

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 is important step of static program analysis. It allows to get structural representation of code. Parser generators are widely used for parser creation. EBNF is very popular for languages syntax description. But transformation to more simple form (BNF, CNF) is required for popular tools. There are number of works on EBNF processing without transformation. But problems. Generalized LL, arbitrary grammars in $O(n^s)$, factorization can increase performance... Factorization can be improved. We propose modification of GLL which can handle arbitrary grammar in EBNF without transformations... performance improvements,

Keywords: Parsing, GLL, SPPF, EBNF, ECFG, RRPg, Automata

1 Introduction

Static program analysis usually performed over structural representation of code and parsing is a classical way to get such representation. Parser generators often used for parser creation automation: these tools allow to create parser from grammar of language which should be specified in appropriate format. It allows to decrease efforts required for syntax analyzer creation and maintenance.

Extended BNF (EBNF) is a useful format of grammar specification. Expressive and compact description of language syntax. This formalism often used in documentation — one of main source of information for parsers developers.

There are a wide range of parsing techniques and algorithms: CYK, LR(k), LALR(k), LL, etc. One of the most popular area is generalized parsing: technique which allows to handle ambiguous grammars. It is possible to simplify language description required for parser generation in case a parser generator is based on generalized algorithm. LL family is more intuitive than LR, can provide better diagnostics, but LL(1) is not enough to process some languages: there are LR, but not LL languages. Moreover, left and hidden left recursion in grammars is a problem. In order to solve these problems generalized LL (GLL) was proposed [15]. This algorithm handles arbitrary context free grammar, even unambiguous and (hidden)left-recursive. Worst-case time and space complexity

of GLL is cubic in terms of input size. For LL grammars it demonstrates linear time and space complexity.

The problem is that classical parsing algorithms requires BNF. It is possible to convert from EBNF to BNF but with this conversion we lose the structure of main grammar and resulting trees are for the BNF grammars.

ELL, ELR [4–6, 8–10, 12, 13] and other can process EBNF but they do not deal with ambiguities in grammars.

Factorization for GLL was introduced [17] but it is not full support of EBNF.

In this work we present modified generalized LL parsing algorithm which handles grammars in EBNF without transformations. Changes are very native for GLL nature. Proposed modifications allow to get sufficient parsing performance improvement.

This article is structured as follows. We start from Extended BNF generalized LL algorithm description. Blah-blah

2 EBNF processing

Extended Backus-Naur Form [19] is a syntax of expressing context-free grammars. In addition to the Backus-Naur Form syntax it uses such constructions:

- alternation |
- option [...]
- repetition { ... }
- grouping (...)

This form is usually used in technical documentation. While parser generators widely use Extended CFG form: right parts of productions are regular expressions. It operates under similar constructions so EBNF can be represented in ECFG form. In this article we will use the notation of ECFG.

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

3 Generalized LL Parsing

There are works about parsing ECFG: ELL(k) [?] and ELR(k) [?] parsers; Early-style parsers [?] but none of them work with arbitrary ECFG. Generalized algorithms (GLL and GLR) was purposed to perform syntax analysis of linear input by any context-free grammar. Unlike the GLR, GLL algorithm [15] is rather intuitive and allows to perform better diagnostics. As an output of GLL we get Shared Packed Parse Forest(SPPF) [16] that represents all possible derivations of input string.

Work of the GLL algorithm based on descriptors, it allows to handle all possible derivations. Descriptor is a four-element tuple (L, i, T, S) that can uniquely

define state of parsing process. L is a grammar slot — pointer to position in grammar of the form $(S \rightarrow \alpha \cdot \beta)$, i — position in input, T — already built SPPF root, S — current Graph Structured Stack(GSS) [?] node. We used efficient GSS described in [2]. In initial state we have descriptors that describe start positions in grammar and input, dummy tree node and bottom of GSS. On each step algorithm processes first descriptor in queue and makes actions depending on the grammar and input. If there are any ambiguity algorithm will queue descriptor for all cases to handle them all.

There are table based approach [14] which allows to generate only tables for given grammar instead of full parser code. The idea is similar to one in original article and main function uses same tree construction and stack processing functions. Code can be found in appendix. Note: we do not include the check for first/follow sets in this paper.

3.1 Factorization

In order to improve performance Elizabeth Scott and Adrian Johnstone offered support of factorised grammars in GLL [17]. The idea is to automatically factorize grammars and use them for parser generation.

Main algorithm creates and queues new descriptors depending on current parse state that we get from unqueued descriptor. In case descriptor was already created it does not add it to queue. For this purpose we have a set of **all** created descriptors. Thus reducing a number of possible descriptors decreases the parse time and required memory.

