Extended Context-Free Grammars Parsing with Generalized LL

Artem Gorokhov and Semyon Grigorev

Saint Petersburg State University
7/9 Universitetskaya nab.
St. Petersburg, 199034 Russia
gorohov.art@gmail.com
semen.grigorev@jetbrains.com

Abstract. Parsing plays an important role in static program analysis: during this step a structural representation of code is created upon which further analysis is performed. Parser generator tools, being provided with syntax specification, automate parser development. Language documentation often acts as such specification. Documentation usually takes form of ambiguous grammar in Extended Backus-Naur Form which most parser generators fail to process. Automatic grammar transformation generally leads to parsing performance decrease. Some approaches support EBNF grammars natively, but they all fail to handle ambiguous grammars. On the other hand, Generalized LL parsing algorithm admits arbitrary context-free grammars and achieves good performance, but cannot handle EBNF grammars. The main contribution of this paper is a modification of GLL algorithm which can process grammars in a form which is closely related to EBNF (Extended Context-Free Grammar). We also show that the modification improves parsing performance as compared to grammar transformation based approach.

Keywords: Parsing, Generalized Parsing, Extended Context-Free Grammar, GLL, SPPF, EBNF, ECFG, RRPG, Recursive Automata

1 Introduction

Static program analysis is usually performed over a structural representation of code and parsing is a classical way to get such representation. Parser generators are often used to automate parser creation: these tools derive a parser from a grammar.

Extended Backus-Naur Form [?] is a metasyntax for expressing context-free grammars. In addition to the Backus-Naur Form syntax it uses the following constructions: alternation |, optional symbols [...], repetition {...}, and grouping (...).

This form is widely used for grammar specification in technical documentation because expressive power of EBNF makes syntax specification more compact and human-readable. Because documentation is one of the main sources of data for parsers developers, it would be helpful to have a parser generator which supports grammar specification in EBNF. Note, that EBNF is a standardized notation for *extended context-free grammars* [?] which can be defined as follows.

Definition 1 *An* ***extended context-free grammar*** *(ECFG) [?] is a tuple* (N, Σ, P, S)*, where* N *and* Σ *are finite sets of nonterminals and terminals respectively,* $S \in N$ *is the start symbol, and* P *(productions) is a map from* N *to regular expressions over alphabet* $N \cup \Sigma$*.*

ECFG is widely used as an input format for parser generators, but classical parsing algorithms often require CFG, and, as a result, parser generators usually require conversion to CFG. It is possible to transform ECFG to CFG [?], but this transformation leads to grammar size increase and change in grammar structure: new nonterminals are added during transformation. As a result, parser constructs derivation tree with respect to the transformed grammar, making it harder for a language developer to debug grammar and use parsing result later.

There is a wide range of parsing techniques and algorithms [?,?,?,?,?,?,?,?] which are able to process grammar in ECFG. Detailed review of results and problems in ECFG processing area is provided in the paper "Towards a Taxonomy for ECFG and RRPG Parsing" [?]. We only note that most of algorithms are based on classical LL [?,?,?] and LR [?,?,?] techniques, and they admit only restricted subclasses of ECFG. Thus, there is no solution for handling arbitrary (including ambiguous) ECFGs.

The LL-based parsing algorithms are more intuitive than LR-based and can provide better error diagnostic. Currently LL(1) seems to be the most practical algorithm. Unfortunately, some languages are not LL(k) for any k, and left recursive grammars are a problem for LL-based tools. Another restriction for LL parsers is ambiguities in grammar which, being combined with previous flaws, complicates industrial parsers creation. Generalized LL, proposed in [?], solves all these problems: it handles arbitrary CFGs, including ambiguous and left recursive. Worst-case time and space complexity of GLL is cubic in terms of input size and, for LL(1) grammars, it demonstrates linear time and space complexity.

In order to improve performance of GLL algorithm, modification for left factorized grammars processing was introduced in [?]. Factorization transforms grammar so that there are no two productions with same prefixes (see fig 1 for example). It is shown, that factorization can reduce memory usage and increase performance by reusing common parts of rules for one nonterminal. Similar idea can be applied to ECFGs processing.

