# EBNF in GLL

Artem Gorokhov

St. Petersburg State University, Universitetsky prospekt, 28,
198504 Peterhof, St. Petersburg, Russia
gorohov.art@gmail.com

**Abstract.** Parsing is important step of static programm analysis because it allows to get structural representation of code. Parser generators a 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, EBNF

## 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 appropreate 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 sintax. 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 he 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 if parser generator is based on generalized algorithm. LL family is more intutively 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 [**?**]. This algorithm handles arbitrary context free grammar, even anaumbiguos and (hidden)left-recursve. Worst-case time and space complexity of GLL is cubic in terms of input size. For LL grammars it demonstartes linear time and space complexity.

But BNF required for classical parsing algorithms. It is possible to convert from EBNF to BNF but ....

ELL, ELR [2–7, 9, 10] and other can process EBNF but what about ambiguities in grammars. It is a problem.

Factorization for GLL, 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 structurerd as follows. We start from .... Extended BNF generalized LL algorithm description. Blah-blah

## 2 EBNF processing

EBNF definition

Extended Backus-Naur Form [**?**] is a syntax of expressing context-free grammars. Unlike the Backus-Naur Form it uses such new constructions:

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

It allows to define grammars in more compact way.

Regular expression syntax? Look at "Towards a Taxonomy for ECFG and RRPG Parsing"

In this article we will use the next notation: ....

ELL, ELR etc

## 3 Generalized LL Parsing

Basic GLL algorithm [12] allows to perform syntax analysis of linear input by any context-free grammar. As a result we get Shared Packed Parse Forest(SPPF) [11] that represents all possible derivations of input string.

Work of the GLL algorithm based on descriptors. Descriptor is a four-element tuple that can uniquely define state of parsing process. It consists of:

- **Slot** — position in grammar
- **Position in input** graph
- Already built **tree root**
- Current **GSS node**

and so on about GLL

### 3.1 Factorization

Elizabeth Scott and Adrian Johnstone offered support of factorised grammars in GLL [14]. But our approach yields more increase in performance on some grammars

# 4 Extended BNF GLL Parsing

In this section we will show an application of Extended Backus-Naur Form(EBNF) grammars in automatons and corresponding GLL-style parsers.

GLL allows analysis only by grammars in Backus-Naur Form. When use of EBNF is more common.

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 set of possible descriptors decreases the parse time and required memory.

Let us spot on **slots**. Grammar written in EBNF is usually more compact then it's representation in BNF. That means EBNF contains less slots and parser creates less descriptors. Thus support of EBNF in GLL can increase parsing performance.

## 4.1 Grammar Transformation

There are some basic methods converting regular expressions to nondeterministic finite state automatons. At the same time context-free grammar productions are regular expressions, that can contain as terminals as nonterminals. Thus for each grammar rule we can build a finite state automaton, with edges tagged with terminals, nonterminals or $\varepsilon$-symbols. We used Thompson's method [15]. In built automatons nonterminals should be replaced with links to initial states of automaton that stands for this nonterminal. An example of constructed automaton for grammar $\Gamma_0 1$ is given on fig.

Produced $\varepsilon$-NFAs can be converted to DFAs. An algorithm is described in [1].

Minimization of the quantity of the DFA states decreases number of GLL descriptors. John Hopcroft's algorithm [8] can be used for it. But we can apply it to all automatons 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** automatons and second class contains all the other. As an algorithm result we get classes which represent states of minimised DFA and transitions between them. Initial state is class that contains initial state of automaton that represents productions of start nonterminal.

Some states have labels: names of nonterminals which productions start in that states.

## 4.2 Input processing

Slots becomes DFA states. And just as we can move through grammar slots we can move through states in DFA. But in DFA we have multiple ways to go because many nonterminals can start with current input symbol.

```
function ADD(S, u, i, w)
    if (S, u, i, w) ∉ U then
        U.add(S, u, i, w)
        R.add(S, u, i, w)
function CREATE(edge, u, i, w)
    (_, Nonterm(A, S_call), S_next)) ← edge
    if (∃ GSS node labeled (A, i)) then
        v ← 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) ∈ P) do
                (y, N) ← getNodes(S_next, u.nonterm, w, z)
                if N ≠ $ then
                    (_, _, h) ← N
                    pop(u, h, N)
                if y ≠ $ then
                    (_, _, h) ← 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 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)
```

### 4.3 SPPF construction

First, we should define derivation trees for DFA's: it is an ordered tree whose root is lable of the start state, leaf nodes are labeled with a terminals from DFA's edges or $\varepsilon$ and interior nodes are labeled with nonterminals from DFA's edges($A$) and have a sequence of children that corresponds to edge labels of path in DFA that starts from the state labeled $A$. More formal.

**Definition 1** *Derivation tree of sentence $\alpha$ in the grammar $G = (\Sigma, N, S, P)$:*

- *Ordered rooted tree. Root labeled with $S$*
- *Leafs are terminals $\in \Sigma$*

– *Nodes is nonterminals*
– *Node with label* $N_i$ *has childs* $l_0, \dots l_n$ *if and only if for* $\omega = l_0 \cdot l_1 \dots l_n \in (\Sigma \cup N)^*$ *exists* $p \to M \in P$ *such that* $\omega \in L(M)$

DFA is ambiguous if there exist string that have more than one derivation trees. Thus, we can define SPPF for DFA. It is similar to SPPF for grammars described in [13]. SPPF contains symbol nodes(like derivation trees), packed nodes and intermediate nodes.

