

A Hybrid GLR Algorithm for Parsing with Epsilon Grammars

Marwan Shaban
shaban@cs.bu.edu

March 22, 1994

BU-CS Tech Report # 94-004

Computer Science Department
Boston University
111 Cummington Street
Boston, MA 02215

Abstract

We give a hybrid algorithm for parsing ϵ -grammars based on Tomita's non- ϵ -grammar parsing algorithm ([Tom86]) and Nozohoor-Farshi's ϵ -grammar recognition algorithm ([NF91]). The hybrid parser handles the same set of grammars handled by Nozohoor-Farshi's recognizer. The algorithm's details and an example of its use are given. We also discuss the deployment of the hybrid algorithm within a GB parser, and the reason an ϵ -grammar parser is needed in our GB parser.

Contents

1	Introduction	3
2	Tomita's Method for Parsing with ϵ-grammars	3
3	Fong's Method for Parsing with ϵ-grammars	4
4	Nozohoor-Farshi's Method for Recognition with ϵ-grammars	5
5	A Family of Algorithms	5
6	The Hybrid Algorithm	8
7	A Crucial Modification	13
8	Handling Cyclic Grammars	21

1 Introduction

Many GB parsers utilize a core context-free parser to recover the x-bar phrase structure of an input sentence. A CF s-structure grammar used in this phrase structure recovery stage will contain several erasing rules. So, the CF parser used for the phrase structure recovery stage must be able to handle so-called ϵ -grammars, *i.e.*, grammars containing epsilon productions. While it is possible to convert an epsilon-grammar into one without epsilon-productions, it is unacceptable to do this both on grounds of faithfulness and clarity.

In our GB parser ([Sha93]), we use Tomita's algorithm (described in [Tom86]), a CF parsing algorithm (also known as Generalized LR parsing, or GLR parsing). The original incarnation of this parsing method couldn't handle ϵ -grammars. To incorporate that ability, two algorithms related to the original GLR parsing algorithm have been proposed. A *parsing* algorithm for ϵ -grammars was proposed by Tomita (in [Tom86]), and a *recognition* algorithm for ϵ -grammars was proposed by Nozohoor-Farshi (in [NF91]). Nozohoor-Farshi's recognizer, however, handles a larger class of ϵ -grammars than Tomita's parsing method does. We describe here a GLR parsing algorithm based on Nozohoor-Farshi's recognition algorithm for ϵ -grammars and Tomita's parsing algorithm for non- ϵ -grammars. Our algorithm performs parsing (whereas Nozohoor-Farshi's only performed recognition), and handles the same set of ϵ -grammars as Nozohoor-Farshi's recognizer does.

2 Tomita's Method for Parsing with ϵ -grammars

It is known that Tomita's ϵ -grammar parser [Tom86] does not work for all ϵ -grammars. As an example of a grammar that cannot be used in conjunction with Tomita's parsing method, Nozohoor-Farshi [NF91] gives Grammar 1 shown below.

G_1 :

- (1) $S \rightarrow A S b$
- (2) $S \rightarrow x$
- (3) $A \rightarrow \epsilon$

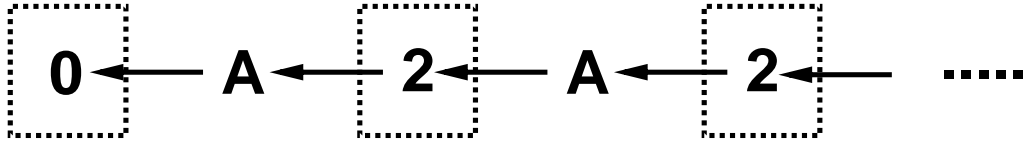


Figure 1: Graph-structured stack in Tomita’s algorithm with G_1 and input string $xbbb$.

Tomita’s ϵ -grammar parser goes into an infinite loop when confronted with the above grammar and the input sentence $xbbb \in L(G_1)$. The graph-structured stack it creates is shown in Figure 1 (Figure 5.2 in [NF91]). The reason Tomita’s algorithm cannot handle this sentence is that it doesn’t know how many empty A nonterminals to hypothesize before shifting the terminal x .