To summarize, if it were possible to handle ECFG specification with tools based on generalized parsing algorithm, it would greatly simplify language development. In this work we present a modification of generalized LL parsing algorithm which handles arbitrary ECFGs without any transformations. Also we demonstrate that proposed modifications improve parsing performance and memory usage comparing to GLL for factorized grammar.

2 ECFG Handling with Generalized LL Algorithm

The purpose of generalized parsing algorithms is to provide arbitrary context-free grammars handling. Generalized LL algorithm (GLL) [?] inherits properties of classical LL algorithms: it is more intuitive and provides better syntax error diagnostic than generalized LR algorithms. Also, our experience shows that GLR-based solutions are more complex than GLL-based, which agrees with the observation in [?] that LR-based ECFG parsers are very complex. Thus, we choose GLL as a base for our solution. In this section we present GLL-style parser for arbitrary ECFG processing.

2.1 Generalized LL Parsing Algorithm

An idea of the GLL algorithm is based on descriptors which can uniquely define state of parsing process. Descriptor is a four-element tuple (L, i, T, S) where:

- L is a grammar slot — pointer to a position in the grammar of the form $(S \rightarrow \alpha \cdot \beta)$;
- i — position in the input;
- T — already built node of parse forest;
- S — current Graph Structured Stack (GSS) [?] node.

GLL moves through the grammar and the input simultaneously, creating multiple descriptors in case of ambiguity, and using queue to control descriptors processing. In the initial state there is only one descriptor which consists of start position in grammar $(L = (S \rightarrow \cdot\beta))$, input $(i = 0)$, dummy tree node, and the bottom of GSS. At each step, the algorithm dequeues a descriptor and acts depending on the grammar and the input. If there is an ambiguity, then algorithm queues descriptors for all possible cases to process them later. To achieve cubic time complexity, it is important to enqueue only descriptors which have not been created before. Global storage of all created descriptors is used to filter descriptors which should be enqueued.

There is a table based approach [?] for GLL implementation which generates only tables for given grammar instead of full parser code. The idea is similar to the one in the original paper and uses the same tree construction and stack processing routines. Pseudocode illustrating this approach can be found in the appendix A. Note that we do not include check for first/follow sets in this paper.

2.2 Grammar Factorization

In order to improve performance of GLL, Elizabeth Scott and Adrian Johnstone proposed a support for left-factorized grammars in this parsing algorithm [?].

It is obvious from GLL description, that to decrease parse time and the amount of required memory, it is sufficient to reduce the number descriptors to process. One of the ways to do it is to reduce the number of grammar slots, and it can be done by grammar factorization. An example of factorization is provided in fig. 1: grammar P_0 transforms to P_0' during factorization. This example is discussed in the paper [?], and it is shown, that, by producing less slots, such transformation can improve performance significantly for some grammars.

$S ::= a\ a\ B\ c\ d \mid a\ a\ c\ d \mid a\ a\ c\ e \mid a\ a$

(a) Original grammar P_0

$S ::= a\ a\ (B\ c\ d \mid c\ (d \mid e) \mid \varepsilon\ )$

(b) Factorized grammar P_0'

Fig. 1. Example of grammar factorization

We can evolve this idea to support ECFG, and we will show how to do it in the next section.

2.3 Recursive Automata and ECFGs

In order to ease adoption of ideas of grammar factorization for handling ECFGs with GLL we use recursive automaton (RA) [?] for ECFG representation. We use the following definition of RA.

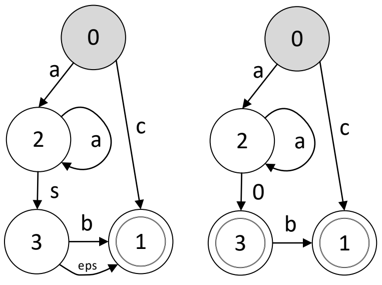
Definition 2 *Recursive automaton (RA) R is a tuple $(\Sigma, Q, S, F, \delta)$, where Σ is a finite set of terminals, Q — finite set of states, $S \in Q$ — start state, $F \subseteq Q$ — set of final states, $\delta : Q \times (\Sigma \cup Q) \to Q$ — transition function.*

The only difference between Recursive Automaton and Finite State Automaton (FSA) is that transitions in RA are labeled either by terminal (Σ) or by state (Q). Further in this paper, we call transitions by elements from Q *nonterminal transitions* and by terminal — *terminal transitions.*

Note that grammar factorization leads to partial minimization of automata in the right-hand sides of productions. Also note that grammar slots are equivalent to states of automata which are built from right-hand sides of productions. Right-hand sides of ECFG productions are regular expressions over the union alphabet of terminals and nonterminals. So, our goal is to build RA with minimal number of states for given ECFG, which can be done by the following steps.

