GENERALIZED ε-SKIP DISCRIMINATING-REVERSE PARSING ON GRAPH-STRUCTURED STACK

Fortes Gálvez, José
Departamento de Informática y Sistemas
Universidad de Las Palmas de Gran Canaria, Spain
jfortes@dis.ulpgc.es

Jacques Farré
Laboratoire I3S
CNRS and Université de Nice-Sophia Antipolis, France
jf@essi.fr

Abstract

We introduce εDR automata, which determine next shift-reduce parsing actions from (typically very short) stack-suffix explorations, while avoiding to process ε-deriving nonterminals. We present their use with a Tomita-like graph-structured stack parser, resulting in acceptance of the general class of (reduced) context-free grammars.

1 Introduction

In a previous paper [8] we directly apply discriminating-reverse, DR, automata [6, 7] to parse context-free grammars [3], CFG, in combination with a Tomita-like graph-structured stack, GSS [20]. However, the method shown there cannot handle grammars showing hidden left recursion [15].

DR(k) accepts the full class of LR(k) grammars [12], while producing efficient deterministic shift-reduce parsers of small size [5, 7]. Parsing actions are determined by efficient stack-suffix explorations, with an average depth usually smaller than 2 symbols.

In this paper we introduce εDR as an alternative underlying parser, which allows to avoid processing problematic ε-deriving nonterminals, in order to extend the method to the general class of reduced CFG.

Notation will be mostly conventional; see [19] for instance. The original (reduced) CFG G will be augmented, with $P' = \{S' \rightarrow \vdash S \dashv\} \cup P$. Productions in P' are numbered from 1 to $|P'|$, and g-th production is sometimes represented by $A \xrightarrow{g} \alpha$. Parsing action p indicates by convention the production number to use in a reduction if $p = g > 0$, or to shift if $p = 0$. We say that A is *null* (resp. *nullable*) if First(A) = $\{\varepsilon\}$ (resp. First(A) $\supset \{\varepsilon\}$). We note with a bar, as in $\bar{\alpha}$, the (possibly empty) string resulting after removal of all null nonterminals in α. We shall use letter ξ to denote some (possibly empty) string of nullable nonterminals.

2 ε-skip Discriminating-Reverse Construction

(Nondeterministic) DR parsers, as well as our εDR parsers, are shift-reduce parsers which determine the next set of legal parsing actions with the help of a stack-suffix exploration automaton. The automaton is started from the different stack tops, and it keeps track of the set of parsing actions which are compatible with the stack suffix already read.

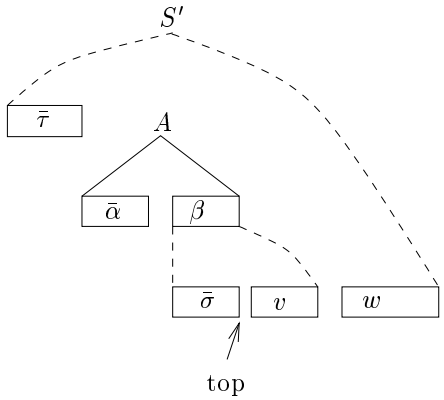

Figure 1: ε-free tree for εDR(0) item $[\dot{p}, A, \bar{\alpha}]$

Differently from DR, our εDR parsers perform no reduction from ε, i.e., every stack nonterminal actually derives some ε-free section of the input string. Accordingly, no production $A \to \varepsilon$ is effectively considered. Furthermore, the construction considers all the possibilities for nonterminals deriving ε, and thus their possible absence from the stack contents. The resulting construction can be seen as a virtual grammar transformation where every nullable nonterminal is both kept and removed from the grammar productions, and where every null nonterminal is removed. A similar idea of "skipping" ε-deriving nonterminals can be found in other general parsers [9, 14, 15].

2.1 The εDR(0) Automaton

In order to simplify the exposition, we shall only consider case $k = 0$.

An εDR(0) automaton $\mathcal{A}_0$ is a deterministic finite-state automaton, DFA, where each state is associated to a set of compatible actions, such that, if fully developed, would read the whole stack from its top, and accept precisely the language of all legal stack contents.

