GLL – Parsing

31/10/07 February 25, 2008

Elizabeth Scott
Department of Computer Science
Royal Holloway, University of London
Egham, Surrey, United Kingdom
e.scott@rhul.ac.uk

Abstract

RIGLR parsers use a Tomita-style GSS and sets of descriptors to parse any given context free grammar. In this paper we use similar machinery to develop a new general parsing technique, GLL. The technique is recursive descent-like and has the properties that the parse follows closely the structure of the grammar rules. Unlike traditional back-tracking recursive descent parsers, a GLL recogniser runs naturally in worst-case cubic time and can be built even for left recursive grammars.

Keywords: generalised parsing, recursive descent, RIGLR parsing, context free languages

Historically the two most popular forms of deterministic parsing have been the bottom-up stack-based LR parsers and the top-down recursive descent parsers. In their deterministic forms both techniques can only be used with limited classes of grammars, but the class of grammars that admit LR-style parsers is larger than the class that admits recursive descent parsers. Despite this, recursive descent remains popular, particularly because the parser displays a close correspondence to the grammar rules, making it easy to write, easy to insert semantic actions, and straightforward for grammar debugging. Also the size of the parser itself is linear in the size of the grammar, whilst in worst case the size of even an LR(0) parse table can be exponential in the size of the grammar.

General parsers that can be used with any context free grammar have also been developed. Techniques that emerged from the natural language community, for example Earley [Ear70] and CYK [You67] parsers, have been not been embraced within mainstream computer science. Rather, techniques that extend the traditional deterministic approaches are preferred. In particular, Tomita-style GLR parsers [Tom86, NF91, SJ06] are used in ASF+SDF [vdBHKO02] and even Bison has a GLR mode [Bis03]. GLR parsers can be written for any context-free grammar, they have linear time complexity on deterministic grammars and can be made worst case cubic [SJE07].

However, for the reasons mentioned above, there is also interest in general parsers that are based on recursive-descent techniques, for example [Par04, Par96, JAV00, WLM03, JS98, BB95]. These parsers usually employ some form of back-tracking or extended lookahead, and cannot be used with grammars that contain left recursion. If not handled carefully, unlimited back-tracking can cause exponential behaviour, thus this type of parser often offers some kind of limited mode in which certain potential parse directions are aborted before

completion. This approach is dangerous; over general claims are sometimes made for the generality of such tools and it can often be difficult to tell precisely which language is accepted. (See [JS] for further discussion on this.)

Aycock and Horspool [AH99] developed an approach designed to reduce the amount of stack activity in a GLR parser, using a family of automata and a Tomita-style *graph structured stack* (GSS) to manage calls from one automaton to another. This algorithm does not admit grammars with hidden left recursion. However, we have modified the GSS and given an alternative algorithm, RIGLR [SJ05], that allows the approach to be applied to all context free grammars, including those with left recursion.

In this paper we use the same ideas to give a general recursive descent-style algorithm, GLL, that can be applied to all context free grammars.

The full RIGLR algorithm begins by terminalising instances of nonterminals in grammar rules until the grammar contains no self embedding. These terminalised nonterminals then correspond to sub-automata calls in the algorithm. We shall outline a simple version of the RIGLR algorithm, in which all instances of nonterminals generate sub-automata calls, using an example grammar that has both hidden left recursion and ambiguity. This both motivates the GLL approach and allows us to establish the general framework we need. We then consider the standard recursive descent style parse functions for the example grammar and show how to build a corresponding GLL recogniser. We give a general method for constructing a worst-case cubic GLL recogniser for any context free grammar. Finally we use a prototype GLL parser generator to give some experimental results.

