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, GLL, SPPF, EBNF, ECFG, RRPG, Recursive 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.

There are a wide range of parsing techniques and algorithms (CYK, LR(k), LALR(k), LL, etc) and parser generation tools, which based on it. The most practical parsing algorithms are LL(k)- and LR(k)-based algorithms. The LL family is more intuitive than LR and can provide better error diagnostic. LL(1) is most practical, but not powerful enough, moreover LL(k) for any k is not enough to process some languages: there are LR, but not LL languages. Also left and hidden left recursion in grammars is a problem for LL-based parsers. At the same time there is a common problem for both LL- and LR-based tools: handling of arbitrary ambiguous grammars. All these facts restrict class of grammars which can be handled, which make parser creation difficult. In order to solve these problems generalized LL (GLL) [?] was proposed [?]. 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 and for LL(1) grammars it demonstrates linear time and space complexity.

Extended BNF (EBNF) [?] is a useful format of grammar specification because it allows to make description of language syntax more expressive and compact. This formalism often used in documentation, which is one of main source of information for parsers developers. So, it is necessary to have a parser generator which supports grammar in EBNF. But classical parsing algorithms requires BNF, and as a result, parser generators requires BNF too. It is possible to convert from EBNF to BNF but with this conversion we loose the structure of main grammar and resulting trees are for the BNF grammars.

In order to provide ability to process grammar in ELL, ELR [?,?,?,?,?,?,?,?] and other can process EBNF but they do not deal with ambiguities in grammars.

At the same time, algorithm for left factorized grammars processing was introduced in [?]. Factorization means that there are no two productions for one nonterminal with equal prefixes (look at fig 1 for example). Shown, that factorization can reduce memory usage and increase performance which achieved by reusing common parts of rules for one nonterminal. Purposed idea can be used for processing grammars in EBNF with exception of same effects.

To summarise, it is possible to simplify language description required for parser generation in case a parser generator is based on generalized algorithm which can handle grammars in ECFG. In this work we present modified generalized LL parsing algorithm which handles grammars in ECFG without transformations. We show that changes of basic algorithm are very native for GLL nature. Also we demonstrate that proposed modifications allow to get parsing performance and memory usage improvement.

2 Extended Contex-Free grammars

Parser generators widely use Extended CFG form: right parts of productions are regular expressions over union alphabet $\Sigma \cup N$.

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, $S \in N$ is the start symbol, and P (the productions) is a map from N to regular expressions over alphabet $N \cup \Sigma$.*

It is possible to transform ECFG to CFG [?], but this transformation leads to grammar size increase and change in grammar structure: new nonterminals addition is required during transformation. As a result, parsing performs not in terms of user defined grammar. This fact leads to such problem: parser build structural representation not by the original grammar but by transformed and it may differ from expected. There are algorithms for parsing ECFG without transformations, based on different classical algorithms: ELL(k) and ELR(k) [?] parsers, Early-style parsers [?]. Some of them point out a problem with parsing conflicts [], and none of them work with arbitrary ECFG. Generalized parsing algorithms can handle arbitrary grammars and in this paper we will show how to use them for parsing with arbitrary ECFG.

3 Generalized LL Parsing Algorithm

Generalized parsing algorithms (GLL and GLR) was purposed to perform syntax analysis by arbitrary context-free grammar. Unlike the GLR, GLL algorithm [?] is rather intuitive and allows to perform better syntax error diagnostic. As an output of GLL we get Shared Packed Parse Forest(SPPF) [?] that represents all possible derivations of input string.

Work of the GLL algorithm based on descriptors, it allows to handle all posible 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($S \to \alpha \cdot \beta$), i — position in input, T — already built SPPF root, S — current Graph Structured Stack(GSS) [?] node.

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 [?] 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. Pseudo code can be found in appendix A. Note that we do not include the check for first/follow sets in this paper.