Each automaton state q_i is associated with a set of εDR(0) items. A generic item $[\dot{p}, A, \bar{\alpha}]$ indicates that next stack symbols to explore are those in $\bar{\alpha}$ from right to left (see Fig. 1, where $\bar{\sigma}$ represents the stack suffix already explored), and then those in all legal stack prefixes $\bar{\tau}$ on the left of A, i.e., only productions of the form $B \to \gamma A \delta$ are considered. Finally, $\dot{p}$ notes a dotted parsing action, which may take the forms "$p\bullet$" or "$\bullet p$". The dot[1] on the right indicates that the corresponding p-th production's right-hand side to reduce has not been fully explored yet. Otherwise the dot is written on the left (this includes by convention the shift action, i.e., $p = 0$).

State item sets I_{q_i} can be computed from a reduced grammar as follows:

- Initial state:

$$I_{q_0} = \{[\dot{p}, A, \bar{\alpha}] \mid A \xrightarrow{g} \alpha\beta \in P',\ \bar{\alpha} \neq \varepsilon\},$$

  where $\dot{p}$ is "$g\bullet$" if $\bar{\beta} = \varepsilon$, and "$\bullet 0$" otherwise. Note that all nonempty null-free prefixes of the right hand sides are considered. See again Fig. 1, considering $\bar{\sigma} = \varepsilon$.

- Transition function:

$$\Theta(I, X) = \mathcal{C}_0(\{[\dot{p}, A, \alpha] \mid [\dot{p}, A, \alpha X \xi] \in I\}).$$

[1] We use the notational convention of not showing the action dot position when it results unchanged or is irrelevant.

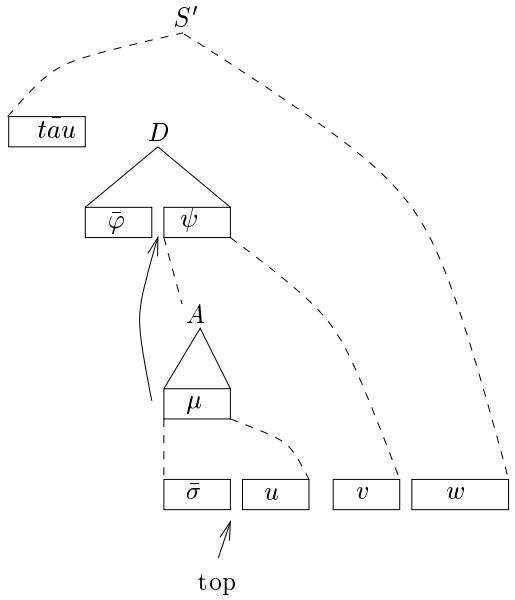

Figure 2: ε-free tree illustrating completer function $\mathcal{C}_0$ for item $[\dot{p}, A, \xi]$

Here we consider all possible ξ actually deriving ε as candidates to skip. Completer function $\mathcal{C}_0$ is defined by

$$\mathcal{C}_0(I) = I \cup \{[\bullet p, D, \bar{\varphi}] \mid [\dot{p}, A, \xi] \in I,\ D \to \varphi\psi \in P',\ \psi \Rightarrow^* Av,\ \bar{\varphi} \neq \varepsilon\}.$$

Again, we skip, if possible, ε-deriving right-hand-side prefixes in order to compute new items which permit to pursue stack exploration after ascent in the parsing tree (see Fig. 2).

In practical grammars, a small section of automaton $\mathcal{A}_0$ will normally suffice to determine parsing actions. In general, it may stop as soon as further stack exploration cannot refine the parsing-action set of current state. In fact, the construction algorithm *begins* building such an automaton, but only generates its useful discriminating (and full right-hand side; see below) section. This mechanism dramatically prunes the automaton, while preserving its discriminatory power. However, since only symbols deriving some nonempty input section will be present on top of the stack, in general there are several possible ε-free handles[2] to reduce using a same production. State construction continues until the longest possibility is verified before triggering the reduction of such handles.