1 Simple RIGLR

The RIGLR approach consists of a method for building a particular type of push down automaton, RCA(Γ), from a given grammar Γ, and an algorithm, the RIGLR algorithm, for finding all traversals of RCA(Γ) on a given input string in at worst cubic time. The PDA consists of a family of finite state automata that call each other, and the stack is used to manage these calls. A GSS is used to represent the multiple stack options that can occur when the PDA is non-deterministic and 'descriptors' represent the corresponding multiple process configurations. The important point here is that by allowing loops in the GSS, the RIGLR algorithm can handle left recursive PDA calls. We shall use the same approach to define a GLL algorithm which will essentially use an explicit GSS to manage the parse function call stack in a recursive descent style parser. To explain the roles of the GSS and the descriptors we begin by describing the use of the RIGLR algorithm with a simple recursion call automaton, SRCA(Γ).

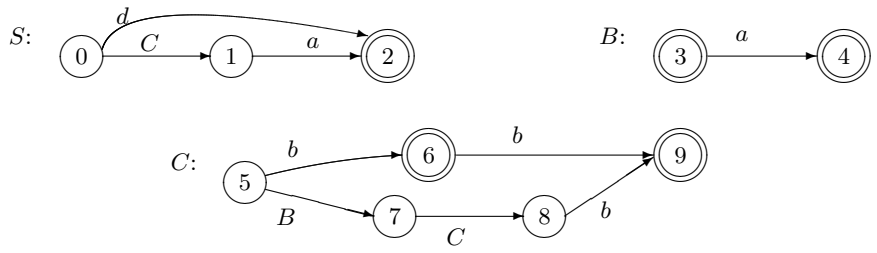
Throughout the paper we illustrate our discussions using the grammar Γ_1

$$\begin{array}{lcl} S & ::= & C\ a \mid d \\ B & ::= & \epsilon \mid a \\ C & ::= & b \mid B\ C\ b \mid b\ b \end{array}$$

1.1 Simple recursion call automata (SRCA)

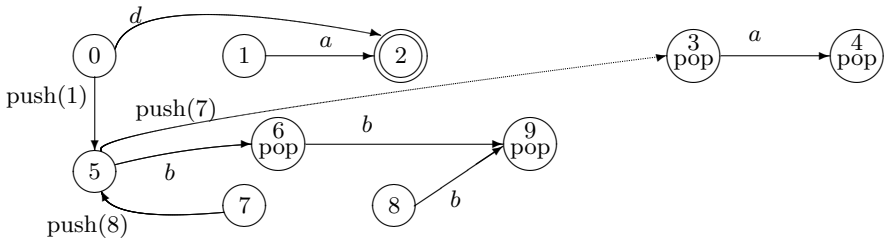
For each nonterminal, X, in the grammar we create D_X, a deterministic automaton (DFA) whose language is precisely the alternates of the grammar rule for X. To reduce the number of states, we merge accepting nodes with no successors.

For Γ_1, the DFAs for S, B and C are (accept states have double lines)



We turn the collection of DFAs into an SRCA by replacing transitions labelled with a nonterminal, X, with actions that push the target state onto the stack and go to the start state of D_X, and allowing pop actions on each accepting state, provided that the stack is nonempty. For start symbol S, the start and accepting states of the SRCA are the start and accepting states of D_S, for grammar start symbol S. (The accepting states of D_S are only pop states if S appears on the right hand side of a grammar rule.)

The following is a graphical representation of SRCA(Γ_1).



1.2 Traversing an SRCA

A *configuration* of an SRCA comprises a current state and a current stack. The initial configuration is the start state and the empty stack. Given an input string $a_1 \dots a_d\$$ (here \$ is the end-of-string symbol) we *traverse* the SRCA by reading each input symbol in turn and moving from one configuration to another according to the SRCA rules.

At the ith step of a traversal, where the current configuration is (h, s), if there is a transition labelled a_i from the state h to the state k then we can move to the configuration (k, s). If there is a transition labelled push(q) from h to k then we can push q onto s, and move to the configuration $(k, (s, q))$. If h is a pop state then we can pop the state, q say, off the top of s, leaving s', and move to the configuration (q, s'). If none of these moves is possible then the parse terminates without success. If h is an accepting state, if the input string has all been read and the stack s is empty then the input is accepted.