In the context of our GB parsing, it is easy to imagine why Tomita’s algorithm would have the same problem. Our s-structure grammar can contain constructs similar to the one in G_1 . For example, take a head-final language (where, *e.g.*, the complement of a verb occurs before the verb), and consider the possibility of CPs with null subjects (or with subjects that occur after the predicate) and verbs that take sentential complements (of category CP). In this situation, it is possible for a CP to be embedded in a matrix CP with no overt material between the start of the matrix CP and the embedded CP, leading Tomita’s method into an infinite loop as in the case of G_1 . We conclude that Tomita’s ϵ -grammar parsing algorithm will not work for us.

3 Fong’s Method for Parsing with ϵ -grammars

In implementing his GB parser, Fong [Fon91] implemented a variation of GLR parsing using a recursive control flow mechanism. His recursive control flow precludes using Nozohoor-Farshi’s ϵ -grammar scheme, yet Fong manages to parse with an s-structure covering grammar which contains ϵ -productions.

Fong’s solution, described on pages 142-145 of [Fon91], depends on an off-line analysis of the S-structure grammar to deduce which nonterminals may cause a problem, and how. Fong then uses a new “structure” stack which

holds housekeeping information used by specially-coded hooks in the LR parser. The code figures out when epsilon-productions may be “licensed,” and when they may not (*i.e.*, when they will cause a problem). Empty categories are licensed only when an antecedent has already been seen in the input, or when an antecedent is detected further ahead in the input stream. By doing this, the prediction of infinitely many empty categories is blocked. Also, infinite nesting of CPs is explicitly blocked by hooks within the LR parser specifically coded for that purpose.

We implemented a version of the LR parser that uses recursive control flow, and augmented it with a mechanism similar to Fong’s which allows ϵ -grammars to be used. This method, however, is not well suited to handle the problem, for reasons given in the next section.

4 Nozohoor-Farshi’s Method for Recognition with ϵ -grammars

The problem with Fong’s scheme is that it depends on an off-line analysis of the S-structure grammar, which must be carried out by a human. If the s-structure grammar changes, the off-line analysis must be redone to make sure that the LR parser will still handle the ϵ -production-containing grammar correctly. A better solution would be to extend Tomita’s algorithm to allow it to handle ϵ -grammars. Nozohoor-Farshi’s recognition method [NF91] gives just such an extension. However, this method, as Nozohoor-Farshi gives it, applies only to the normal (Tomita-style) control flow which uses a graph-structured stack. It cannot readily be used to fix the Fong-style GLR parser that uses recursive control flow.

5 A Family of Algorithms

Figure 2 shows a family of GLR recognition and parsing algorithms. To implement the hybrid parsing algorithm, it was necessary to take Tomita’s parsing algorithm, remove the modifications (denoted on figure 2 as *B*) used by Tomita to handle ϵ -grammars, and add Nozohoor-Farshi’s ϵ -grammar modifications (denoted on figure 2 as *A*). The process was not a trivial one, since the changes in Tomita’s optimization step (denoted on figure 2 as *C*) changed