2.2 An εDR(0) Generation Algorithm

Figure 3 shows the new version of the grammar compilation algorithm. A parsing table *StTbl* is built in order to explore some stack suffix from its top. Item sets I are computed, and their corresponding *StTbl* rows q_I are built as needed. Transitions on next stack symbols to (possibly new) states are built as far as it is still possible to further discriminate amongst the different parsing actions in the set, and to guarantee that all possible ε-free handles which are compatible with the reduce actions are completely explored. Eventually, the decision to (nondeterministically) perform that set of actions is taken on next stack symbol.

[2] We say that a nonempty stack-suffix ρ is an (ε-free) handle compatible with some right-hand side α if after removing zero or more nullable nonterminal occurrences from $\bar{\alpha}$ we obtain ρ.

algorithm Generator of nondeterministic εDR(0) parsing tables with right-hand side check
input Reduced context-free grammar G
output Table $StTbl$(*st-state*, *symb*)
begin
  $StAut := \{I_{q_0}\}$; $StTbl := \emptyset$;

  for every new $I \in StAut$ **do**
    for $X \mid \exists[\dot{p}, A, \alpha X\xi] \in I$ **do**
      if $\forall[p\bullet, A, \alpha X\beta] \in I : \beta \Rightarrow^* \varepsilon = \alpha$ **then**

        % decisions upon stack symbol (if same context for all actions)
        $St := \{p \mid [\dot{p}, A, \alpha X\xi] \in I\}$
        if $\forall A\alpha : \{p \mid [\dot{p}, A, \alpha X\xi] \in I\} \in \{St, \emptyset\}$
        then $StTbl\,(q_I, X) := St$
        else

          % state transitions upon stack symbol
          $I' := \Theta(I, X)$; add I' to $StAut$
          $StTbl(q_I, X) := \{\text{go-st } q_{I'}\}$
end

Figure 3: Compilation algorithm for GεDR(0) parsing

$S' \xrightarrow{1} \vdash S \dashv \quad S \xrightarrow{2} x \quad S \xrightarrow{3} BSb \quad S \xrightarrow{4} ASb \quad B \xrightarrow{5} AA \quad A \xrightarrow{6} \varepsilon$

	$\vdash$	S	$\dashv$	x	b
q_0	0	0	q_1	2	q_2
q_1		q_3			
q_2		3,4			
q_3	1				

Figure 4: Grammar G_{hlr} and its εDR(0) parsing table

In order to illustrate the method, Fig. 4 shows a simple grammar with hidden left recursivity, and its εDR(0) parsing table.[3] Note that A and B are null nonterminals, and thus nor they neither their rules are considered for the construction. Accordingly, the corresponding state item sets are the following:

$$
\begin{aligned}
I_{q_0} &= \{[1\bullet, S', \vdash S \dashv], [2\bullet, S, x], [3\bullet, S, Sb], [4\bullet, S, Sb], [\bullet 0, S', \vdash], [\bullet 0, S', \vdash S], [\bullet 0, S, S]\},\\
I_{q_1} &= \{[1\bullet, S', \vdash S]\},\\
I_{q_2} &= \{[3\bullet, S, S], [4\bullet, S, S]\},\\
I_{q_3} &= \{[1\bullet, S', \vdash]\}.
\end{aligned}
$$

States q_1 and q_3 only serve to verify the right-hand side of rule 1. In state q_2, it is clear that, beyond stack symbol S, actions 3 and 4 will follow exactly the same language of stack suffixes[4] on the left of left-hand side S, so we can decide here to perform both actions and to stop construction.

[3] The size of the εDR(0) automaton can be compared to the 10-state LALR(1) automaton for the same grammar (named G_8) in [16].

[4] In fact that same conclusion could have been reached already in q_0: state q_2 serves in fact to end the verification of ε-free handle Sb.

