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: String-embedded languages, Parsing, GLL, EBNF

1 Introduction

Static program analysis is a well-known technique ... blabla bla bla blablalba

2 EBNF

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.

3 GLL

Main GLL algorithm[3] 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**

It 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 Base functions

dd

```
function ADD(L, u, i, w)
    if (L, u, i, w) ∉ U then
        U.add(L, u, i, w)
        R.add(L, u, i, w)
function CREATE(L, u, i, w)
    if (∃ GSS node labeled (A, i)) then
        v ← 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) ∈ P) do
                y ← getNodeP(L, w, z)
                add(L, u, h, y) where h is the right extent of y
    else
        v ← 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(A ::= ·α_k, v, i, $)
    return v
function POP(u, i, z)
    if ((u, z) ∉ P) then
        P.add(u, z)
        for all GSS edges (u, L, w, v) do
            y ← getNodeP(L, w, z)
            add(L, v, i, y)
function GETNODET(x, i)
    if (x = ϵ) then
        h ← i
    else
        h ← i + 1
    if (∄ SPPF node labelled (x, i, h)) then
        create SPPF node labelled (x, i, h)
     return SPPF node labelled (x, i, h)
```

```
function getNodeP(X ::= α · , w, z)
    if (α is a terminal or a non-nullable nontermial & β ≠ ϵ) then
        return z
    else
        if (β = ϵ) then
            t ← X
        else
            t ← (X ::= α · β)
        suppose that z has label (q, k, i)
        if (w ≠ $) then
            suppose that w has label (s, j, k)
            if (∄ SPPF node y labelled (t, j, i)) then
                create such node
            if (∄ child of y labelled ((X ::= α · β), k)) then
                create such node with left child w and right child z
        else
            if (∄ SPPF node y labelled (t, k, i)) then
                create such node
            if (∄ child of y labelled ((X ::= α · β), k)) then
                create such node with child z
        return y
```

5 New Functions

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($S_{curr}, S_{call}, S_{next}, u, i, w$)
  if ($\exists$ GSS node labeled (S_{call}, i)) **then**
    $v \leftarrow$ GSS node labeled (S_{call}, 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**
        $y \leftarrow$ **getNodeP**(S_{next}, w, z)
        add(S_{next}, u, h, y) where h is the right extent of y
  else
    $v \leftarrow$ **new** GSS node labeled (S_{call}, 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_{next}, w, v) **do**
      $y \leftarrow$ **getNodeP**(S_{next}, w, z)
      add(S_{next}, v, i, y)

function GETNODET(x, i)
  if $(x = \epsilon)$ **then**
    $h \leftarrow i$
  else
    $h \leftarrow i + 1$
  if ($\nexists$ SPPF node labelled (x, i, h)) **then**
    create SPPF node labelled (x, i, h)
  return SPPF node labelled (x, i, h)

function GETNODEP(S, w, z)
  if (α is a terminal or a non-nullable nontermial & $\beta \neq \epsilon$) **then**
    return z
  else
    if $(\beta = \epsilon)$ **then**
      $t \leftarrow X$

```
else
    t ← (X ::= α · β)
suppose that z has label (q, k, i)
if (w ≠ $) then
    suppose that w has label (s, j, k)
    if (∄ SPPF node y labelled (t, j, i)) then
        create such node
    if (∄ child of y labelled ((X ::= α · β), k)) then
        create such node with left child w and right child z
else
    if (∄ SPPF node y labelled (t, k, i)) then
        create such node
    if (∄ child of y labelled ((X ::= α · β), k)) then
        create such node with child z
return y
```

6 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 epsilon-symbols. We used Thompson's method[5]. In built automatons nonterminals should be replaced with links to initial states of automaton that stands for this nonterminal.

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[2] 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.

7 GLL Modification

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. So we need to prevent creation of descriptors for each nonterminal on out edges. We can

generate tables that tells us what nonterminals can infer strings that starts with current terminal. And add descriptors only for this edges. Moreover we need to create descriptor for edge that marked with current terminal if such exists. Other aspects of GLL remains unchanged.

8 Related works

Elizabeth Scott and Adrian Johnstone offered support of factorised grammars in GLL[4]. 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. J. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, DTIC Document, 1971.
3. E. Scott and A. Johnstone. Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010.
4. E. Scott and A. Johnstone. Structuring the gll parsing algorithm for performance. *Science of Computer Programming*, 125:1–22, 2016.
5. K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.