Let us spot on **slots**. Factorization decreases the number of grammar slots. Consider example from the paper [17] on fig. 1.

$$\begin{array}{lcl}
 S ::= a a B c d & & \\
 \quad | a a c d & S ::= a a (B c d \mid c (d \mid e) \mid \varepsilon) & \\
 \quad | a a c e & \text{(b) Production } P_0' & \\
 \quad | a a & & \\
 \text{(a) Production } P_0 & &
 \end{array}$$

Fig. 1. Example of factorization

Production P_0 factorises to P_0' . Second is much compact and contains much less possible slots, so parser creates less descriptors. It gives significant performance improvement on some grammars.

But this idea can be extended to full EBNF support. Let us show how to do it.

4 Extended CFG GLL Parsing

In this section we will show an application of Extended Context-Free Grammars(ECFG) in automaton and corresponding GLL-style parsers.

The idea of factorisation was evolved to use of automaton and their minimization. Right parts of ECFG are regular expressions under alphabet of terminals and nonterminals. There are some basic methods of conversion regular expressions to nondeterministic finite state automaton. Thus for each right-hand side of grammar productions we can build a finite state automaton, with edges tagged with terminals, nonterminals or ε -symbols. Thompson's method [18] can be used for this purpose. In built automaton each nonterminal edge should be complemented with name of initial state of automaton that stands for this nonterminal. An example of constructed automaton for grammar Γ_0 (fig. 2a) is given on fig. 2b. State 0 is start state. We will call final states of automaton "pop" states.

Produced automaton are ε -NFAs under alphabet of nonterminals, terminals and ε symbols. Decrease of the quantity of the automaton states decreases number of GLL descriptors, as it was with factorization. Thus to increase performance of parsing we can minimize the number of states in produced automaton.

First, every automaton should be converted to determinate FA. An algorithm is described in [3]. Then John Hopcroft's algorithm [11] can be applied to all automaton at one time. An algorithm is based on dividing all states on equivalent classes. Initial state of algorithm consist of 2 classes: first contains final states and second contains all other. For our problem we can set an initial state as follow: first class contains all final states of **all** automaton and second class contains all the other. As an algorithm result we get classes which represent new states of automaton. An example for grammar G_0 is shown on fig. 2c.

Now we can define automaton that will be used for parsing.

Definition 1 Automaton A is a tuple $(\Sigma, N, Q, S, A, P, \delta, F)$, where Σ is a set of terminals, N — set of nonterminals, Q — set of states, $S \in Q$ — start state, $A \in N$ — start nonterminal, $P \in Q$ — set of "pop" states, $\delta : Q \times M \rightarrow Q$, $M \in \Sigma \cup N$ — edges of automaton, $F : N \rightarrow Q$ — start states of nonterminals.

Note: this automaton are not an abstract machines, but just a "convenient" representations of grammars.

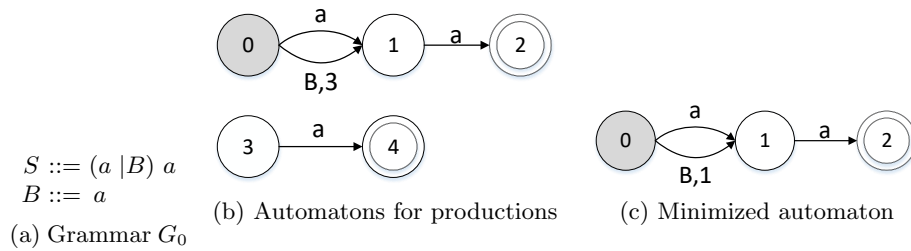


Fig. 2. Example of automaton

4.1 Input processing

Slots have become automaton states. And just as we can move through grammar slots we can move through states of automaton. But in automaton we have nondeterministic choice because there can be many edges to other states. Such significant cases:

- edge that contains current input terminal exists a) it leads to “pop” state
b) next state is not “pop”
- nonterminal edge exists

Both cases can be simultaneously, this brings nondeterminism. For second case we just can call create function for each nonterminal. But for the terminal case we need to add descriptor that describes next position to queue without checking it's existence in descriptor elimination set.

```
function ADD( $S, u, i, w$ )
  if ( $S, u, i, w$ )  $\notin U$  then
     $U.add(S, u, i, w)$ 
     $R.add(S, u, i, w)$ 
```

Function **add** queues descriptor if it was not already created.

```
function CREATE( $edge, u, i, w$ )
  ( $\_, Nonterm(A, S_{call}), S_{next}$ )  $\leftarrow edge$ 
  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( $u, h, N$ )
        if  $y \neq \$$  then
          ( $\_, \_, h$ )  $\leftarrow y$ 
          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 **create** is called when we meet nonterminal on edge. It performs necessary operations with GSS and checks if there are already built SPPF for current input position and nonterminal.