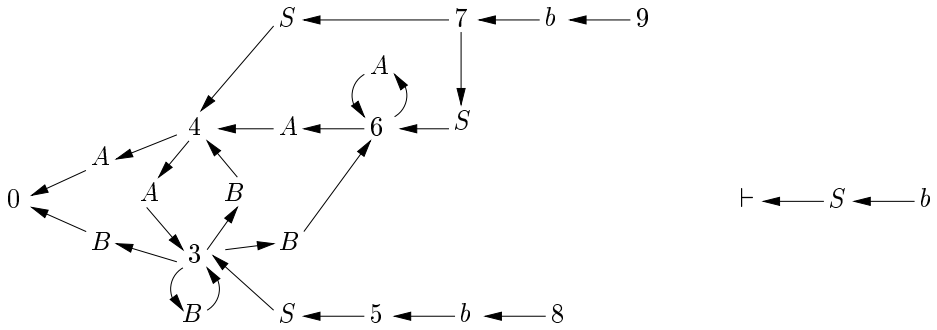

Figure 5: GLR GSS (left) and GεDR GSS (right) for grammar G_{hlr} after shift of first b

3 Generalized ε-skip Discriminating-Reverse Parsing

Since GSS are relatively more intuitive than other constructions, we present them in combination with our εDR(0) automata in order to parse context-free grammars. We show, for the sake of completeness, the algorithm [8] for parsing and for computing the parse (shared) forest, where we have included some minor adaptations due to the form of εDR(0) parsing tables and to the ε-free forests.

Differently from GLR parsers, our GSS do not contain automaton states, but plain grammar symbols. This often results in a drastic simplification of the GSS compared to GLR. For example, for grammar G_{hlr}, GLR needs first to build a relatively complex GSS for ε-deriving viable prefixes $\varepsilon + (A + B)^*(A + AA + B)$, what is not required for GεDR. Figure 5 shows the graphs for a GLR parser (left, according to the LALR(1) table in [16]) and for our GεDR(0) parser (right) after the input xb has been read.

3.1 GSS-Based εDR(0) Parsing

In our algorithm (see Fig. 6) a node ν of the GSS is a tuple [*symbol*, *set of predecessor nodes*, *set of right-hand sides*]. Obviously, the third component is empty for nodes of terminal symbols. Current exploration points in the GSS are represented by tuples $[\nu, \nu_t, q]$, where ν is the next node to check, ν_t the topmost node from which the exploration started, and q the next εDR(0) automaton state.

The underlying εDR(0) automaton uses the parsing table previously generated from a context-free grammar. State item sets themselves need not to be stored at parsing time, and thus state codes are used instead.

Before each input-shift step is performed, the GεDR parser has to start exploring, from the current tops of the GSS, all the possible paths allowed by the εDR(0) automaton. The same steps as a single-stack parser are followed in parallel[5], eventually performing the corresponding reduce actions on the GSS. Stack tops will be merged if they correspond to the same symbol and have exactly the same predecessors *Pred* in the GSS. Successive exploration phases are restarted from new GSS tops, until no more reductions can be performed and the only remaining action is to shift. The next input terminal is jointly shifted for the different tops.

As a result of this construction, all right-hand sides associated to a same node actually derive the same section of the input text, i.e., they are in local ambiguity. Although the notion of predecessor set is more natural when using a GSS, our construction allows to replace it by a single index pointing