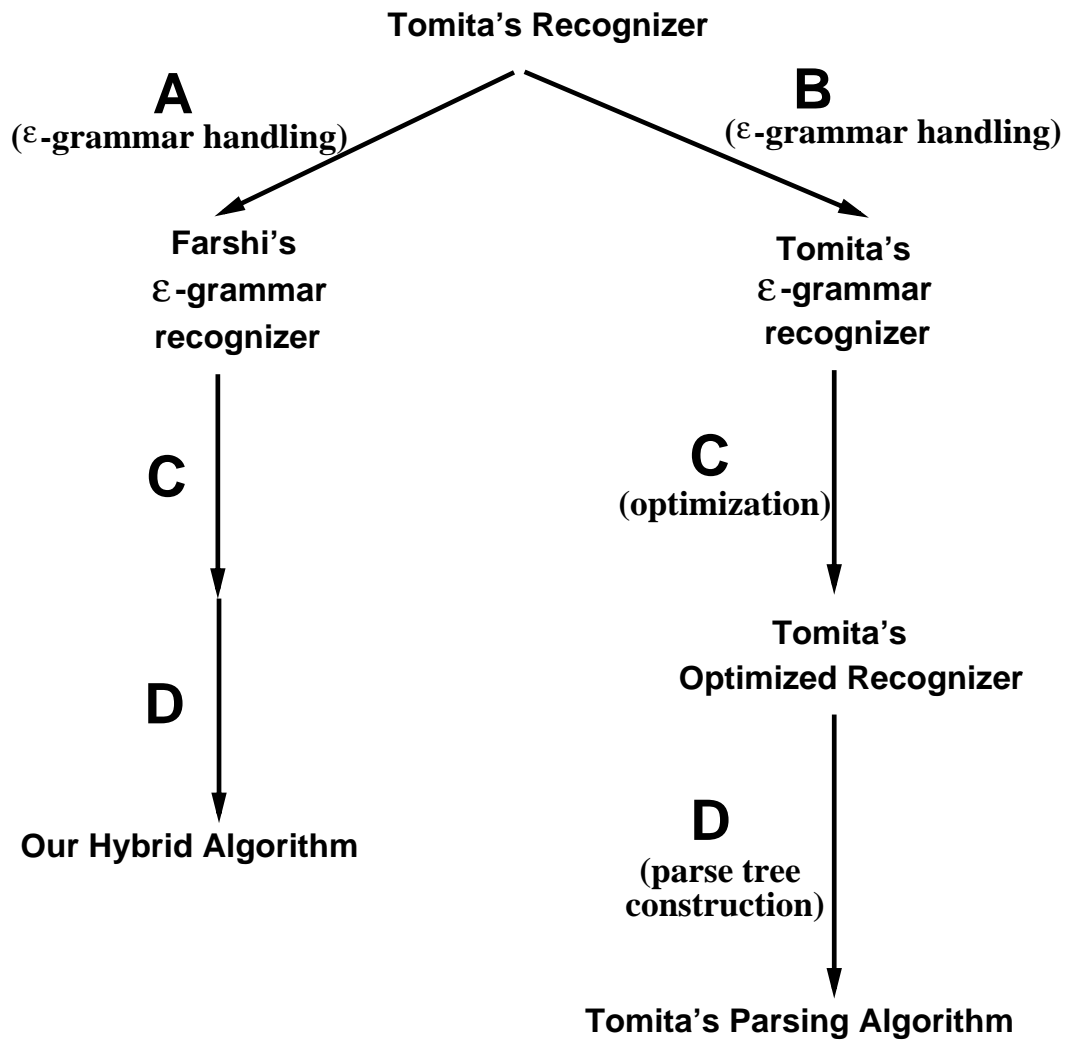


Figure 2: Family of GLR algorithms. Labels on the arcs denote modifications made to a source algorithm to derive the destination algorithm.

many details of the algorithm, which had to be reconciled with Nozohoor-Farshi's modifications.

6 The Hybrid Algorithm

The following is the hybrid GLR parsing algorithm that was derived as explained in the above section. Figure 3 shows the global variables used by the algorithm and Figure 4 shows the pre-defined functions used by the algorithm.

PARSE($G, a_1 \cdots a_n$)

- $\Gamma \Leftarrow \phi$
- $T \Leftarrow \phi$
- $r \Leftarrow \text{NIL}$
- $a_{n+1} \Leftarrow \text{'\$'}$
- create in Γ a vertex v_0 labeled s_0 .
- $U_0 \Leftarrow \{v_0\}$
- for $i \Leftarrow 0$ to n do PARSEWORD(i).
- return r , as the root of the parse forest.

PARSEWORD(i)

- $A \Leftarrow U_i$.
- $R, Q \Leftarrow \phi$
- repeat
 - if $A \neq \phi$ then do ACTOR
 - else if $R \neq \phi$ then do REDUCER
- until $A = \phi \wedge R = \phi$.
- do SHIFTER.

<p>G: Grammar, <i>i.e.</i>, a set of productions.</p> <p>$a_1 \dots a_n$: Input string of length n.</p> <p>Γ: The graph-structured stack.</p> <p>r: Contains the result. If r is the root of a parse forest, then the sentence is accepted, else the sentence is rejected. It is altered in ACTOR, and initialized in PARSE.</p> <p>U_i: A set of vertices in Γ created when parsing a_i. Let a_i be the word most recently shifted. Then, U_i is a set of top vertices. It is altered in PARSE, REDUCER and SHIFTER.</p> <p>A: A set of active vertices in U_i to be processed. ACTOR will take care of them. It is also altered in REDUCER, and initialized in PARSEWORD.</p> <p>R: A set of top edges to be reduced. Each element is a 3-tuple $\langle v, x, p \rangle$, where $v \in U_i$, $x \in \Gamma$ and p is a production rule. The existence of $\langle v, x, p \rangle$ in R means that ‘reduce p’ will be applied on all paths starting with the vertex v and passing through the vertex x. REDUCER will take care of them. It is also altered in ACTOR, and initialized in PARSEWORD.</p> <p>Q: A set of vertices to be shifted. Each element is a 2-tuple $\langle v, s \rangle$, where $v \in U_i$ and s is a state number. The existence of $\langle v, s \rangle$ in Q means that ‘shift s’ is to be applied on v. SHIFTER will take care of them. It is also altered in ACTOR, and initialized in PARSEWORD.</p> <p>T: The parse forest.</p>

Figure 3: Global variables used by the algorithm.

