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 derives parser from grammar. It decreases amount of effort required for syntax analyzer development and maintenance.

Extended Backus-Naur Form [22] 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 much more

compact and human-readable. Because documentation is one of the main sources of information for parsers developers, it would be helpful to have a parser generator which supports grammar in EBNF. Note, that EBNF is a standardized notation for *extended context-free grammars* [10] which can be defined as follows.

Definition 1 *An **extended context-free grammar** (ECFG) [10] 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 (the 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 [9], 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 [2, 3, 5, 8–10, 12, 13] 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" [10]. We only note that most of algorithms are based on classical LL [8, 2, 4] and LR [13, 12, 3] techniques, and they can handle only appropriate subclasses of ECFG. As a result, parsing techniques for ECFG is an actual problem, and 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 is 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 [16],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 [18]. 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 (module some details).

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.

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) [16] 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 rather then GLL-based, which agrees with the observation in [10] 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 elements are defined as fallows:

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

In the initial state there is only one descriptor which consists of start positions in grammar ($L = (S \to \cdot\beta)$) and input ($i = 0$), dummy tree node and the bottom of GSS. Queue is used for descriptors processing control. The algorithm dequeues the first descriptor and acts depending on the grammar and input. If there is an ambiguity, then algorithm queues descriptors for all cases to handle them later.

There is a table based approach [14] 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 appendix A. Note, that we do not include check for first/follow sets in this paper.

2.2 Factorization

In order to improve performance of GLL Elizabeth Scott and Adrian Johnstone offered support of left-factorized grammars in this parsing algorithm [18].

The basic algorithm creates and queues new descriptors depending on current parse state that we get from dequeued descriptor. In case descriptor has been 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.

Grammar factorization decreases the number of grammar slots, which leads to decreasing of total descriptor number. Consider example of grammar and its factorized version the from paper [18] on fig. 1.

$$
\begin{aligned}
S ::= {} & a\ a\ B\ c\ d \\
\mid {} & a\ a\ c\ d \\
\mid {} & a\ a\ c\ e \\
\mid {} & a\ a
\end{aligned}
$$

(a) Production P_0

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

(b) Production $P_0\prime$

Fig. 1. Example of factorization

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

This idea can also be evolved to full ECFG support. Let us show how to do it.

2.3 Recursive automata

In order to use ideas of grammar factorization for handling ECFGs with GLL we propose to use recursive automata (RA) [20] 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 set of terminals, Q — set of states of R, $S \in Q$ — start state, $F \in Q$ — set of final states, $\delta : Q \times (\Sigma \cup Q) \to Q$ — transition function.*

The only difference between RA and FSA is that in RA transition can be labeled either by terminal (Σ) or by state (Q). Further in this paper we will call transitions by elements from Q as nonterminal transitions and by terminal as terminal transitions.

Right parts of ECFG are regular expressions over alphabet of terminals and nonterminals. Thus for each right-hand side of grammar productions we can build a finite state automaton using Thompson's method [21]. To transform the set of produced automata we need to eliminate ε-transitions and replace transitions by nonterminals with transitions labeled by start states of corresponding to nonterminal FSA. An example of constructed recursive automata for grammar Γ_0(fig. 2a) is given on fig. 2b, state 0 is start state.

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.

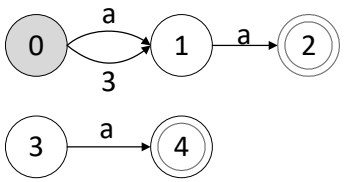
First, RA should be converted to deterministic RA using the algorithm for FSA described in [1]. Then John Hopcroft's algorithm [11] can be applied to RA to minimize the number of states. An example for grammar G_0 is shown on fig. 2c.

2.4 Input processing

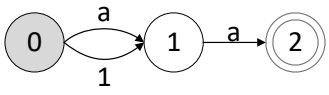
A GLL idea is to traverse through grammar and input simultaneously, creating multiple descriptors for the case of ambiguity.

$S ::= (a\ |B)\ a$
$B ::= a$

(a) Grammar G_0.



(b) Initial RA.



(c) Minimized RA.