[5] Blank parsing table entries simply result in abandoning current exploration line.

```
algorithm Generalized ε-skip discriminating-reverse parser
input εDR(0) parsing table StTbl and input string ⊢z⊣
output Forest of non-ε deriving parsing trees rooted at reduced, or rejection on erroneous input
begin
  shifttops := ∅

  repeat % parsing actions on next input symbol
    read (a); ν0 := NewNode(a); Pred(ν0) := shifttops
    currtops := {ν0}; shifttops := ∅; reduced := ∅

    repeat % GSS explorations while reductions are done
      reductions := ∅
      curr := {[νt, νt, q0] | νt ∈ currtops}

      repeat % check next symbol
        next := ∅
        for [ν, νt, q] ∈ curr do
          at := StTbl(q, Symb(ν))
          if at = {go-st q'} then for ν' ∈ Pred(ν) do add [ν', νt, q'] to next
          else
            if 0 ∈ at then add νt to shifttops
            for g ∈ at−{0} do add [g, ν, νt] to reductions
        curr := next;
      until curr = ∅

      currtops := ∅ % compute new tops from reductions and shared forest
      for [g, ν, νt] ∈ reductions and A ⟶g α do
        for [H, π] ∈ handle-corr(g, ν, νt) and νl = 1 : π do
          if ∃ν't ∈ reduced | Symb(ν't) = A and Pred(ν't) = Pred(νl) then
            add [g, H, π] to SD(ν't)        % node merge
          else
            ν't := NewNode(A); Pred(ν't) := Pred(νl); SD(ν't) := {[g, H, π]}
            add ν't to reduced, currtops
      until currtops = ∅

  until shifttops = ∅
  if ∃ν ∈ reduced | Symb(ν) = S' then accept else reject input % error
end
```

Figure 6: Gε DR(0) parsing algorithm

to the right end of the text section covered by the symbols in the predecessor set.

During the process, as in GLR, some GSS sections are eventually found to be nonviable and discarded together with their corresponding forest sections.[6] However, differently from GLR, increased GSS sharing may introduce possible paths that do not correspond to viable prefixes.

3.2 Shared Forest Computation

For a reduction with $A \xrightarrow{g} X_1 \cdots X_n$, function *handle-corr* (g, ν, ν_t) returns the set of all pairs $[H, \pi]$, where π are node sequences corresponding to g-compatible (ε-free) handles in the paths from ν_t to ν, and $H \subseteq \{1, \ldots, n\}$ codes the correspondence between right-hand side symbols and (ε-free) handle nodes.[7] Some of these cases may later be found incompatible with the rest of the input, and will

[6] Since nodes from incorrect paths will eventually become inaccessible, their deletion is implicit and can be left to a garbage collector.

[7] If we note $j_1 < \cdots < j_{|H|}$ for all $j_h \in H$ and $\pi = \nu_1 \cdots \nu_{|H|}$, then ν_h corresponds to X_{j_h}.

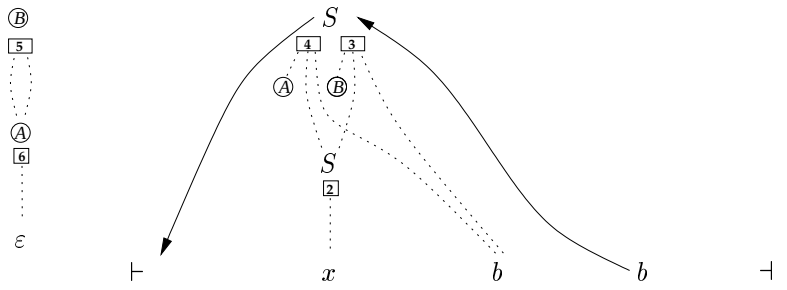

Figure 7: ε-subforest model (for circled nodes), and ε-free parse forest for input xbb (grammar G_{hlr})

thus be implicitly removed.[8] In the algorithm, ν_l represents the leftmost node of these compatible handles π. Not all predecessors of ν_l are necessarily legal, since they can be on a path that is not a correct suffix. Computation of predecessors can be improved, with a small cost, in order to avoid to keep paths that have been found illegal during the GSS exploration, by eliminating nodes from the *Pred* component of the leftmost node on a legal path. However, such paths cannot lead to wrong constructions, and they should typically be eliminated rapidly from next reductions. Thus, it is not clear that such an improvement would lead to a more efficient parsing.

Nodes of the GSS are easily and naturally reused for the packed shared forest representation. Single derivations from a node are represented by its corresponding set *SD* containing [*production-number*, *symbol-node correspondence*, *handle-nodes*] triples.[9]

In the end, the different parses of the input text can be easily and efficiently recovered from the single derivations starting from the top node, whose symbol is S' if the input is a legal sentence.