<p>LEFT(p): The left hand side symbol of production p.</p> <p>p: The length of the right hand side of production p.</p> <p>STATE(v): Takes a vertex in Γ as its argument, and returns the state number that the vertex is labeled with.</p> <p>SYMBOL(x): Takes a vertex in Γ as its argument, and returns the symbol which the vertex is labeled with.</p> <p>SUCCESSORS(v): Takes a vertex in Γ as its argument, and returns a set of all vertices in Γ such that there is an edge from v to each of these vertices.</p> <p>GOTO(s, N): Look up the goto table. s is a state number and N is a non-terminal. Returns a state number.</p> <p>ACTION(s, a): Look up the action table. s is a state number and a is a terminal symbol. Returns a list of actions, each of which is of one of these forms: ‘accept’, ‘shift s’, ‘reduce p’ or ‘error’.</p> <p>SUBNODES(v): Takes a vertex v in T as its argument, and returns a set of successor lists, $\{L_1, L_2, \dots\}$ such that $\langle v, L_i \rangle \in E$ for all i.</p> <p>ADDSUBNODE(v, L): Adds a successor list $\langle v, L \rangle$ to E in $T = (V, E)$.</p>

Figure 4: Pre-defined functions used by the algorithm.

ACTOR

- remove one element v from A .
- for all $\alpha \in \text{ACTION}(\text{STATE}(v), a_{i+1})$ do
 - if $\alpha = \text{'accept'}$ then
 - * Let s be the first (and only) element of $\text{SUCCESSORS}(v)$.
 - * $r \leftarrow \text{SYMBOL}(s)$.
 - if $\alpha = \text{'shift } s\text{'}$ then add $\langle v, s \rangle$ to Q .
 - if $\alpha = \text{'reduce } p\text{'}$ then
 - * add $\langle v, v, p \rangle$ to R .

SHIFTER

- $U_{i+1} \leftarrow \phi$.
- create in T a node n labeled a_{i+1} .
- for all s such that $\exists v (\langle v, s \rangle \in Q)$,
 - create in Γ vertices w and x labeled s and n , respectively.
 - create in Γ an edge from w to x .
 - add w to U_{i+1} .
 - for all v such that $\langle v, s \rangle \in Q$ do
 - * create an edge in Γ from x to v .

REDUCER

- remove one element $\langle v, x, p \rangle$ from R .
- $N \Leftarrow \text{LEFT}(p)$.
- $Y \Leftarrow \{y \mid \text{there exists a directed walk of length } 2|p| \text{ from } v \text{ to } y \text{ (for } \epsilon\text{-rules this is a trivial walk, i.e., } v = y) \text{ that goes through vertex } x\}$.
- for all j such that $\exists y(y \in Y \wedge y \in U_j)$ do
 - $Y_j \Leftarrow \{y \mid y \in Y \wedge y \in U_j\}$
 - for all s such that $\exists w(w \in Y_j \wedge \text{GOTO}(\text{STATE}(w), N) = s)$ do
 - * $W \Leftarrow \{w \mid w \in Y_j \wedge \text{GOTO}(\text{STATE}(w), N) = s\}$
 - * $lhs\text{-lists} \Leftarrow \{L \mid L \text{ is the list } (\text{SYMBOL}(z_{|p|}), \text{SYMBOL}(z_{|p|-1}), \dots, \text{SYMBOL}(z_2), \text{SYMBOL}(z_1)) \text{ where } z_n, \text{ for } n = 1 \dots |p|, \text{ are symbol vertices in a path from } v \text{ to } w \text{ for some } w \in W\}$. (Note that $lhs\text{-lists}$ is a set, so it should not contain duplicate lists. Note also that if two paths lead from v to w , both must be added to $lhs\text{-lists}$.)
 - * if there exists u such that $u \in U_j \wedge \text{STATE}(u) = s$ then
 - if there already exists an edge from u to a vertex z such that $\text{SUCCESSORS}(z) = W$ then
 - for all $L \in lhs\text{-lists}$ do
 - $\text{ADDSUBNODE}(\text{SYMBOL}(z), L)$.
 - else
 - create a node n in T labeled N .
 - for all $L \in lhs\text{-lists}$ do
 - $\text{ADDSUBNODE}(n, L)$.
 - create in Γ a vertex z labeled n .
 - create an edge in Γ from u to z .
 - for all $w \in W$ do