4 Extended CFG GLL Parsing

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

4.1 Factorization

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

The algorithm creates and queues new descriptors depending on current parse state that we get from unqueued 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.

Factorization decreases the number of grammar slots. Consider example from the paper [**?**] 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 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.

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

4.2 Recursive automata

The idea of factorisation was evolved to use of automatons and their minimization.

ECFG can be converted to recursive automata [**?**].

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 ($\in \Sigma$) or by state ($\in Q$). Further in this peper 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 [**?**]. 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 automaton 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 automatons.

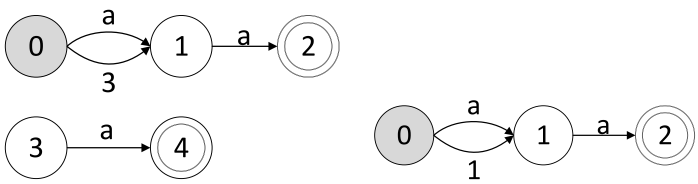
First, RA should be converted to deterministic RA using the algorithm for FSA described in [**?**]. Then John Hopcroft's algorithm [**?**] can be applied to

RA to minimize the number of states. An example for grammar G_0 is shown on fig. 2c.

Note: later we will need a nonterminal names to build a SPPF, for this purpose we define function $\Delta : Q \to N$ where N is nonterminal name.

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

(a) Grammar G_0



(b) Initial RA (c) Minimized RA

Fig. 2. Example of automatons

4.3 Input processing

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

Just as we can move through grammar slots we can move through states of automaton. Grammar slot in descriptor changes to state in 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 it's 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 was not already 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) ∈ P) do
                (y, N) ← getNodes(S_next, u.nonterm, w, z)
                if N ≠ $ then
                    (_, _, h) ← N
                    pop(u, h, N)
                (_, _, 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 **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 pop 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.

4.4 Parse forest construction

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

First, we should define derivation trees for recursive automatons: it is an ordered tree whose root labeled with start state, leaf nodes are labeled with a 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 $\Delta(A)$. More formal.

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

- *Ordered rooted tree. Root labeled with $\Delta(S)$*
- *Leafs are terminals $\in \Sigma$*
- *Nodes are nonterminals $\in \Delta(Q)$*
- *Node with label $N_i \in \Delta(q_i)$ has children $l_0 \dots l_n (l_i \in \Sigma \cup \Delta(Q))$ iff exists path $q_i \xrightarrow{l_0} \dots \xrightarrow{l_n} q_m$, $q_m \in F$.*

RA is ambiguous if there exist string that have more than one derivation trees. We work with arbitrary grammars, thus our RA can be ambiguous and we can define SPPF that can represent all possible derivation trees. It is similar to SPPF for grammars described in [**?**]. 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. i is position in input before leftmost leaf terminal, j — position after rightmost leaf.

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 the space complexity is $O(n^3)$.

function getNodeT(x, i) did not change

We defined function **getNodes** 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 ← 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
```

5 Evaluation(under construction)

Left factorization vs EBNF

Small demo example (message to Scott)

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

Fig. 3. Grammar G_0.

We have compared our parsers built on factorized grammar and on minimized automatons. 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.

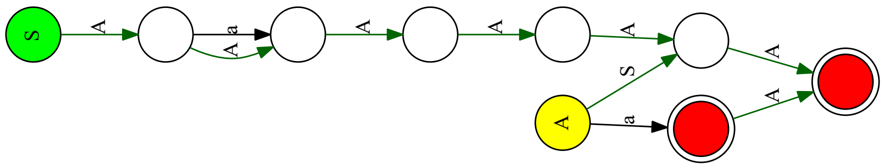


Fig. 4. Minimized automaton for grammar G_0

Explanation of slots difference: for BNF, for factorized, for ECFG
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 -¿ 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: [**?**].

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 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 [?]. Theory of tree languages seems more mature than theory of general SPPF manipulations.

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 ($\beta = \varepsilon$) **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 → 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()
```