4 Example of Parsing and Forest Construction

Let us first follow the construction steps of the ε-free section of the parsing forest for an example sentence xbb (see again G_{hlr} parsing table of Fig. 4):

1. The stack is initialized with the bottom marker $\vdash$, which decides in initial state q_0 to shift. Exploration is restarted from q_0 using the new topmost symbol x, which indicates to reduce using production $S \xrightarrow{2} x$. Then, topmost S decides to shift in q_0. The GSS at this point is shown by the right graph of Fig. 5.

2. Now, reverse exploration of suffix Sb decides reductions $S \xrightarrow{3} BSb$ and $S \xrightarrow{4} ASb$. Accordingly, function *handle-corr* returns, for both reductions, $[\{2,3\}, \nu_S \nu_b]$, indicating that ε-deriving symbols B and A, respectively, have not been reduced. Thus both reductions are performed for a same new S node having $\{\vdash\}$ as *Pred* set. After shifting b we have GSS and forest shown[10] in Fig. 7.

3. The previous process would be likewise repeated in case of more b's. As a last step, stack suffix $\vdash S \dashv$ is found compatible with action set $\{1\}$, and thus parse successfully ends.

[8] [15] also introduce this form of nondeterminism. A theoretical, but probably impractical, alternative to avoid this effect is to apply the discrimination process among all possible ε-free handles instead of productions.

[9] Strictly, *handle-corr* computation could be left to a post-parsing phase, specially if only a small section of the trees in the forest will be processed by the application. In this case, *SD* could instead contain the arguments of *handle-corr*.

[10] Solid lines in the figure correspond to the *Pred* relation, and dotted lines to the shared forest.

$S' \xrightarrow{1} \vdash S \dashv \quad S \xrightarrow{2} A \quad S \xrightarrow{3} S \quad A \xrightarrow{4} SS \quad A \xrightarrow{5} a \quad A \xrightarrow{6} \varepsilon$

	$\vdash$	S	$\dashv$	A	a
q_0	0	q_1	q_2	2	5
q_1	0,3,4	0,3,4			
q_2	1	q_3			
q_3	1				

Figure 8: Grammar G_{ia} and its εDR(0) parsing table

Of course, the resulting forest lacks the ε-deriving sections. In the case of nonterminals with a finite number of possible ε-deriving trees, it is trivial to pre-compile their ε-subforest models and to attach them if needed to the parse forest. In the case of nonterminals with infinitely many possible ε-deriving trees, the convenient representation might be application-dependent. Nevertheless, it is again trivial —and should usually suffice— to produce compact subforest models where each nonterminal is represented by only one node, as shown in Fig. 7 for our example grammar.

Another Example

Consider now infinitely ambiguous grammar G_{ia}, which is shown with its corresponding GεDR parsing table in Fig. 8. In this case, there is no null but two nullable nonterminals —A and S. Accordingly, the corresponding state item sets are the following:

$$
\begin{aligned}
I_{q_0} &= \{[1\bullet, S', \vdash S \dashv], [2\bullet, S, A], [3\bullet, S, S], [4\bullet, A, SS], [5\bullet, A, a], [\bullet 0, S', \vdash], [\bullet 0, S', \vdash S], [\bullet 0, A, S]\}, \\
I_{q_1} &= \{[3\bullet, S, \varepsilon], [4\bullet, A, S], [4\bullet, A, \varepsilon], [\bullet 0, S', \vdash], [\bullet 0, A, \varepsilon]\} \\
&\quad \cup \{[\bullet 3, S', \vdash], [\bullet 3, A, S], [\bullet 4, S', \vdash], [\bullet 4, A, S], [\bullet 0, A, S]\}, \\
I_{q_2} &= \{[1\bullet, S', \vdash S]\}, \\
I_{q_3} &= \{[1\bullet, S', \vdash]\}.
\end{aligned}
$$

The second subset of I_{q_1} is added after applying completer function $\mathcal{C}_0$. Note that, in q_2, action 1 can be decided on $\vdash$, since S is nullable. Again, it is useless to continue construction beyond q_1, since action set $\{0, 3, 4\}$ cannot be further refined.

