarrow A\,S\,b$
(2) $S \rightarrow x$
(3) $A \rightarrow t$
(4) $A \rightarrow \varepsilon$



In this example, the left context just before shifting the word "x" can represented as the regular expression $(AA^* \mathbf{A} + A)\ A^*$. For clarity, the bold faced **A** represents the non-terminal obtained by reducing "t". For the same reason, we are not combining identical symbol vertices which are adjacent to the same state vertex, (a measure of optimization suggested in [8]), in the illustrated examples or in the algorithm that to follow.

As another example, an interested reader using the parse table in Appendix 3 may verify that U_0 for the grammar G_8 will have the following format.

G_8:
$S \rightarrow x$
$S \rightarrow B\ S\ b$
$S \rightarrow A\ S\ b$
$B \rightarrow A\ A$
$A \rightarrow \varepsilon$



In the above examples, we have used an LALR(1) parser generator, similar to YACC [4], to obtain the parse tables with multi-valued entries. Tomita [8,9] also uses LALR(1) tables, however, using non-optimized LR(1) tables will decrease the number of superfluous reductions in general.

We are now in a position to present the modified algorithm. For simplicity, we give an algorithm for a recognizer rather than a parser. The recognizer can be augmented in a way similar to that of [8] to provide a parser that also creates the parse forest.

Recognition Algorithm:

```
PARSE (G, $a_1 \cdots a_n$)
  • $\Gamma := \varnothing$.
  • $a_{n+1}$ := '#'.
  • r := FALSE.
  • Create a vertex $v_0$ labeled $s_0$ in $\Gamma$.
  • $U_0 := \{v_0\}$.
  • For i := 1 to n do PARSEWORD (i).
  • Return r.

PARSEWORD (i)
  • A := $U_i$.
  • R := $\varnothing$; Q := $\varnothing$.
  • Repeat
       if A $\neq \varnothing$ then do ACTOR
       else if R $\neq \varnothing$ then do COMPLETER
    until R = $\varnothing$ and A = $\varnothing$.
  • Do SHIFTER.

ACTOR
  • Remove an element $v$ from A.
  • For all $\alpha \in$ ACTION (STATE ($v$), $a_{i+1}$) do
      begin
       if $\alpha$ = 'accept' then r := TRUE;
```

```
        if $\alpha$ = 'shift s' then add <$v$ ,s> to Q;
        if $\alpha$ = 'reduce p' then
            for all vertices $w$ such that there exists a directed
            walk of length 2 | RHS (p) | from $v$ to $w$  /* For $\varepsilon$-rules this is a trivial walk, i.e. $w$=$v$ */
            do add <$w$ ,p> to R
      end.

COMPLETER
  • Remove an element <$w$ ,p> from R.
  • N := LHS (p);  s := GOTO (STATE ($w$), N)..
  • If there exists $u \in U_i$ such that STATE($u$) = s then
          begin
              if there does not exist a path of length 2 from $u$ to $w$ then
                  begin
                      create a vertex z labeled N in $\Gamma$;
                      create two arcs in $\Gamma$ from $u$ to z and from z to $w$;
                      for all $v \in (U_i$ - A) do
                      /* In the case of non-$\varepsilon$-grammars this loop executes for $v$=$u$ only */
                          for all q such that 'reduce q' $\in$ ACTION (STATE ($v$), $a_{i+1}$) do
                              for all vertices $t$ such that there exists a directed walk of
                              length 2 | RHS (q) | from $v$ to $t$ that goes through vertex z
                              do add <$t$ ,q> to R
                  end
          end
     else /* i.e., when there does not exist $u \in U_i$ such that STATE ($u$) = s */
          begin
              create in $\Gamma$ two vertices $u$ and z labeled s and N respectively;
              create two arcs in $\Gamma$ from $u$ to z and from z to $w$;
              add $u$ to both A and $U_i$
          end.

SHIFTER
  • $U_{i+1}$ := $\varnothing$.
  • Repeat
        remove an element <$v$ ,s> from Q;
        create a vertex x labeled $a_{i+1}$ in $\Gamma$;
        create an arc from x to $v$;
        if there exists a vertex $u \in U_{i+1}$ such that STATE ($u$) = s then
            create an arc from $u$ to x
        else
            begin
                create a vertex $u$ labeled s and an arc from $u$ to x in $\Gamma$;
                add $u$ to $U_{i+1}$
            end.
    until Q = $\varnothing$.
```