1. Build an FSA using Thompson's method for each right-hand side of productions [?].
2. Create a map M from nonterminal to a corresponded start state. This map should be kept consistent during all the following steps.
3. Convert FSAs from previous step to a deterministic FSAs without ε-transitions using the algorithm described in [?].
4. Minimize DFSAs, for example, by using John Hopcroft's algorithm [?].
5. Replace transitions by nonterminals with transitions labeled by start states by using map M. Result of this step is a required RA. We also use map M to define function $\Delta : Q \to N$ where N is nonterminal name.

An example of ECFG to RA transformation is presented in fig. 2, where state 0 is the start state of resulting RA.



$S ::= a^{+} S\ b? \mid c$

(a) Grammar G_0 (b) FSA for G_0 (c) RA R_0 for G_0

Fig. 2. Grammar to RA transformation

2.4 Input Processing

In this section we describe changes required in control functions of basic GLL algorithm to handle ECFG. Main loop is similar to basic GLL one: at each step the main function **parse** dequeues next descriptor to be processed. Suppose that current descriptor is a tuple (C_S, C_U, C_i, C_N), where C_S — state of RA, C_U — GSS node, C_i — position in the input string ω, and C_N — SPPF node. It is possible to get the following nonexclusive cases during this descriptor processing.

- C_S is a final state. Perform pop action (call **pop** function), because processing of nonterminal is finished.
- There is a terminal transition $C_S \xrightarrow{\omega.[C_i]} q$. Move right: create descriptor with state q and position $(C_i + 1)$. Enqueue it regardless of whether it has been created before.
- There are nonterminal transitions from C_S. It means that processing of new nonterminal should be started, thus new GSS nodes should be created. To do it, **create** function should be called for each transition. It performs necessary operations with GSS and checks if there is an already created SPPF node for the current input position and nonterminal.

All required functions are presented below. Function **add** enqueues descriptor if it has not already been created, and this function has not been changed.

```
function CREATE($S_{call}, S_{next}, u, i, w$)
    $A \leftarrow \Delta(S_{call})$
    if ($\exists$ GSS node labeled $(A, i)$) then
        $v \leftarrow$ GSS node labeled $(A, i)$
        if (there is no GSS edge from $v$ to $u$ labeled $(S_{next}, w)$) then
            add a GSS edge from $v$ to $u$ labeled $(S_{next}, w)$
            for ($(v, z) \in \mathcal{P}$) do
                $(y, N) \leftarrow$ getNodes($S_{next}, u.nonterm, w, z$)
                if $N \neq \$$ then
                    $(\_, \_, h) \leftarrow N$
                    pop($u, h, N$)
                $(\_, \_, h) \leftarrow y$
                **add**$(S_{next}, u, h, y)$
    **else**
        $v \leftarrow$ **new** GSS node labeled $(A, i)$
        create a GSS edge from $v$ to $u$ labeled $(S_{next}, w)$
        **add**$(S_{call}, v, i, \$)$
    **return** $v$
**function** POP$(u, i, z)$
    **if** $((u, z) \notin \mathcal{P})$ **then**
        $\mathcal{P}.add(u, z)$
        **for all** GSS edges $(u, S, w, v)$ **do**
            $(y, N) \leftarrow$ **getNodes**$(S, v.nonterm, w, z)$
            **if** $N \neq \$$ **then** **pop**$(v, i, N)$
            **if** $y \neq \$$ **then** **add**$(S, v, i, y)$
**function** PARSE
    $R.add(StartState, newGSSnode(StartNonterminal, 0), 0, \$)$
    **while** $R \neq \varnothing$ **do**
        $(C_S, C_U, C_i, C_N) \leftarrow R.Get()$
        $C_R \leftarrow \$$
        **if** $(C_N = \$)\&(C_S$ is final state$)$ **then**
            $eps \leftarrow$ **getNodeT**$(\varepsilon, C_i)$
            $(\_, N) \leftarrow$ **getNodes**$(C_S, C_U.nonterm, \$, eps)$
            **pop**$(C_U, C_i, N)$
        **for each** $transition(C_S, label, S_{next})$ **do**
            **switch** $label$ **do**
                **case** $Terminal(x)$ where $(x = input[i])$
                    $R \leftarrow$ **getNodeT**$(x, C_i)$
                    $(y, N) \leftarrow$ **getNodes**$(S_{next}, C_U.nonterm, C_N, R)$
                    **if** $N \neq \$$ **then** **pop**$(C_U, i + 1, N)$
                    $R.add(S_{next}, C_U, i + 1, y)$
                **case** $Nonterminal(S_{call})$
                    **create**$(S_{call}, S_{next}, C_U, C_i, C_N)$
    **if** exists SPPF node $(StartNonterminal, 0, input.length)$ **then**
        return this node
    **else** report failure
```