Figure 9 shows the final shared forest for input aa, including the different GSS at different times during parsing.[11]

After reducing a to A, exploration is restarted from q_0, indicating reduction $S \xrightarrow{2} A$. Now, stack $\vdash S$ is found compatible with action set $\{0, 3, 4\}$, what results in S being considered in the *Pred* set for next topmost a, and in the reductions shown in the figure.[12] Eventually, stack suffix SS is found compatible with action set $\{0, 3, 4\}$. The latter reduction has now also SS as possible ε-free handle[13] —apart from (topmost) S— resulting in a new A-node, since its *Pred* set ($\vdash$ node) is different from the lower A-node *Pred* set (left-hand node S).

[11] The corresponding Nozohoor-Farshi GLR construction, for instance, would result in a too complicated figure to be shown here.

[12] Note that reduction $A \xrightarrow{4} SS$ has in fact a single ε-free handle S with two possible correspondences in the right-hand side. Accordingly, function *handle-corr* returns $[\{1\}, \nu_S]$ and $[\{2\}, \nu_S]$.

[13] Corresponding to $[\{1, 2\}, \nu_S \nu'_S]$.

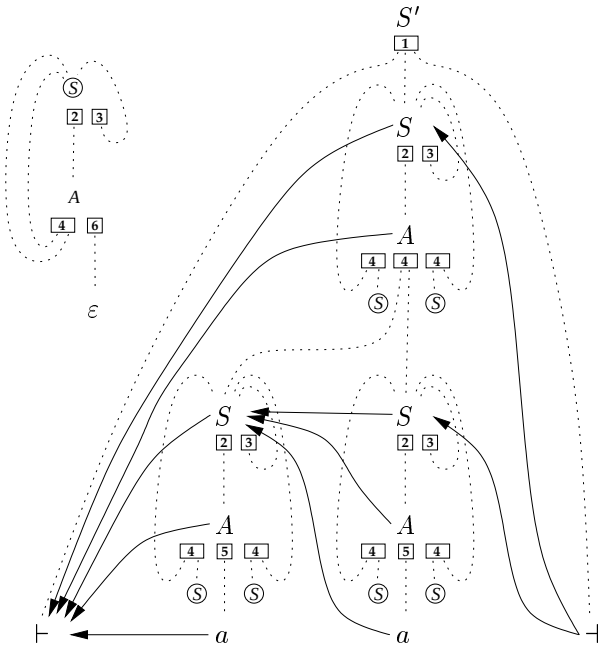

Figure 9: ε-subforest model (for circled nodes), and ε-free parse forest for input aa (grammar G_{ia})

5 Evaluation and Conclusion

General CFG parsers [10, 18] have historically received considerable attention. Amongst the most acknowledged techniques we can cite the Earley [4], CYK [17, 21], and GHR [9]. More recently, natural language applications have given place to GLR [20, 16] —although its theoretical base can be traced back to [13]—, which is also being adopted, in some cases with restrictions, for programming and specification languages, e.g. [1].

Unrestricted GLR, and in particular the use of a GSS, has the disadvantage of requiring [11], in the worst case, $\mathcal{O}(n^{p+1})$ time, where p is the size of the longest right-hand side. However, it has been shown that a grammar transformation into Chomsky Normal Form [3] or, equivalently, sharing of right-hand side sections, e.g. [11], allows to attain $\mathcal{O}(n^3)$. In our parser, we have the additional cost of exploring stack suffixes. We conjecture that the small average exploration depth characteristic of deterministic DR parsers will mostly hold for non-LR grammars. However, our explorations tend to "develop" the different possibilities, so it is unclear whether the above solutions would be useful for our method.