As noted earlier, the above recognition algorithm can be changed into a parsing algorithm to produce the shared parse forest among the different parses. In the parsing algorithm the elements of R are triples <w, p, L> where L is a list of vertices that represent RHS symbols of p. One must note that our algorithm creates ε-deriving non-terminals that may be shared as a son by other non-terminals that are in ancestor-descendant relationship in the parse forest. To illustrate this point, we show the full parse graphs and corresponding parse trees of example sentences in Appendix 4. As an alternative, in building a parse forest one may replicate a null yielding subtree whenever this subtree participates in a reduction where at least one other sibling has non-empty yield.

As a final remark, we may add that the above algorithm can obtain the minimal parses in the case of cyclic grammars, but does not detect their cyclicity. It is also possible to precompile some subsets of each U_i that are obtained under the transitions with respect to null-deriving non-terminals.

4. Conclusion

We have modified Tomita's parsing algorithm so that it can handle some ill-designed grammars with ε-rules that caused a problem in the original algorithm. We have introduced cycles in the parse graph in a restricted way. This makes the parse graph in the new algorithm a cyclic directed graph in some general cases. However, the new algorithm works exacltly like the original one in case of grammars that have no ε-productions. This algorithm has no extra costs beyond that of the original algorithm.

We believe that the modified algorithm is a precompiled equivalent of Earley's algorithm with respect to its coverage, though we have not provided a formal proof for it. The resulting algorithm suggests that Tomita's graph-structured parsing approach can be used with a broader class of context-free grammars.

Appendix 1: Ambiguous grammars

Definition: A context-free grammar G has bounded ambiguity of degree k if each sentence in L(G) has at most k distinct derivation trees.

Definition: A context-free grammar G has unbounded ambiguity if for each $i \geq 1$, there exists a sentence in L(G) which has at least i distinct derivation trees.

Definition: The degree of direct ambiguity of a non-terminal A with respect to a string x is the number of distinct tuples $(p, x_1, x_2, \ldots, x_n)$, where p is a production $A \rightarrow B_1 B_2 \cdots B_n$, and $x_1 x_2 \cdots x_n = x$ is a factorization of x such that $B_i \overset{*}{\Longrightarrow} x_i$ for $1 \leq i \leq n$.

Definition: A context-free grammar has bounded direct ambiguity of degree k if the degree of direct ambiguity of any of its non-terminals with respect to any string is at most k.

For example, the grammar G_5 has direct ambiguity of degree 2, in spite of being unboundedly ambiguous.

Appendix 2: Identifying the ε-grammars that cannot be parsed by the original algorithm.

Let G = (N , T , P , S) be a context-free grammar with ε productions. The following algorithm decides whether G can be parsed by the original algorithm.

(1) Compute the set of non-terminals $E = \{ C \mid C \overset{+}{\Longrightarrow} \varepsilon \}$ that can derive a null string.

(2) Let $\rho \subset N \times N$ be a binary relation such that $(A, B) \in \rho$ if and only if $A \rightarrow C_1 C_2 \cdots C_n B \alpha$ is a production in P and $C_i \in E$ for $1 \leq i \leq n$.

(3) Compute ρ^+ the closure of ρ. If there exists a non-terminal A where $(A, A) \in \rho^+$ then G cannot be parsed by the Tomita's original algorithm for ε-grammars.