2.5 Parse Forest Construction

Result of the parsing process is a structural representation of the input — a derivation tree, or parse forest in case of multiple derivations.

First, we should define derivation tree for recursive automaton: it is an ordered tree whose root is labeled with the start state, leaf nodes are labeled with terminals or ε, and interior nodes are labeled with nonterminals N and their children for a sequence of transition labels of a path in the automaton which starts from the state q_i, where $\Delta(q_i) = N$.

Definition 3 *Derivation tree of sentence α for the recursive automaton $R = (\Sigma, Q, S, F, \delta)$:*

- *Ordered rooted tree; root is labeled with $\Delta(S)$;*
- *Leaves are terminals $a \in \Sigma$;*
- *Nodes are nonterminals $A \in \Delta(Q)$;*
- *Node with label $N_i \in \Delta(q_i)$ has children $l_0 \dots l_n (l_i \in \Sigma \cup \Delta(Q))$ iff there exists a path $q_i \xrightarrow{l_0} q_{i+1} \xrightarrow{l_1} \dots \xrightarrow{l_n} q_m$, $q_m \in F$.*

For arbitrary grammars, RA can be ambiguous in terms of accepted paths, and, as a result, it is possible to get multiple derivation trees for one input string. Shared Packed Parse Forest (SPPF) [**?**] can be used as a compact representation of all possible derivation trees. We use the binarized version of SPPF, which is proposed in [**?**], in order to decrease memory usage and achieve cubic worst-case time and space complexity. Binarized SPPF can be used in GLL [**?**] and contains the following types of nodes (here i and j are the start and the end of derived substring in terms of positions in the input string):

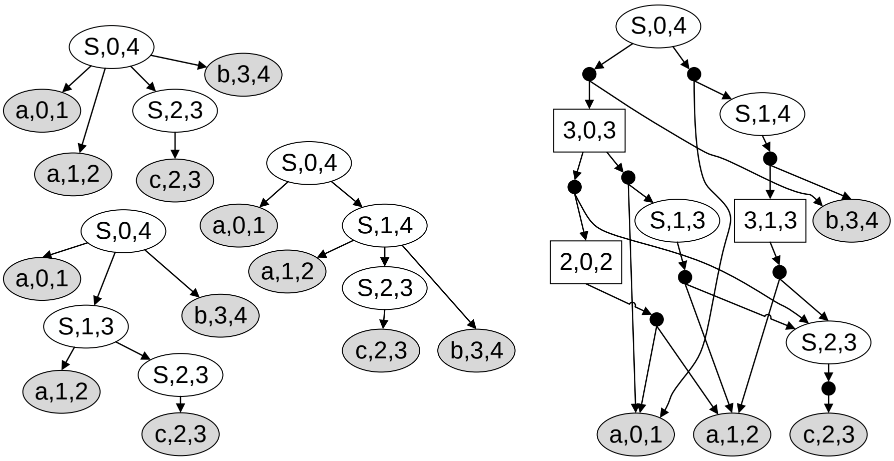
- Packed nodes are of the form (S, k), where S is a state of automaton, k — start of derived substring of right child. Packed node necessarily has a right child node — symbol node, and optional left child node — symbol or intermediate node.
- Symbol nodes have labels (X, i, j) where $X \in \Sigma \cup \Delta(Q) \cup \{\varepsilon\}$. Terminal symbol nodes ($X \in \Sigma \cup \{\varepsilon\}$) are leaves. Nonterminal nodes ($X \in \Delta(Q)$) may have several packed children nodes.
- Intermediate nodes have labels (S, i, j), where S is a state of automaton, and may have several packed children nodes.