In general SRCA(Γ) will be nondeterministic. We can consider all possible traversals by maintaining sets, U_i 09 $\leq i \leq m$, that contain all the configurations that can be obtained on reading $a_1 \ldots a_{i-1}$. The problem is that there can be a large number, even infinitely many, of such configurations. To address this problem we combine all of the stacks that appear in the configurations into a single GSS, merging common initial portions of stacks and recombining stacks of configurations in the same U_i if they have the same stack top.

In detail, the sets U_i contain *descriptors*, pairs (h, u) where h is an SRCA state and u is a GSS node. This descriptor corresponds to all of the configurations whose state is h and whose stack is any of the paths from u to the GSS base node. The bookkeeping required to ensure that all possible traversals are computed is done by maintaining sets $\mathcal{R}_i$ of descriptors not yet processed. A further complication is the possibility that a new stack is created with a top node that has already had a pop action applied. This is accommodated by maintaining sets, $\mathcal{P}_i$, of GSS nodes that have been popped, so that subsequent descriptors can be created if appropriate.

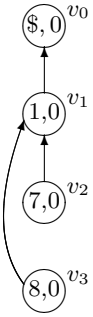
Example SRCA(Γ_1) with the input string *abbba*.

We begin by creating the base GSS node $v_0 = (\$, 0)$ and setting $U_0 = \mathcal{R}_0 = \{(0, v_0)\}$. We remove $(0, v_0)$ from $\mathcal{R}_0$ and apply the action push(1). We create a GSS node, v_1, labelled $(1, 0)$ and make v_0 a child of v_1. We then add the descriptor $(5, v_1)$ to U_0 and $\mathcal{R}_0$, so $U_0 = \{(0, v_0), (5, v_1)\}$ and $\mathcal{R}_0 = \{(5, v_1)\}$.

Next we remove $(5, v_1)$ from $\mathcal{R}_0$ and apply the action push(7), creating a GSS node v_2 labelled $(7, 0)$ with child v_1. We then add the descriptor $(3, v_2)$ to U_0 and $\mathcal{R}_0$.

Removing $(3, v_2)$ from $\mathcal{R}_0$, we apply the action which reads a and create the descriptor $(4, v_2)$ which is added to U_1 and $\mathcal{R}_1$. We also apply the pop action. Since v_2 has label $(7, 0)$ and child v_1 we create the descriptor $(7, v_1)$ which is added to U_0 and $\mathcal{R}_0$. We then add v_2 to $\mathcal{P}_0$.

Removing $(7, v_1)$ from $\mathcal{R}_0$, we apply the action push(8), creating a GSS node v_3 labelled 8 with child v_1 and adding the descriptor $(5, v_3)$ to U_0 and $\mathcal{R}_0$.

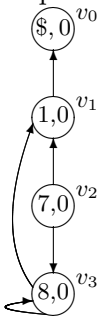


$U_0 = \{(0, v_0),\ (5, v_1),\ (3, v_2),\ (7, v_1),\ (5, v_3)\}$
$\mathcal{R}_0 = \{(5, v_3)\}$
$\mathcal{P}_0 = \{v_2\}$
$U_1 = \{(4, v_2)\}$
$\mathcal{R}_1 = \{(4, v_2)\}$

When we remove $(5, v_3)$ from $\mathcal{R}_0$ and apply the action push(7), we find that there is already a GSS node v_2 labelled $(7, 0)$, thus we re-use this node and add v_3 as a child. The descriptor $(3, v_2)$ is already in U_0 but $v_2 \in \mathcal{P}_0$. Thus we perform the pop action and create the descriptor $(7, v_3)$, which is added to U_0 and $\mathcal{R}_0$.