Fig. 2. Example of automata

Just as we can move through grammar slots we can move through states of automaton for grammar. We replace grammar slot in descriptor by in constructed RA. The problem is that in automaton we have nondeterministic choice because there can be many transitions to other states. Consider such significant cases:

- there are transition by current input terminal to final state
- there are transition by current input terminal to state that is not final
- there are nonterminal transition

All of them should be handled and this leads to nondeterminism. For the last case we just can call create function for each state. But for the terminal cases we need to add descriptor that describes next position to queue without checking its existence in descriptor elimination set. Thus we use descriptors queue to handle nondeterminism in states, while original algorithm uses it to handle ambiguity in grammars.

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 has not already been created.

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 ← 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 transition. 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) ∉ P) then
        P.add(u, z)
        for all GSS edges (u, S, w, v) do
            (y, N) ← getNodes(S, v.nonterm, w, z)
            if N ≠ $ then
                pop(v, i, N)
            if y ≠ $ then
                add(S, v, i, y)
```

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

```
function PARSE
    R.add(StartState, newGSSnode(StartNonterminal, 0), 0, $)
    while R ≠ ∅ do
        (C_S, C_U, C_i, C_N) ← R.Get()
        C_R ← $
        if (C_N = $)&(C_S is final state) then
            eps ← getNodeT(ε, C_i)
            (_, N) ← 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 ← getNodeT(x, C_i)
                    (y, N) ← getNodes(S_next, C_U.nonterm, C_N, R)
                    if N ≠ $ 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)
```

The main function **parse** handles queued descriptor and checks all transitions from current state to be appropriate for current input terminal, or calls create function when meets nonterminal transitions.

2.5 Parse forest construction

!!!!!! Later, in order to build SPPF, we will need not only start state number, but nonterminal names. For this purpose we define function $\Delta : Q \to N$ where N is nonterminal name.

Result of the parsing process is structural representation of input — a derivation tree, or parse forest for the case of many derivation variants.

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

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 $q_i \xrightarrow{l_0} q_{i+1} \xrightarrow{l_1} \dots \xrightarrow{l_n} q_m$, $q_m \in F$.*

RA is ambiguous if there exists string that has multiple derivation trees. We work with arbitrary grammars, thus our RA can be ambiguous and we can reuse Shared Packed Parse Forest (SPPF) [15] that can represent all possible derivation trees. It is similar to SPPF for grammars described in [17]. 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 $X \in \Sigma \cup \Delta(Q) \cup \{\varepsilon\}$. Intermediate nodes have labels (S, i, j), where S is a state of automaton. Position in the input previous to leftmost leaf terminal is denoted by i, and position succeeding rightmost leaf — j.

Packed node necessarily has right child — symbol node, and optional left child — symbol or intermediate node. Nonterminal and intermediate nodes may have several packed children. Terminal symbol nodes are leaves.

Use of intermediate and packed nodes leads to binarization of SPPF and thus provides better sharing. So in general this representation of parse forest requires less memory [19].

2.6 SPPF construction functions

To handle nondeterminism in states we defined function **getNodes** which checks if the next state of RA is final and for that case constructs nonterminal nodes in addition to intermediate. It uses modified function **getNodeP** that takes additional argument: state or nonterminal name. Symbol in constructed SPPF node becomes the value of the arguments.

```
function getNodes(S, A, w, z)
    if (S is final state) then
        x ← getNodeP(S, A, w, z)
    else
        x ← $