Let us describe modifications of original SPPF construction functions. The function **getNodeT**(x, i) which creates terminal nodes is reused without any modification from basic algorithm. To handle nondeterminism in states, we define function **getNodes** which checks if the next state of RA is final and, if that is case, constructs nonterminal nodes in addition to the intermediate one. It uses modified function **getNodeP**: instead of grammar slot it takes separately a state of RA and symbol for new SPPF node: current nonterminal or the next RA state.

```
function GETNODES($S, A, w, z$)
    if ($S$ is final state) then
        $x \leftarrow$ getNodeP($S, A, w, z$)
    else $x \leftarrow \$$
    if ($w = \$$)& not ($z$ is nonterminal node and it's extents are equal) then
        $y \leftarrow z$
    else $y \leftarrow$ getNodeP($S, S, w, z$)
    return $(y, x)$
```

```
function getNodeP($S, L, w, z$)
    $(\_, k, i) \leftarrow z$
    if $(w \neq \$)$ then
        $(\_, j, k) \leftarrow w$
        $y \leftarrow$ find or create SPPF node labelled $(L, j, i)$
        if ($\nexists$ child of $y$ labelled $(S, k)$) then
            $y' \leftarrow$ new $packedNode(S, k)$
            $y'.addLeftChild(w)$
            $y'.addRightChild(z)$
            $y.addChild(y')$
    else
        $y \leftarrow$ find or create SPPF node labelled $(L, k, i)$
        if ($\nexists$ child of $y$ labelled $(S, k)$) then
            $y' \leftarrow$ new $packedNode(S, k)$
            $y'.addRightChild(z)$
            $y.addChild(y')$
    return $y$
```

Let us demonstrate an SPPF example for ECFG grammar G_0 (fig. 2a). This grammar contains constructions (option symbols and repetition) that should be converted with the use of extra nonterminals to build regular GLL parser. Our generator constructs recursive automaton R_0 (fig. 2c) and parser for it. Possible trees for input *aacb* are shown in fig. 3a. SPPF build by parser (fig. 3b) combines all of them.



(a) Possible derivation trees for R_0 and input *aacb* (b) SPPF for R_0 and input *aacb*

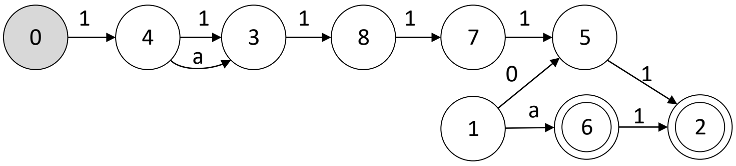
Fig. 3. Example for input *aacb*

3 Evaluation

We have compared our parsers built on factorized grammar and on minimized recursive automaton. Grammar G_1 (fig. 4a) was used for the tests, it has long tails in alternatives which are not unified with factorization. FSA built for this grammar is presented in fig. 4b.

$$S ::= K\ (K\ K\ K\ K\ K \mid a\ K\ K\ K\ K)$$
$$K ::= S\ K \mid a\ K \mid a$$

(a) Grammar G_1



(b) RA for grammar G_1

Fig. 4. Grammar G_1 and RA for it

For this grammar parser for RA should create less GSS edges because the tails of alternatives in producions are represented by the only path in RA. This fact leads to decrease of SPPF nodes and descriptors.

Experiments were performed on inputs of different length and are presented in fig. 5. Exact values for the input a^{40} are shown in the table 1.