Packed nodes are of the form $(S, k)$, where $S$ is a state of DFA. Symbol nodes have labels $(X, i, j)$ where $X$ is an edge symbol or a nonterminal. Intermediate nodes have labels $(S, i, j)$, where $S$ is a state of DFA

A packed node has one or two children right child can be symbol node, left child can be symbol or intermediate node. Nonterminal symbol nodes have packed children nodes 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 $O(n^3)$.

$S ::= (aa)|(ab)$

**Fig. 1.** Grammar $\Gamma_0$

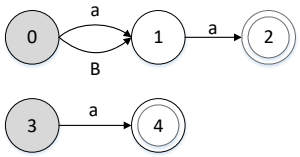


**Fig. 2.** Automaton for $\Gamma_0$

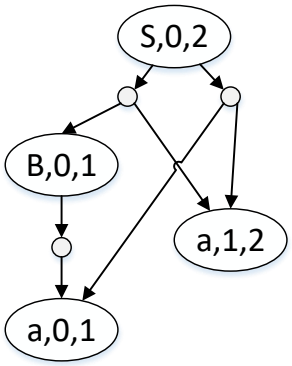


**Fig. 3.** SPPF for input "aa"

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

In states of parsing we can have a nondeterministic choice because the states of DFA can be "pop" states. In this case we need to create nonterminal node and raise **pop** function. But if there exist out edges from this state we also need to create intermediate node. For this purpose we defined function **getNodes** which can construct two nodes: intermediate and nonterminal (at least one of them, at most both of them). So if current state is "pop" state it constructs nonterminal node

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

    if S.outedges = ∅ then
        y ← $
    else
        if (isFiR[S][A]) 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 parse
    R.add(StartState, newGSSnode(StartNonterminal, 0), 0, $)
    while not R ≠ ∅ do
        (C_S, C_u, C_i, C_N) ← R.Get()
        C_R ← $
```

**for each** $edge(C_S, symbol, S_{next})$ **do**
  **switch** $symbol$ **do**
    **case** $Terminal(x)$ where $(x = input[i])$
      $C_R \leftarrow$ **getNodeT**$(x, C_i)$
      $C_i \leftarrow C_i + 1$
      $(C_N, N) \leftarrow$ **getNodes**$(S_{next}, C_u.nonterm, C_N, C_R)$
      **if** $N \neq \$$ **then**
        **pop**$(C_u, C_i, N)$
      **if** $C_N \neq \$$ **then**
        $R.add(S_{next}, C_N, C_i, C_N)$
    **case** $Nonterminal(A, S_{call})$
      **create**$(edge, C_u, C_i, C_N)$

## 5 Evaluation

Left factorization vs EBNF

Small demo example (message to Scott)

```
S ::= A A A A A A | A a A A A A
A ::= S A | a A | a
```

**Fig. 4.** Grammar $G_0$.

We have compared our parsers built on factorized grammar and on minimized FSA. Grammar $G_0$(fig. 4) was used for the tests. FSA built for this grammar presented on fig. 5.

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 FSA approach in metagenomic assemblies parsing and it gives us visible performance increase. Furthermore we can use EBNF constructions without any conversion.

A bit more discussion on evaluation.

## 6 Conclusion and Future Work

Described allgorithm implemented in F# as part of the YaccConstructor project. Source code available here:!!!.

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 [**?**] shows that GLL can be used for graph parsing. In some areas graphs can be really huge: assemblies in bioinfomatics ($10^8$...). 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 semantic is an attributed grammars. YARD support it but our algorithm is not. So, attributed grammar and semantic calculation is a future work.

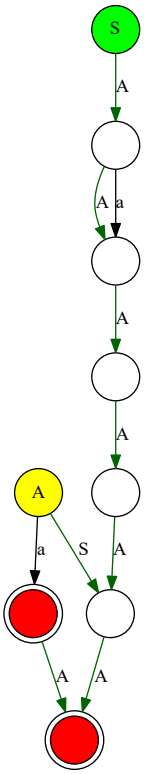


**Fig. 5.** Minimized DFA for grammar $G_0$

## 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. Bruggemann-Klein and D. Wood. The parsing of extended context-free grammars. 2002.
5. R. Heckmann. An efficient ell (1)-parser generator. *Acta Informatica*, 23(2):127–148, 1986.
6. S. Heilbrunner. On the definition of elr (k) and ell (k) grammars. *Acta Informatica*, 11(2):169–176, 1979.
7. 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.
8. J. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, DTIC Document, 1971.

9. S.-i. Morimoto and M. Sassa. Yet another generation of lalr parsers for regular right part grammars. *Acta informatica*, 37(9):671–697, 2001.
10. P. W. Purdom Jr and C. A. Brown. Parsing extended lr (k) grammars. *Acta Informatica*, 15(2):115–127, 1981.
11. J. G. Rekers. *Parser generation for interactive environments*. PhD thesis, Citeseer, 1992.
12. E. Scott and A. Johnstone. Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010.
13. E. Scott and A. Johnstone. Gll parse-tree generation. *Science of Computer Programming*, 78(10):1828–1844, 2013.
14. E. Scott and A. Johnstone. Structuring the gll parsing algorithm for performance. *Science of Computer Programming*, 125:1–22, 2016.
15. K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

## 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 ← i
    else
        h ← i + 1
    y ← find or create SPPF node labelled (x, i, h)
     return y
function getNodeP(X ::= α · β, w, z)
    if (α is a terminal or a non-nullable nontermial) & (β ≠ ε) 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 CONTROL
    while not stop do
        if dispatch then
            dispatcher()
        else
            processing()
```