Appendix 3: Parse Table for Grammar G_3

state	x	b	#	A	B	S
0	sh2,re5			4	3	1
1			acc			
2		re1	re1			
3	sh2,re5			4	3	5
4	sh2,re5			6	3	7
5		sh8				
6	sh2,re4,re5			6	3	7
7		sh9				
8		re2	re2			
9		re3	re3			

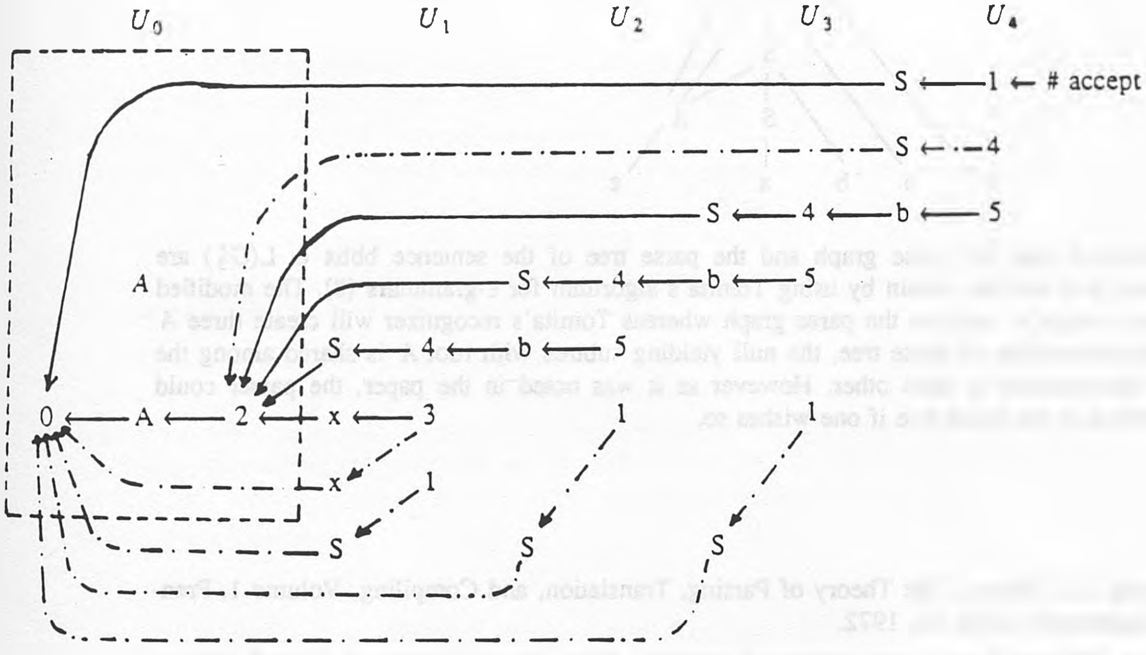
Action table Goto table

Grammar G_3

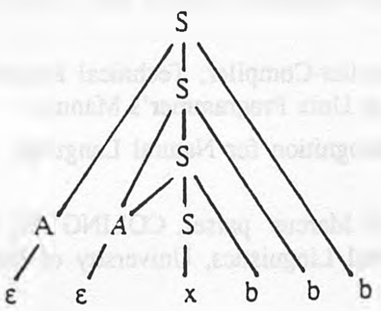
(1) $S \rightarrow x$
(2) $S \rightarrow B\ S\ b$
(3) $S \rightarrow A\ S\ b$
(4) $B \rightarrow A\ A$
(5) $A \rightarrow \varepsilon$