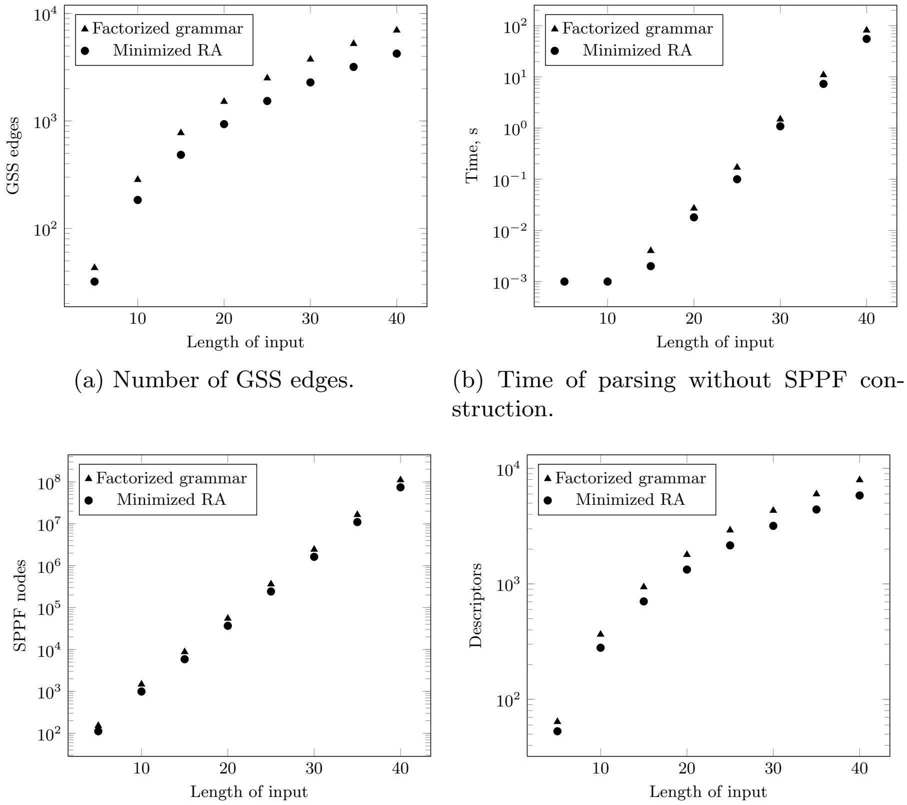
All tests were run on a PC with the following characteristics:

- OS: Microsoft Windows 10 Pro x64
- CPU: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, 3601 Mhz, 4 Cores, 4 Logical Processors
- RAM: 32 GB

	Time, s	Descriptors	GSS Edges	GSS Nodes	SPPF Nodes
Factorized grammar	81.814	7940	6974	80	111127244
Minimized RA	54.637	5830	4234	80	74292078

Table 1. Experiments results for input a^{40}

Results of performed experiments agree with the fact that on some grammars our approach shows better results than parsers built on factorized grammars. With grammar G_1 in general minimized RA version works 33% faster, uses 27% less descriptors, 29% less GSS edges and 33% less SPPF nodes.



(a) Number of GSS edges.

(b) Time of parsing without SPPF construction.

(c) Number of SPPF nodes.

(d) Number of descriptors.

Fig. 5. Experiments results.

4 Conclusion and Future Work

Described algorithm and parser generator based on it are implemented in F# programming language as a part of the YaccConstructor project. Source code is available here: `https://github.com/YaccConstructor/YaccConstructor`.

As we showed in evaluation, proposed modification not only increases performance, but also decreases memory usage. It is crucial for big input processing. For example, Anastasia Ragozina in her master's thesis [**?**] shows that GLL can be used for graph parsing. Some areas deal with big graphs, for example, metagenomic assemblies in bioinfomatics and social graphs. We hope that using the proposed modification we can improve performance of graph parsing algorithm too. We perform some tests that demonstrate performance increase in metagenomic analysis, but further integration with graph parsing is required.

One of the ways to specify semantic of language is attributed grammars, but it is not supported in the algorithm which is presented in this article. There is a number of works on subclasses of attributed ECFGs (for example [**?**]), however still there is no general solution for arbitrary ECFGs. Thus, arbitrary attributed ECFGs and semantic calculation support is a work for future.

Another question is a possibility of unification of our results with tree languages theory: our definition of derivation tree for ECFG is quite similar to unranked tree and SPPF is similar to automata for unranked trees [**?**]. Theory of tree languages seems to be more mature than theory of SPPF manipulations in general. Moreover some relations between tree languages and ECFG are discussed in the paper [**?**]. We hope that the investigation of relations between tree languages and SPPF may produce interesting results.