- create an edge in Γ from z to w .
- for all $v \in (U_i - A)$ (in the case of non- ϵ -grammars this loop executes for $v = u$ only) do
 - for all q such that ‘reduce q ’ \in ACTION(STATE(v), a_{i+1}) do
 - add $\langle v, z, q \rangle$ to R
- * else
 - create in T a node n labeled N .
 - for all $L \in$ lhs-lists do
 - ADDSUBNODE(n, L).
 - create in Γ two vertices u and z labeled s and n .
 - create in Γ an edge from u to z .
 - for all $w \in W$ do
 - create in Γ an edge from z to w .
 - add u to both A and U_i .

7 A Crucial Modification

The introduction of Nozohoor-Farshi’s ϵ -production handling code to Tomita’s original parse forest construction scheme causes it to generate incorrect parse forests in some situations. For example, consider Grammar 2 (shown below). Grammar 2’s LR parse tables are shown in Figure 5. The algorithm’s (incorrect) graph-structured stack when applied to Grammar 2 and the input string “noun verb” is shown in Figure 6. Figure 7 shows the incorrect parse forest generated by this algorithm. The indices given to the terminals and nonterminals in Figures 6 and 7 illustrate which stack nodes point to which forest nodes.

G_2 :
 (1) $CP \rightarrow NP IP$

State	<i>action</i>			<i>goto</i>		
	noun	verb	\$	NP	IP	CP
0	sh3, re3	re3		1		2
1	sh6	re3		4	5	
2			acc			
3	re4	re4				
4		sh7				
5			re1			
6		re4				
7			re2			

Figure 5: LR Parse tables for Grammar 2.

- (2) $IP \rightarrow NP \text{ verb}$
- (3) $NP \rightarrow \epsilon$
- (4) $NP \rightarrow \text{noun}$

The parse forest shown in Figure 7 is incorrect. Its lisp representation (as output by the parser) is as follows:¹

```
(CP
  ((NP NIL) . #1=((IP ((NP NIL) . #2=((VERB)))
    ((NP (#3=(NOUN))) . #2#))))
  ((NP (#3#)) . #1#))
```

After unpacking, this forest corresponds to the following four parse trees; the first and last are incorrect because they do not yield the terminal string “noun verb:”

```
((CP (NP) (IP (NP) (VERB)))
 (CP (NP) (IP (NP (NOUN)) (VERB)))
 (CP (NP (NOUN)) (IP (NP) (VERB)))
```

¹The *sharp* symbols are Lisp’s way of indicating the structure sharing within the forest. A $\#x=$ symbol indicates the first occurrence of a shared structure, and a $\#x\#$ symbol indicates a further occurrence of a shared structure. So, in the above example, the $\#3\#$ indicates a pointer to the list $\#3=(NOUN)$.