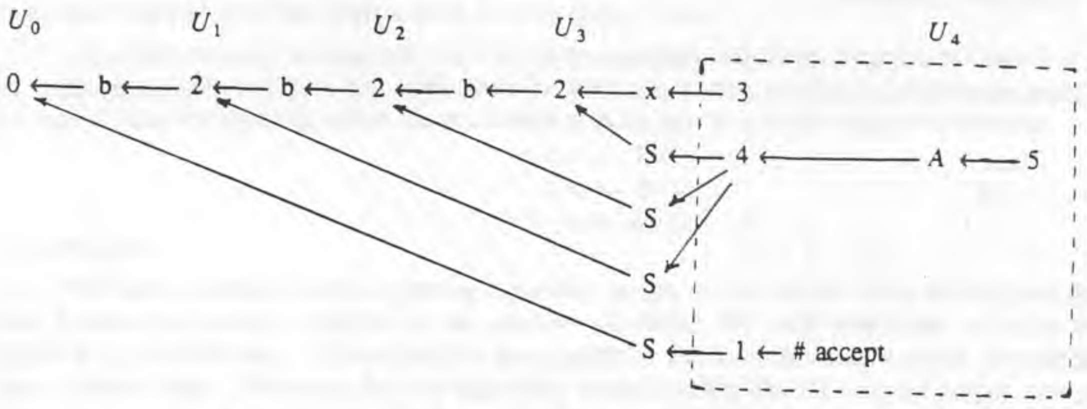
Appendix 4: Parsing of example sentences

The following figures illustrate parsing of the sentences xbbb $\in L(G_3)$ and bbbx $\in L(G_3')$. The dotted lines indicate the rejected paths. The shared non-terminals are shown in italics.

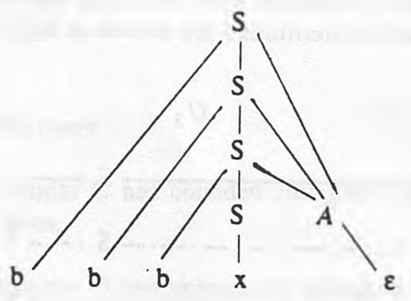


Parse graph and parse tree of the sentence xbbb $\in L(G_3)$





Parse graph and parse tree of the sentence bbbx $\in$ L(G_3')



One may observe that the parse graph and the parse tree of the sentence bbbx $\in$ L(G_3') are different from those that one can obtain by using Tomita's algorithm for ε-grammars [8]. The modified recognizer creates a single *A* node in the parse graph whereas Tomita's recognizer will create three *A* vertices. In our representation of parse tree, the null yielding subtree with root *A* is shared among the S nodes that are descendants of each other. However as it was noted in the paper, the parser could replicate such subtrees in the parse tree if one wishes so.

References

[1] A.V. Aho and J.D. Ullman, The Theory of Parsing, Translation, and Compiling, Volume 1, Prentice Hall, Englewood Cliffs, NJ, 1972.

[2] J. Earley, An Efficient Context-free Parsing Algorithm, Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, Pittsburg, PA, 1968.

[3] J. Earley, An efficient context-free parsing algorithm, CACM, vol. 13, no. 2, pp. 94-102, February 1970.

[4] S.C. Johnson, YACC: Yet Another Compiler-Compiler, Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975. Also reproduced in Unix Programmer's Manual.

[5] M.P. Marcus, A Theory of Syntactic Recognition for Natural Language, MIT Press, Cambridge, MA, 1980.

[6] R. Nozohoor-Farshi, On formalizations of Marcus' parser, COLING' 86, Proceedings of the 11th International Conference on Computational Linguistics, University of Bonn, West Germany, pp. 533-535, August 1986.

[7] R. Nozohoor-Farshi, LRRL(k) Grammars: A Left to Right Parsing Technique with Reduced Lookaheads, Ph.D. Thesis, Department of Computing Science, University of Alberta, Edmonton, Canada, 1986.

[8] M. Tomita, Efficient Parsing for Natural Language, Kluwer Academic Publishers, Boston, MA, 1986.

[9] M. Tomita, An efficient augmented-context-free parsing algorithm, Computational Linguistics, vol. 13, no. 1-2, pp. 31-46, January 1987.

Acknowledgement

The research reported in this paper was supported by the Natural Sciences and Engineering Research Council of Canada grant A9447.