```
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
```

```

     $(y, N) \leftarrow \text{getNodes}(S, v.\text{nonterm}, w, z)$ 
    if  $N \neq \$$  then
        pop( $v, i, N$ )
    if  $y \neq \$$  then
        add( $S, v, i, y$ )

```

Pop function is called when we reach “pop” state. It queues descriptors for all outgoing edges from current GSS node.

```

function PARSE
     $R.\text{add}(\text{StartState}, \text{newGSSnode}(\text{StartNonterminal}, 0), 0, \$)$ 
    while  $R \neq \emptyset$  do
         $(C_S, C_U, C_i, C_N) \leftarrow R.\text{Get}()$ 
         $C_R \leftarrow \$$ 
        if  $(C_N = \$) \& (C_S \text{ is pop state})$  then
             $\text{eps} \leftarrow \text{getNodeT}(\varepsilon, C_i)$ 
             $(\_, N) \leftarrow \text{getNodes}(C_S, C_U.\text{nonterm}, \$, \text{eps})$ 
            pop( $C_U, C_i, N$ )
        for each  $\text{edge}(C_S, \text{symbol}, S_{\text{next}})$  do
            switch  $\text{symbol}$  do
                case  $\text{Terminal}(x)$  where  $(x = \text{input}[i])$ 
                     $C_R \leftarrow \text{getNodeT}(x, C_i)$ 
                     $C_i \leftarrow C_i + 1$ 
                     $(C_N, N) \leftarrow \text{getNodes}(S_{\text{next}}, C_U.\text{nonterm}, C_N, C_R)$ 
                    if  $N \neq \$$  then
                        pop( $C_U, C_i, N$ )
                    if  $C_N \neq \$$  then
                         $R.\text{add}(S_{\text{next}}, C_N, C_i, C_N)$ 
                case  $\text{Nonterminal}(A, S_{\text{call}})$ 
                    create( $\text{edge}, C_U, C_i, C_N$ )

```

The main function **parse** handles queued descriptor and checks all outgoing edges from current state to be appropriate for transition depending on current input terminal, and symbol on edge.

4.2 SPPE construction

First, we should define derivation trees for automaton: it is an ordered tree whose root labeled with start state, leaf nodes are labeled with a terminals from automaton’s edges or ε and interior nodes are labeled with nonterminals from automaton’s edges and have a sequence of children that corresponds to edge labels of path in automaton that starts from the start state of this nonterminal. More formal.

Definition 2 *Derivation tree of sentence α for the automaton $A = (\Sigma, N, Q, S, A, P, \delta, F)$:*

- Ordered rooted tree. Root labeled with A

- *Leafs are terminals* $\in \Sigma$
- *Nodes are nonterminals* $\in N$
- *Node with label* $N_i \in N$ *has children* $l_0 \dots l_n (l_i \in \Sigma \cup N)$ *iff exists path* $F(N_i) \xrightarrow{l_0} \dots \xrightarrow{l_n} q_m, q_m \in P.$

Automaton is ambiguous if there exist string that have more than one derivation trees. Thus, we can define SPPF for automaton. It is similar to SPPF for grammars described in [16]. SPPF contains symbol nodes, packed nodes and intermediate nodes.

Packed nodes are of the form (S, k) , where S is a state of automaton. Symbol nodes have labels (X, i, j) where $X \in \Sigma \cup N \cup \varepsilon$. Intermediate nodes have labels (S, i, j) , where S is a state of automaton. i is position in input before leftmost leaf terminal, j — position after rightmost leaf.

Packed node has one or two children: right child is a symbol node, left child is — symbol or intermediate node. Nonterminal nodes have packed children of the form (S, k) where S is pop state. Terminal symbol nodes are leafs.

Use of intermediate and packed nodes leads to binarization of SPPF and thus the space complexity is $O(n^3)$.

State 1 can be matched with two grammar slots: $S ::= (a \cdot a)|(b \ a)$ and $S ::= (a \ a)|(b \cdot a)$. But SPPF represents WHAT???

function getNodeT(x, i) does not change

We defined function **getNodeP** which can construct two nodes: intermediate and nonterminal (at least one of them, at most both). It uses modified function **getNodeP** that takes additional argument: state or nonterminal name. Symbol in returned SPPF node will be this argument's value.

