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 ... blabla bla bla blablalba blabla bla bla blablalba blabla bla bla blablalba blabla bla bla blablalba

2 Background — GLL parsing

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**

and so on about GLL

3 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.

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[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.

5 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.

5.1 Functions Modification

function $\textsc{add}(S, u, i, w)$
  if $(S, u, i, w) \notin U$ **then**
    $U.add(S, u, i, w)$

$R.add(S, u, i, w)$

function CREATE$(edge, u, i, w)$
    $edge$ is $(S_{curr}, Nonterm(A, S_{call}), S_{next}))$
    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**
                $y \leftarrow$ **getNodeP**$(S_{next}, u.nonterm, w, z)$
                $(\_, \_, h) \leftarrow y$
                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**
            $y \leftarrow$ **getNodeP**$(S, v.nonterm, w, z)$
            add(S, v, i, y)

5.2 SPPF construction

function **getNodeT**(x, i) does not change

function GETNODEP(S, A, w, z)
    if $(isFiR[S][A])$ & (S is not pop state) **then**
        return z
    else
        if (S is pop state) **then**
            $L \leftarrow A$
        else
            $L \leftarrow S$
        $(\_, k, i) \leftarrow z$
        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 (∄ child of y labelled (S, k)) then
            y′ ← new packedNode(S, k)
            y′.addRightChild(z)
            y.addChild(y′)
    return y
function PARSING
    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] ‖ x = ε)
                    C_R ← getNodeT(x, C_i)
                    if x ≠ ε then
                        C_i ← C_i + 1
                    C_N ← getNodeP(S_next, C_N, C_R)
                    R.add(S_next, C_u, C_i, C_N)
                case Nonterminal(A, S_call)
                    create(edge, C_u, C_i, C_N)
            if C_S is pop state then
                pop(C_u, C_i, C_N)
```

6 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.

A GLL pseudocode

```
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)
    (X ::= αA · β) ← L
    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(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
    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()
```