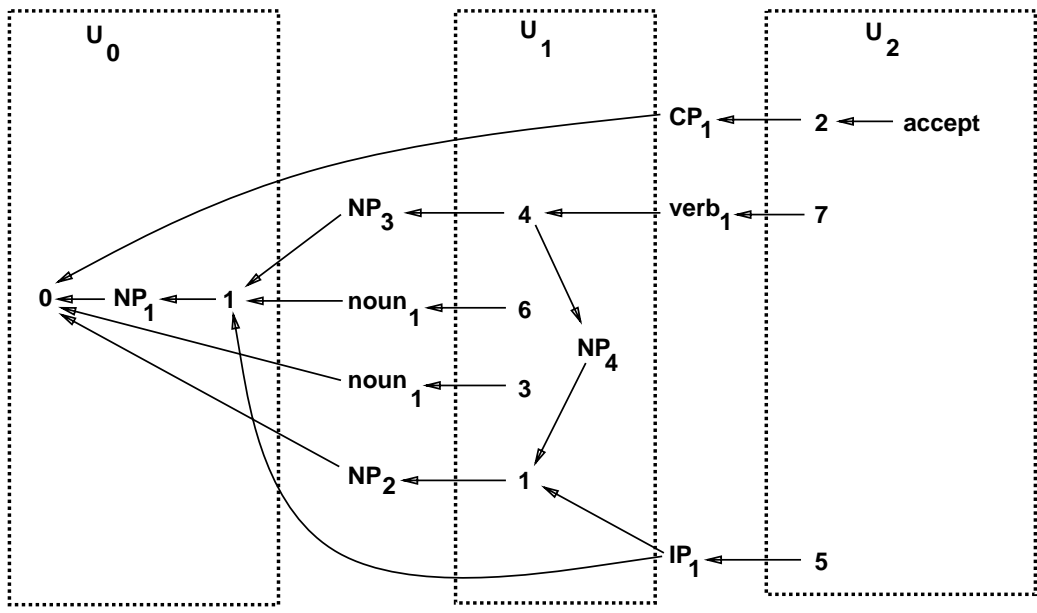


Figure 6: Graph-structured stack for Grammar 2 and the input sentence “noun verb” prior to our fix.

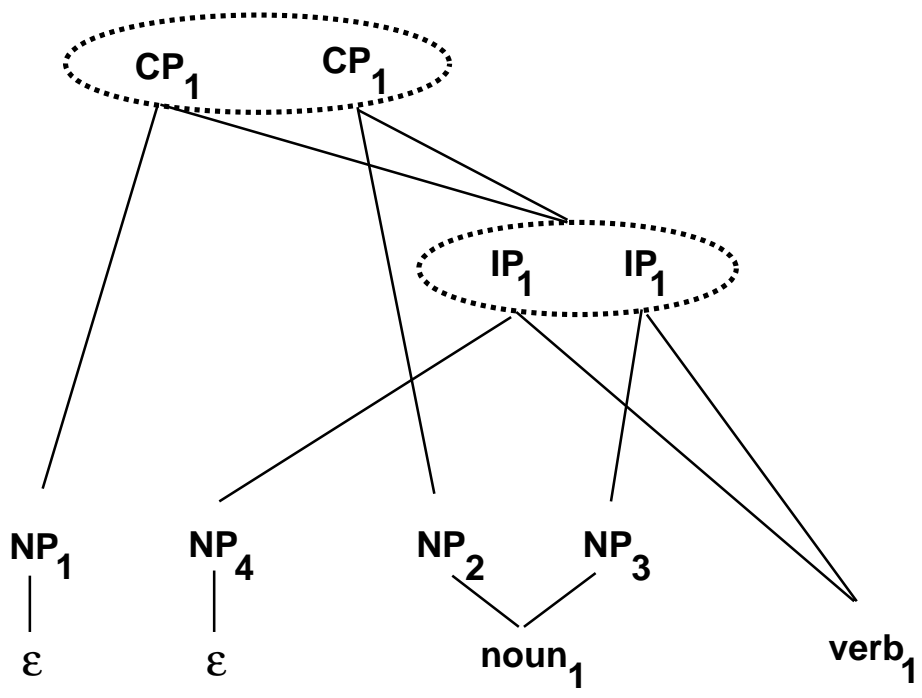


Figure 7: Parse forest for Grammar 2 and the input sentence “noun verb” prior to our fix.

(CP (NP (NOUN)) (IP (NP (NOUN)) (VERB))))

The reason for this problem is that two arcs start from IP_1 in Figure 6, one pointing to state 1 in U_1 and the other pointing to state 1 in U_0 . This corresponds to the local ambiguity leading to the packed IP node in Figure 7. This IP node should not be packed. It should be split into two separate nodes, since sub-nodes of a packed node must (by definition) span the same set of terminals in the input string, which is not the case in Figure 6. The use of ϵ -productions in the grammar has caused the algorithm to predict this incorrect local ambiguity.

Our algorithm (given in section 6) fixes this problem in the “reducer” subroutine by separating the set of vertices Y (which corresponds to the destinations of directed walks from the current node v of length $2|p|$) into the sets Y_j , each of which contain vertices located in the same U_j set. Each of these Y_j sets is then processed separately. In the case of our example, this results in two separate IP nodes (IP_1 and IP_2) in the graph-structured stack and the parse forest (see Figures 8 and 9, respectively). The resulting parse forest is the following:

```
(CP
  ((NP (#1=(NOUN)))
   (IP ((NP NIL) . #2=((VERB))))))
((NP NIL)
 (IP ((NP (#1#)) . #2#))))
```

When the above parse forest is unpacked, we obtain the following two correct parse trees:

```
((CP (NP #1=(NOUN))
      (IP (NP) #2=(VERB)))
 (CP (NP)
      (IP (NP #1#) #2#)))
```