    if (w = $)& not (z is nonterminal node and it's extents are equal) then
        y ← z
    else
        y ← getNodeP(S, S, w, z)
    return (y, x)
function getNodeP(S, L, w, z)
    (_, k, i) ← z
    if (w ≠ $) then
        (_, j, k) ← w
        y ← find or create SPPF node labelled (L, j, i)
        if (∄ child of y labelled (S, k)) then
            y′ ← new packedNode(S, k)
            y′.addLeftChild(w)
            y′.addRightChild(z)
            y.addChild(y′)
    else
        y ← find or create SPPF node labelled (L, k, i)
        if (∄ child of y labelled (S, k)) then
            y′ ← new packedNode(S, k)
            y′.addRightChild(z)
            y.addChild(y′)
    return y
```

function getNodeT(x, i) was not been changed.

3 Evaluation

We have implemented parser generator for proposed algorithm, and in this section we show SPPF example and performance comparison with the parser built on factorized grammar.

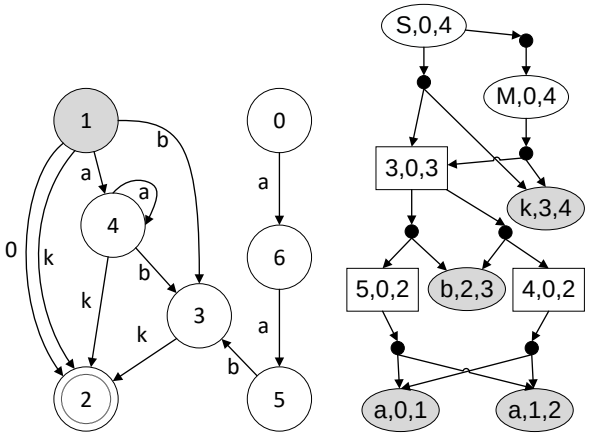
3.1 SPPF example

For the SPPF example we have taken ECFG grammar G_1(fig. 3). It contains constructions(option and repetition) that should be converted with use of extra nonterminals to build regular GLL parser. Our generator constructs recursive automaton R_1(fig. 4a) and parser for it.

For input *aabk* this parser builds SPPF showed on fig. 4b.

$$S ::= (a^* \ b? \ k) \mid M$$
$$M ::= \quad a \ a \ b \ k$$

Fig. 3. Grammar G_1.



(a) RA R_1 for G_1.

(b) SPPF for R_1 and input *aabk*.

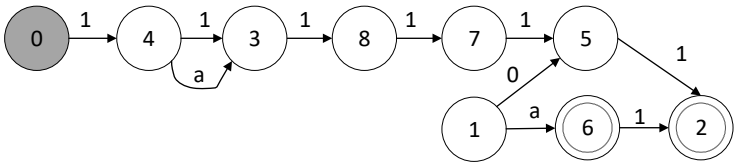
Fig. 4. Example of SPPF.

3.2 Performance measure

We have compared our parsers built on factorized grammars and on minimized recursive automaton. Grammar G_2(fig. 5a) was used for the tests, it has long tails in alternatives which is not unified with factorization. FSA built for this grammar presented on fig. 5b.

$$
\begin{aligned}
S ::= \; & K\ K\ K\ K\ K\ K \\
& | K\ a\ K\ K\ K\ K \\
K ::= \; & S\ K \mid a\ K \mid a
\end{aligned}
$$

(a) Grammar G_2.



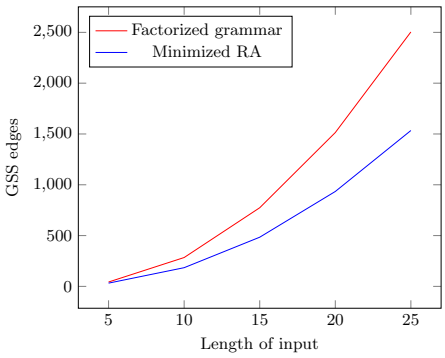
(b) RA for grammar G_2.

Fig. 5. Grammar G_2 and RA for it.

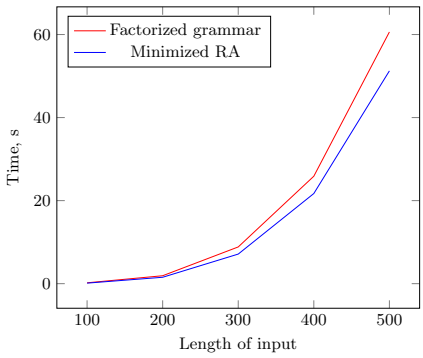
Explanation of slots difference: for BNF, for factorized, for ECFG
Description of input.
Short info about PC.

	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}.

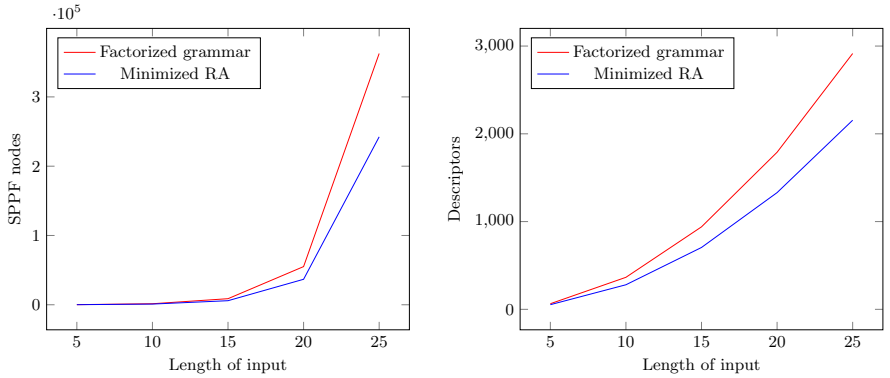


(a) Number of GSS edges.



(b) Time of parsing without SPPF construction.

Fig. 6. Experiments results(1).



(a) Number of SPPF nodes. (b) Number of descriptors.

Fig. 7. Experiments results(2).

Fig. 6 and fig. 7 show experiments results. In general minimized RA version works 33% faster, uses 27% less descriptors, 29% less GSS edges and 33% less SPPF nodes.

We also 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 -¿ a* a* and so on

4 Conclusion and Future Work

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

As we show in evaluation, proposed modification 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: metagenomic assemblies in bioinfomatics, social graphs. We hope that 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, but it is not supported in the algorithm which presented in this article. There is number of works on subclasses of attributed ECFGs (for example [2]), however still no solution for arbitrary ECFGs. Thus, arbitrary attributed ECFGs and semantic calculation support 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 and relations between tree languages and SPPF investigation may get interesting results. For example, relations between tree languages and ECFG discussed here [6] but mainly in context of XML manipulation.

References

1. A. V. Aho and J. E. Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.
2. H. Alblas and J. Schaap-Kruseman. An attributed ell (1)-parser generator. In *International Workshop on Compiler Construction*, pages 208–209. Springer, 1990.
3. 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.
4. A. Brüggemann-Klein and D. Wood. On predictive parsing and extended context-free grammars. In *International Conference on Implementation and Application of Automata*, pages 239–247. Springer, 2002.
5. A. Bruggemann-Klein and D. Wood. The parsing of extended context-free grammars. 2002.
6. A. Brüggemann-Klein and D. Wood. Balanced context-free grammars, hedge grammars and pushdown caterpillar automata. In *Extreme Markup Languages®*, 2004.
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 rrpg 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. J. G. Rekers. *Parser generation for interactive environments*. PhD thesis, Citeseer, 1992.
16. E. Scott and A. Johnstone. Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010.
17. E. Scott and A. Johnstone. Gll parse-tree generation. *Science of Computer Programming*, 78(10):1828–1844, 2013.
18. E. Scott and A. Johnstone. Structuring the gll parsing algorithm for performance. *Science of Computer Programming*, 125:1–22, 2016.

19. E. Scott, A. Johnstone, and R. Economopoulos. Brnglr: a cubic tomita-style glr parsing algorithm. *Acta informatica*, 44(6):427–461, 2007.
20. I. Tellier. Learning recursive automata from positive examples. *Revue des Sciences et Technologies de l'Information-Série RIA: Revue d'Intelligence Artificielle*, 20(6):775–804, 2006.
21. K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.
22. 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**
        $y \leftarrow$ **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 α_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 (α is a terminal or a non-nullable nontermial) & $(\beta \neq \varepsilon)$ **then**