On the other hand, GεDR parsers present some very particular advantages. There is no need to store stack states, but plain vocabulary symbols. This considerably reduces the complexity of the GSS or alternative structures. The automaton tables are very small, in particular for $k > 0$ in comparison with LR(k). Moreover, it has been shown [2] that increasing the deterministic LR power in fact reduces efficiency because it increases the stack-state complexity, but this is not the case for GεDR, where we can improve deterministic discrimination without increasing the complexity of the GSS at a relatively low cost. Finally, in most practical applications, specially in programming languages, sentences are not highly ambiguous. A clever implementation should allow to parse unambiguous sections without excessive inefficiency with respect to a deterministic parser, what is hardly the case

for non-compiling methods.

In conclusion, our GεDR parsers accept in a natural way unrestricted context-free grammars, with simpler and cycle-free GSS. This new approach to CFG parsing deserves further research, from both theoretical and practical viewpoints, in order to evaluate its interest as an alternative to more conventional general parsers.

Although we have only presented the $k = 0$ case, we expect that some $k > 0$ will show to be better in practice. Finally, while the construction shown minimizes nondeterminism, it is still possible to use a "pruned" construction resulting in parsers showing more nondeterminism. On the other hand, there is the possibility to further reduce nondeterminism by using a correct-prefix variant, at the cost of a larger automaton.

References

[1] J. Aycock, R. N. Horspool, J. Janousek, and B. Melichar. Even faster generalized LR parsing. *Acta Informatica*, 37(9):633–651, 2001.

[2] S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 143–151. ACL, 1989.

[3] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2:136–167, 1959.

[4] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, Feb. 1970.

[5] J. Fortes Gálvez. Experimental results on discriminating-reverse LR(1) parsing. In P. Fritzson, editor, *Proceedings of the Poster Session of CC'94 - International Conference on Compiler Construction*, pages 71–80. Department of Computer and Information Science, Linköping University, Mar. 1994. Research report LiTH-IDA-R-94-11.

[6] J. Fortes Gálvez. A practical small LR parser with action decision through minimal stack suffix scanning. In J. Dassow, G. Rozenberg, and A. Salomaa, editors, *Developments in Language Theory II*, pages 460–465. World Scientific, 1996.

[7] J. Fortes Gálvez. *A Discriminating Reverse Approach to LR(k) Parsing*. PhD thesis, Universidad de Las Palmas de Gran Canaria and Université de Nice-Sophia Antipolis, 1998.

[8] J. Fortes Gálvez and J. Farré. Practical nondeterministic DR(k) parsing on graph-structured stack. In A. Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, volume 2004 of *Lecture Notes in Computer Science*, pages 411–422. Springer, 2001.

[9] S. L. Graham, M. A. Harrison, and W. L. Ruzzo. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, July 1980.

[10] D. Grune and C. J. H. Jacobs. *Parsing Techniques*. Ellis Horwood, 1990.

[11] J. R. Kipps. GLR parsing in time O(n^3). In M. Tomita, editor, *Current Issues on Parsing Technology*, pages 182–192. Kluwer, 1991.

[12] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[13] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Automata, Languages and Programming*, volume 14 of *Lecture Notes in Compter Science*, pages 255–269. Springer-Verlag, Berlin, 1974.

[14] R. Leermakers. A recursive ascent Earley parser. *Information Processing Letters*, 41(2):87–91, 1992.

[15] M.-J. Nederhof and J. J. Sarbo. Increasing the applicability of LR parsing. In H. Bunt and M. Tomita, editors, *Recent Advances in Parsing Technology*, pages 35–57. Kluwer, 1996.

[16] R. Nozohoor-Farshi. Handling of ill-designed grammars in Tomita's parsing algorithm. In M. Tomita, editor, *Current Issues on Parsing Technology*, pages 182–192. Kluwer, 1991.

[17] I. Sakai. Syntax in universal translation. In *Proceedings 1961 International Conference on Machine Translation of Languages and Applied Language Analysis*, pages 593–608. Her Majesty's Stationary Office, London, 1962.

[18] K. Sikkel. *Parsing Schemata*. Springer, 1997.

[19] S. Sippu and E. Soisalon-Soininen. *Parsing Theory*. Springer, 1988 and 1990.

[20] M. Tomita. *Efficient Parsing for Natural Language*. Kluwer, 1985.

[21] D. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.