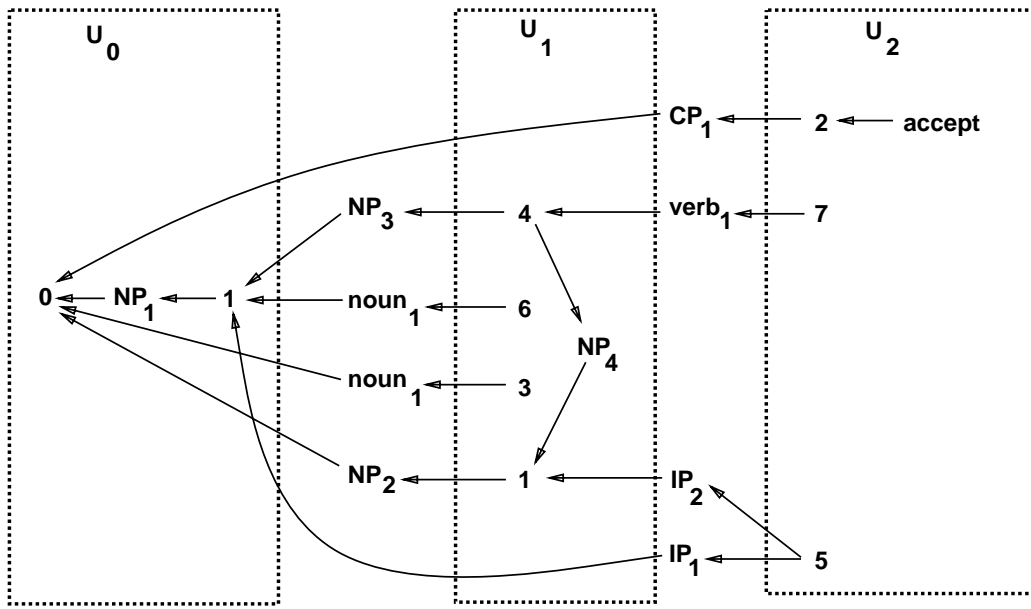


Figure 8: Correct graph-structured stack for Grammar 2 and the input sentence "noun verb."