A GLL pseudocode

```
function ADD($L, u, i, w$)
    if $(L, u, i, w) \notin U$ then
        $U.add(L, u, i, w)$
        $R.add(L, u, i, w)$
function CREATE($L, u, i, w$)
    $(X ::= \alpha A \cdot \beta) \leftarrow L$
    if ($\exists$ GSS node labeled $(A, i)$) then
        $v \leftarrow$ GSS node labeled $(A, i)$
        if (there is no GSS edge from $v$ to $u$ labeled $(L, w)$) then
            add a GSS edge from $v$ to $u$ labeled $(L, w)$
            for ($(v, z) \in \mathcal{P}$) do
                $y \leftarrow$ getNodeP($L, w, z$)
                add($L, u, h, y$) where $h$ is the right extent of $y$
    else
        $v \leftarrow$ new GSS node labeled $(A, i)$
        create a GSS edge from $v$ to $u$ labeled $(L, w)$
        for each alternative $\alpha_k$ of $A$ do
            add($\alpha_k, v, i, \$$)
    return $v$
function POP($u, i, z$)
    if ($(u, z) \notin \mathcal{P}$) then
        $\mathcal{P}.add(u, z)$
        for all GSS edges $(u, L, w, v)$ do
            $y \leftarrow$ getNodeP($L, w, z$)
            add($L, v, i, y$)
function GETNODET($x, i$)
    if ($x = \varepsilon$) then  $h \leftarrow i$
    else $h \leftarrow i + 1$
    $y \leftarrow$ find or create SPPF node labelled $(x, i, h)$
     return $y$
function GETNODEP($X ::= \alpha \cdot \beta, w, z$)
    if ($\alpha$ is a terminal or a non-nullable nontermial) & ($\beta \neq \varepsilon$) then
```

```
        return $z$
    else
        if $(\beta = \varepsilon)$ then $L \leftarrow X$
        else $L \leftarrow (X ::= \alpha \cdot \beta)$
        $(\_, k, i) \leftarrow z$
        if $(w \neq \$)$ then
            $(\_, j, k) \leftarrow w$
            $y \leftarrow$ find or create SPPF node labelled $(L, j, i)$
            if $(\nexists$ child of $y$ labelled $(X ::= \alpha \cdot \beta, k))$ then
                $y\prime \leftarrow$ new $packedNode(X ::= \alpha \cdot \beta, k)$
                $y\prime.addLeftChild(w)$
                $y\prime.addRightChild(z)$
                $y.addChild(y\prime)$
        else
            $y \leftarrow$ find or create SPPF node labelled $(L, k, i)$
            if $(\nexists$ child of $y$ labelled $(X ::= \alpha \cdot \beta, k))$ then
                $y\prime \leftarrow$ new $packedNode(X ::= \alpha \cdot \beta, k)$
                $y\prime.addRightChild(z)$
                $y.addChild(y\prime)$
        return $y$
function DISPATCHER
    if $R \neq \varnothing$ then
        $(C_L, C_u, C_i, C_N) \leftarrow R.Get()$
        $C_R \leftarrow \$$
        $dispatch \leftarrow false$
    else $stop \leftarrow true$
function PROCESSING
    $dispatch \leftarrow true$
    switch $C_L$ do
        case $(X \to \alpha \cdot x\beta)$ where $(x = input[C_i] \parallel x = \varepsilon)$
            $C_R \leftarrow$ getNodeT$(x, C_i)$
            if $x \neq \varepsilon$ then $C_i \leftarrow C_i + 1$
            $C_L \leftarrow (X \to \alpha x \cdot \beta)$
            $C_N \leftarrow$ getNodeP$(C_L, C_N, C_R)$
            $dispatch \leftarrow false$
        case $(X \to \alpha \cdot A\beta)$ where $A$ is nonterminal
            create$((X \to \alpha A \cdot \beta), C_u, C_i, C_N)$
        case $(X \to \alpha \cdot)$
            pop$(C_u, C_i, C_N)$
function PARSE
    while not $stop$ do
        if $dispatch$ then dispatcher()
        else processing()
```

```
**if** exists SPPF node $(StartNonterminal, 0, input.length)$ **then**
    return this node
**else** report failure
```