```

function GETNODES( $S, A, w, z$ )
  if ( $S$  is pop state) then
     $x \leftarrow \text{getNodeP}(S, A, w, z)$ 
  else
     $x \leftarrow \$$ 

  if  $S.outedges = \emptyset$  then
     $y \leftarrow \$$ 
  else
    if ( $w = \$$ ) & not ( $z$  is nonterminal node and it's extents are equal) then
       $y \leftarrow z$ 
    else
       $y \leftarrow \text{getNodeP}(S, S, w, z)$ 
  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

```

```

     $y' \leftarrow \text{new packedNode}(S, k)$ 
     $y'.addLeftChild(w)$ 
     $y'.addRightChild(z)$ 
     $y.addChild(y')$ 
  else
     $y \leftarrow$  find or create SPPF node labelled  $(L, k, i)$ 
    if ( $\nexists$  child of  $y$  labelled  $(S, k)$ ) then
       $y' \leftarrow \text{new packedNode}(S, k)$ 
       $y'.addRightChild(z)$ 
       $y.addChild(y')$ 
  return  $y$ 

```

5 Evaluation

Left factorization vs EBNF

Small demo example (message to Scott)

$$\begin{aligned}
 S &::= A A A A A A \\
 &\quad | A a A A A A \\
 A &::= S A | a A | a
 \end{aligned}$$

Fig. 3. Grammar G_0 .

We have compared our parsers built on factorized grammar and on minimized automata. Grammar G_0 (fig. 3) was used for the tests, it has long “common tail” which is not unified with factorization. FSA built for this grammar presented on fig. 4.

Description of input. Short info about PC.

Note: SPPF construction was disabled while testing.

| Length | Time, seconds | | Descriptors | | GSS Nodes | | GSS Edges | |
|--------|-------------------|-----------|-------------------|-----------|------------------|-----------|-------------------|-----------|
| | factorized | minimized | factorized | minimized | factorized | minimized | factorized | minimized |
| 100 | 0.206 | 0.127 | 52790 | 38530 | 200 | 200 | 42794 | 28534 |
| 200 | 1.909 | 1.54 | 215540 | 157030 | 400 | 400 | 175544 | 117034 |
| 300 | 8.844 | 7.125 | 488290 | 355530 | 600 | 600 | 398294 | 265534 |
| 400 | 25.876 | 21.707 | 871040 | 634030 | 800 | 800 | 711044 | 474034 |
| 500 | 60.617 | 51.245 | 1363790 | 992530 | 1000 | 1000 | 1113794 | 742534 |
| 1000 | 842.779 | 768.853 | 5477540 | 3985030 | 2000 | 2000 | 4477544 | 2985034 |
| | Average gain: 19% | | Average gain: 27% | | Average gain: 0% | | Average gain: 33% | |

Table 1. Experiments results.

Table 1 shows that in general minimized version works 19% faster, uses 27% less descriptors and 33% less GSS edges. Also we use this automaton approach in metagenomic assemblies parsing and it gives visible performance increase. A bit more discussion on evaluation.

Examples of SPPF. May be some nontrivial cases: $s -j$ a^* a^* and so on

6 Conclusion and Future Work

Described algorithm implemented in F# as part of the YaccConstructor project. Source code available here: [1].

Proposed modification can not only increase performance, but also decrease memory usage. It is critical for big input processing. For example, Anastasia Ragozina in her master's thesis [14] shows that GLL can be used for graph parsing. In some areas graphs can be really huge: assemblies in bioinformatics ($10^8 \dots$). Proposed modification can improve performance not only in case of classical parsing, but in graph parsing too. We perform some tests that shows performance increasing in metagenomic analysis, but full integration with graph parsing and formal description is required.

One of way to specify any useful manipulations on derivation tree (or semantic of language) is an attributed grammars [?]. YARD supports it but our algorithm is not. So, attributed grammar and semantic calculation is a future work.

Yet another question is possibility of unification our results with tree languages: our definition of derivation tree for ECFG is quite similar to unranked tree and SPPF is similar to automata for unranked trees [7]. Theory of tree languages seems more mature than theory of general SPPF manipulations.

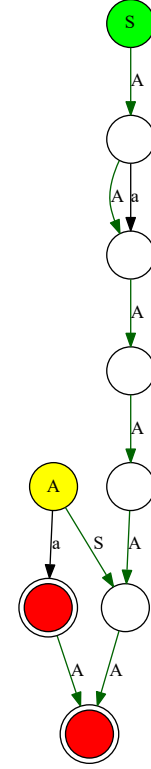


Fig. 4. Minimized automaton for grammar G_0

References

1. Yaccconstructor project repository. <https://github.com/YaccConstructor/YaccConstructor>.
2. A. Afrozeh and A. Izmaylova. Faster, practical gll parsing. In *International Conference on Compiler Construction*, pages 89–108. Springer, 2015.