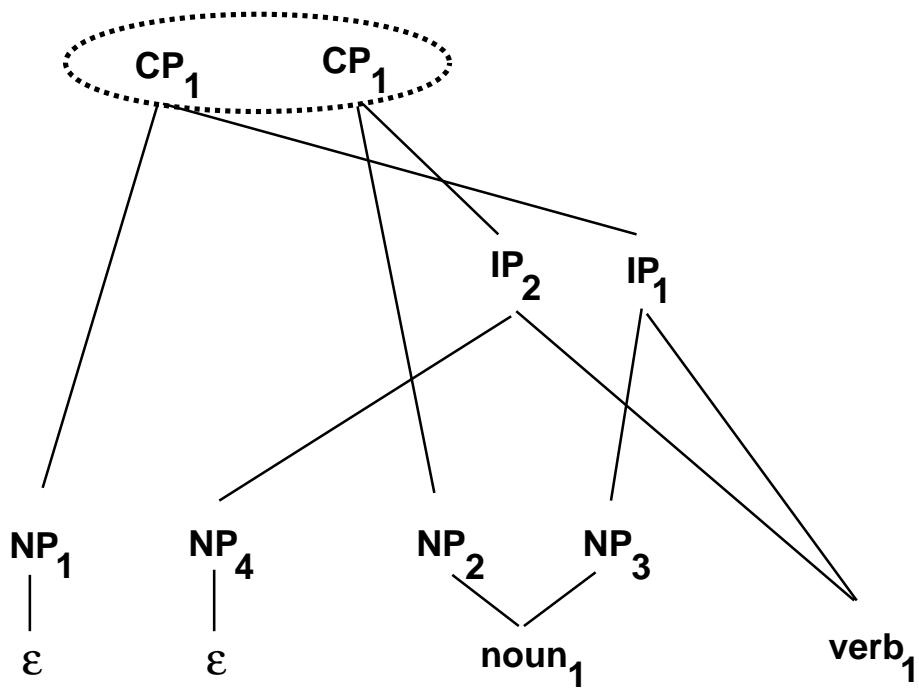


Figure 9: Correct parse forest for Grammar 2 and the input sentence "noun verb."

```

(CP
  ((C-BAR-SPEC
    ((POSSIBLY-EMPTY-NP
      (NP
        ((N-BAR-SPEC NIL)
          (N-BAR (#1=(N (FEATURES (WORD I) (CASE NOMINATIVE)
                                (PRONOMINAL T)))
                    (N-COMP NIL))))))))))
  (C-BAR
    ((C NIL)
     (C-COMP
      (IP
        ((I-BAR-SPEC NIL ((POSSIBLY-EMPTY-NP NIL))) .
         #2=((I-BAR
              ((I NIL)
               (I-COMP
                (#3=(VP
                    ((V-BAR-SPEC NIL)
                     (V-BAR
                      ((V
                        ((VERB
                          (FEATURES (WORD SAW) (EXT-ARG (AGENT))
                                       (INT-ARG (GOAL))
                                       (TENSE PAST))))
                         (V-COMP
                          ((POSSIBLY-EMPTY-NP
                            ((NP ((N-BAR-SPEC NIL)
                                   (N-BAR ((N (FEATURES (WORD JOHN))
                                               (N-COMP NIL))))))))))
                          (#3# (ADV NIL))))))))))))))
          ((C-BAR-SPEC NIL ((POSSIBLY-EMPTY-NP NIL)) ((ADV NIL)))
           (C-BAR
            ((C NIL)
             (C-COMP
              (IP
                ((I-BAR-SPEC ((POSSIBLY-EMPTY-NP
                              ((NP ((N-BAR-SPEC NIL)
                                     (N-BAR (#1# (N-COMP NIL)))))))) .
                               #2#))))))))))

```

Figure 10: Cyclic parse forest for the sentence “I saw John.”

8 Handling Cyclic Grammars

The presence of epsilon-productions may make a grammar “cyclic.”² If this is the case, Tomita’s algorithm (augmented with Nozohoor-Farshi’s ϵ -production handling scheme) may produce parse trees containing cycles. Our current s-structure grammar is in fact cyclic as illustrated by the parse forest for the sentence “I saw John” in figure 10. In that forest, the list #3# (corresponding to the VP “saw John”) contains a copy of itself. *I.e.*, a circular reference exists within the forest. This circular reference corresponds to infinite adjunction of an empty adverb to the VP. This infinite adjunction results from the adjunction rule “VP \rightarrow VP adv” and the adverb erasing rule “adv \rightarrow ϵ ” in our s-structure grammar. The infinite adjunction is undesirable since it serves no useful purpose, and its resulting circular reference complicates traversals of the forest.

To get rid of infinite adjunction in our parse forests, we simply delete all circular references in the forests output by the LR parser. This does not affect the parse forest’s correctness since the circular reference is not part of the GB analysis (even though the s-structure grammar is cyclic).

While our hybrid algorithm handled the above cyclic grammar problem (producing the circular parse forest discussed above), Nozohoor-Farshi’s recognizer, and therefore our hybrid algorithm, cannot handle arbitrary cyclic grammars. A more trouble-making cyclic feature of our s-structure grammar is that every terminal type can erase, and a VP can select a CP as an internal argument, so we can have CP $\stackrel{\dagger}{\Rightarrow}$ CP. The preferable solution to this problem is to extend Nozohoor-Farshi’s algorithm to handle arbitrary cyclic grammars. If that doesn’t work, we will have to use a scheme such as Fong’s (*i.e.*, to hard-code within the parser a limit on the level of CP nesting).

References

- [Fon91] Sandiway Fong. *Computational Properties of Principle-Based Grammatical Theories*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., 1991.

²A cyclic grammar is one where a grammar symbol can derive itself after a positive number of productions are applied, *i.e.*, $\alpha \stackrel{\dagger}{\Rightarrow} \alpha$.

- [NF91] Rahman Nozohoor-Farshi. GLR parsing for ϵ -grammars. In Masaru Tomita, editor, *Generalized LR Parsing*. Kluwer Academic Publishers, Dordrecht, 1991.
- [Sha93] Marwan Shaban. A Minimal GB Parser. Technical Report 93-013, Computer Science Department, Boston University, Boston, Mass., 1993.
- [Tom86] Masaru Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Dordrecht, 1986.
- [Tom91a] Masaru Tomita, editor. *Generalized LR Parsing*. Kluwer Academic Publishers, Dordrecht, 1991.
- [Tom91b] Masaru Tomita. The generalized LR parsing algorithm. In Masaru Tomita, editor, *Generalized LR Parsing*. Kluwer Academic Publishers, Dordrecht, 1991.