EBNF in GLL

Artem Gorokhov

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

Abstract. At least 70 and at most 150 words. *abstract* environment.

Keywords: Parsing, GLL, EBNF

1 Introduction

Static program analysis usually performed over structural representation of code. Parsing

Extended BNF is a useful format of grammar specification.

But BNF required for classical parsing algorithms. ELL, ELR [3, 9, 10, 4, 2, 5–7] and other can process EBNF but what about ambiguities in grammars. It is aproblem.

Generalized LL is good chice for parsing. Let's adopt it!

2 Background — GLL parsing

Main GLL algorithm[11] allows to perform syntax analysis of linear input by any context-free grammar. As a result we get Shared Packed Parse Forest(SPPF) 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 EBNF GLL parsing

In this section we will show an application of EBNF grammars in automatons and corresponding GLL-style parsers.

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

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 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 ε-symbols. We used Thompson's method[14]. 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 ε-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.

5 SPPFs For Automatons

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 ε 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. 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 [12]. 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 Γ_0

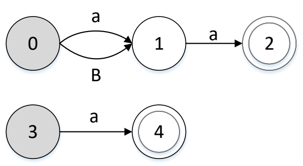

Fig. 2. Automaton for Γ_0

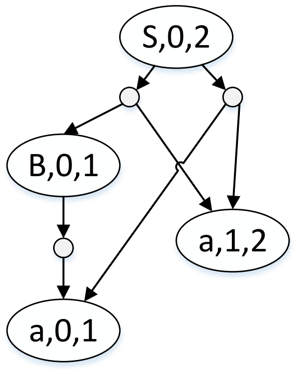

Fig. 3. SPPF for input "aa"

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???

Derivation tree of sentence α 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 $\omega \in L(M)$

6 GLL For Automatons

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.

6.1 Functions Modification

```
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 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 POP($u, i, z$)
    if ($(u, z) \notin \mathcal{P}$) then
        $\mathcal{P}.add(u, z)$
        for all GSS edges $(u, S, w, v)$ do
            $(y, N) \leftarrow$ getNodes$(S, v.nonterm, w, z)$
            if $N \neq \$$ then
                pop$(v, i, N)$
        **if** $y \neq \$$ **then**
            **add**$(S, v, i, y)$
```

6.2 SPPF construction

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
        $x \leftarrow$ getNodeP$(S, A, w, z)$
    else
        $x \leftarrow \$$

    if $S.outedges = \varnothing$ then
        $y \leftarrow \$$
    else
        if ($isFiR[S][A]$) then
            $y \leftarrow z$
        else
            $y \leftarrow$ getNodeP$(S, S, w, z)$
    return $(y, x)$
function getNodeP$(S, L, w, z)$
    $(\_, k, i) \leftarrow z$
    if ($w \neq \$$) then
```

```
        $(\_, j, k) \leftarrow w$
        $y \leftarrow$ find or create SPPF node labelled $(L, j, i)$
        **if** ($\nexists$ child of $y$ labelled $(S, k)$) **then**
            $y\prime \leftarrow$ **new** $packedNode(S, k)$
            $y\prime.addLeftChild(w)$
            $y\prime.addRightChild(z)$
            $y.addChild(y\prime)$
    **else**
        $y \leftarrow$ find or create SPPF node labelled $(L, k, i)$
        **if** ($\nexists$ child of $y$ labelled $(S, k)$) **then**
            $y\prime \leftarrow$ **new** $packedNode(S, k)$
            $y\prime.addRightChild(z)$
            $y.addChild(y\prime)$
    **return** $y$
**function** PARSE
    $R.add(StartState, newGSSnode(StartNonterminal, 0), 0, \$)$
    **while** not $R \neq \varnothing$ **do**
        $(C_S, C_u, C_i, C_N) \leftarrow R.Get()$
        $C_R \leftarrow \$$
        **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**$(edge, C_u, C_i, C_N)$
```

7 Related works

Elizabeth Scott and Adrian Johnstone offered support of factorised grammars in GLL[13]. But our approach yields more increase in performance on some grammars

Moreover there is a modification that allows to use it with regular approximations It was introduced by Anastasia Ragozina in her master's thesis.

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. E. Scott and A. Johnstone. Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010.
12. E. Scott and A. Johnstone. Gll parse-tree generation. *Science of Computer Programming*, 78(10):1828–1844, 2013.
13. E. Scott and A. Johnstone. Structuring the gll parsing algorithm for performance. *Science of Computer Programming*, 125:1–22, 2016.
14. 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
        $h \leftarrow i$
    else
        $h \leftarrow i + 1$
    $y \leftarrow$ find or create SPPF node labelled $(x, i, h)$
    return $y$
function GETNODEP($X ::= \alpha \cdot \beta, w, z$)
    if ($\alpha$ is a terminal or a non-nullable nontermial) & ($\beta \neq \varepsilon$) then
        return $z$
    else
        if ($\beta = \varepsilon$) then
            $L \leftarrow X$
        else
            $L \leftarrow (X ::= \alpha \cdot \beta)$
        $(\_, k, i) \leftarrow z$
        if ($w \neq \$$) then
            $(\_, j, k) \leftarrow w$
            $y \leftarrow$ find or create SPPF node labelled $(L, j, i)$
            if ($\nexists$ child of $y$ labelled $(X ::= \alpha \cdot \beta, k)$) then
                $y\prime \leftarrow$ new $packedNode(X ::= \alpha \cdot \beta, k)$
                $y\prime.addLeftChild(w)$
                $y\prime.addRightChild(z)$
                $y.addChild(y\prime)$
        else
            $y \leftarrow$ find or create SPPF node labelled $(L, k, i)$
            if ($\nexists$ child of $y$ labelled $(X ::= \alpha \cdot \beta, k)$) then
                $y\prime \leftarrow$ new $packedNode(X ::= \alpha \cdot \beta, k)$
                $y\prime.addRightChild(z)$
                $y.addChild(y\prime)$
        return $y$
function DISPATCHER
    if $R \neq \varnothing$ then
        $(C_L, C_u, C_i, C_N) \leftarrow R.Get()$
        $C_R \leftarrow \$$
        $dispatch \leftarrow false$
    else
        $stop \leftarrow true$
function PROCESSING
    $dispatch \leftarrow true$
    switch $C_L$ do
        case $(X \to \alpha \cdot x\beta)$ where $(x = input[C_i] \parallel x = \varepsilon)$
            $C_R \leftarrow$ getNodeT$(x, C_i)$
            if $x \neq \varepsilon$ then
                $C_i \leftarrow C_i + 1$
            $C_L \leftarrow (X \to \alpha x \cdot \beta)$
            $C_N \leftarrow$ getNodeP$(C_L, C_N, C_R)$
            $dispatch \leftarrow false$
        case $(X \to \alpha \cdot A\beta)$ where $A$ is nonterminal
            create$((X \to \alpha A \cdot \beta), C_u, C_i, C_N)$
        case $(X \to \alpha \cdot)$
            pop$(C_u, C_i, C_N)$
function CONTROL
    while not $stop$ do
        if $dispatch$ then
            dispatcher()
        else
            processing()
```