3. A. V. Aho and J. E. Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.
4. H. Alblas and J. Schaap-Kruseman. An attributed ell (1)-parser generator. In *International Workshop on Compiler Construction*, pages 208–209. Springer, 1990.

5. L. Breveglieri, S. C. Reghizzi, and A. Morzenti. Shift-reduce parsers for transition networks. In *International Conference on Language and Automata Theory and Applications*, pages 222–235. Springer, 2014.
6. A. Bruggemann-Klein and D. Wood. The parsing of extended context-free grammars. 2002.
7. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. 2007.
8. R. Heckmann. An efficient ell (1)-parser generator. *Acta Informatica*, 23(2):127–148, 1986.
9. S. Heilbrunner. On the definition of elr (k) and ell (k) grammars. *Acta Informatica*, 11(2):169–176, 1979.
10. K. Hemerik. Towards a taxonomy for ecfg and rrpq parsing. In *International Conference on Language and Automata Theory and Applications*, pages 410–421. Springer, 2009.
11. J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. Technical report, DTIC Document, 1971.
12. S.-i. Morimoto and M. Sassa. Yet another generation of lalr parsers for regular right part grammars. *Acta informatica*, 37(9):671–697, 2001.
13. P. W. Purdom Jr and C. A. Brown. Parsing extended lr (k) grammars. *Acta Informatica*, 15(2):115–127, 1981.
14. A. Ragozina. Gll-based relaxed parsing of dynamically generated code. Masters thesis, SpBU, 2016.
15. E. Scott and A. Johnstone. Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010.
16. E. Scott and A. Johnstone. Gll parse-tree generation. *Science of Computer Programming*, 78(10):1828–1844, 2013.
17. E. Scott and A. Johnstone. Structuring the gll parsing algorithm for performance. *Science of Computer Programming*, 125:1–22, 2016.
18. K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.
19. N. Wirth. Extended backus-naur form (ebnf). *ISO/IEC*, 14977:2996, 1996.

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
             $y \leftarrow \text{getNodeP}(L, w, z)$ 
            add  $(L, u, h, y)$  where  $h$  is the right extent of  $y$ 
    else
         $v \leftarrow \text{new GSS node labeled } (A, i)$ 
        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
             $y \leftarrow \text{getNodeP}(L, w, z)$ 
            add  $(L, v, i, y)$ 
function GETNODET( $x, i$ )
    if  $(x = \varepsilon)$  then
         $h \leftarrow i$ 
    else
         $h \leftarrow i + 1$ 
     $y \leftarrow \text{find or create SPPF node labelled } (x, i, h)$ 
    return  $y$ 
function GETNODEP( $X ::= \alpha \cdot \beta, w, z$ )
    if ( $\alpha$  is a terminal or a non-nullable nonterminal) &  $(\beta \neq \varepsilon)$  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
             $(-, j, k) \leftarrow w$ 
             $y \leftarrow \text{find or create SPPF node labelled } (L, j, i)$ 
            if ( $\nexists$  child of  $y$  labelled  $(X ::= \alpha \cdot \beta, k)$ ) then
                 $y' \leftarrow \text{new packedNode}(X ::= \alpha \cdot \beta, k)$ 
                 $y'.addLeftChild(w)$ 
                 $y'.addRightChild(z)$ 
                 $y.addChild(y')$ 
            else
                 $y \leftarrow \text{find or create SPPF node labelled } (L, k, i)$ 
                if ( $\nexists$  child of  $y$  labelled  $(X ::= \alpha \cdot \beta, k)$ ) then
                     $y' \leftarrow \text{new packedNode}(X ::= \alpha \cdot \beta, k)$ 

```

```

        y'.addRightChild(z)
        y.addChild(y')
    return y

function DISPATCHER
    if R ≠ ∅ then
        (CL, Cu, Ci, CN) ← R.Get()
        CR ← $
        dispatch ← false
    else
        stop ← true

function PROCESSING
    dispatch ← true
    switch CL do
        case (X → α · xβ) where (x = input[Ci] || x = ε)
            CR ← getNodeT(x, Ci)
            if x ≠ ε then
                Ci ← Ci + 1
            CL ← (X → αx · β)
            CN ← getNodeP(CL, CN, CR)
            dispatch ← false
        case (X → α · Aβ) where A is nonterminal
            create((X → αA · β), Cu, Ci, CN)
        case (X → α ·)
            pop(Cu, Ci, CN)

function PARSE
    while not stop do
        if dispatch then
            dispatcher()
        else
            processing()

```