```
        return z
    else
        if (β = ε) then
            L ← X
        else
            L ← (X ::= α · β)
        (_, k, i) ← z
        if (w ≠ $) then
            (_, j, k) ← w
            y ← find or create SPPF node labelled (L, j, i)
            if (∄ child of y labelled (X ::= α · β, k)) then
                y′ ← new packedNode(X ::= α · β, k)
                y′.addLeftChild(w)
                y′.addRightChild(z)
                y.addChild(y′)
        else
            y ← find or create SPPF node labelled (L, k, i)
            if (∄ child of y labelled (X ::= α · β, k)) then
                y′ ← new packedNode(X ::= α · β, k)
                y′.addRightChild(z)
                y.addChild(y′)
        return y
function DISPATCHER
    if R ≠ ∅ then
        (C_L, C_u, C_i, C_N) ← R.Get()
        C_R ← $
        dispatch ← false
    else
        stop ← true
function PROCESSING
    dispatch ← true
    switch C_L do
        case (X → α · xβ) where (x = input[C_i] || x = ε)
            C_R ← getNodeT(x, C_i)
            if x ≠ ε then
                C_i ← C_i + 1
            C_L ← (X → αx · β)
            C_N ← getNodeP(C_L, C_N, C_R)
            dispatch ← false
        case (X → α · Aβ) where A is nonterminal
            create((X → αA · β), C_u, C_i, C_N)
        case (X → α·)
            pop(C_u, C_i, C_N)
function PARSE
```

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