Finally we remove $(7, v_3)$ from $\mathcal{R}_0$ and apply the action push(8). There is already a GSS node v_3 labelled $(8, 0)$ so we add an edge from this node to itself.

As the descriptor $(5, v_3)$ is already in U_0 and $\mathcal{R}_0$ is empty, this traversal step is now complete.



$$
\begin{aligned}
U_0 &= \{(0, v_0),\ (5, v_1),\ (3, v_2),\ (7, v_1),\ (5, v_3),\ (7, v_3)\} \\
\mathcal{R}_0 &= \emptyset \\
\mathcal{P}_0 &= \{v_2\} \\
U_1 &= \{(4, v_2)\} \\
\mathcal{R}_1 &= \{(4, v_2)\}
\end{aligned}
$$

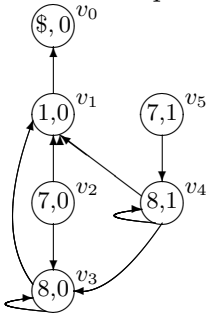
At the start of step 2 we remove $\{(4, v_2)\}$ from $\mathcal{R}_1$ and we pop v_2, creating descriptors $(7, v_3)$ and $(7, v_1)$ which are added to U_1 and $\mathcal{R}_1$. (As v_2 was not created at this step no further children will be added to it so it need not be added to $\mathcal{P}_1$.)

We remove $(7, v_1)$ from $\mathcal{R}_1$ and apply the action push(8). This creates a new GSS node v_4 labelled $(8, 1)$ with child v_1, and the descriptor $(5, v_4)$ which is added to U_1 and $\mathcal{R}_1$. Removing $(7, v_3)$ from $\mathcal{R}_1$ we find that there is already a GSS node v_4 labelled $(8, 1)$ so we make v_3 a child of this node. Since $(5, v_4)$ is already in U_1 nothing is added to $\mathcal{R}_1$.

Next we remove $(5, v_4)$ from $\mathcal{R}_1$ and apply the action push(7), creating a GSS node v_5 with label $(7, 1)$ and child v_4, and add $(3, v_5)$ to U_1 and $\mathcal{R}_1$. We also apply the action read b and add $(6, v_4)$ to U_2 and $\mathcal{R}_2$.

Removing $(3, v_5)$ from $\mathcal{R}_1$, we apply the pop action and add $(7, v_4)$ to U_1 and $\mathcal{R}_1$. Since v_5 was created at this step we add v_5 to $\mathcal{P}_1$.

We remove $(7, v_4)$ from $\mathcal{R}$, apply the action push(8) and add v_4 as a child of itself. Then step 2 is complete.



$$
\begin{aligned}
U_1 &= \{(4, v_2),\ (7, v_1),\ (7, v_3),\ (5, v_4),\ (3, v_5),\ (7, v_4)\} \\
\mathcal{R}_1 &= \emptyset \\
\mathcal{P}_1 &= \{v_5\} \\
U_2 &= \{(6, v_4)\} \\
\mathcal{R}_2 &= \{(6, v_4)\}
\end{aligned}
$$

At the start of step 3 we remove $\{(6, v_4)\}$ from $\mathcal{R}_2$ and we pop v_4, adding $(8, v_4)$, $(8, v_3)$ and $(8, v_1)$ to U_2 and $\mathcal{R}_2$. We also apply the action read b and add $(9, v_4)$ to U_3 and $\mathcal{R}_3$. Removing $(8, v_4)$, $(8, v_3)$ and $(8, v_1)$ from $\mathcal{R}_2$ we apply the action read b and add $(9, v_4)$, $(9, v_3)$ and $(9, v_1)$ to U_3 and $\mathcal{R}_3$. As $\mathcal{R}_2$ is now empty, step 3 is complete.

For step 4 we remove $(9, v_4)$, $(9, v_3)$ and $(9, v_1)$ from $\mathcal{R}_3$ and perform the pop actions, adding $(8, v_4)$, $(8, v_3)$, $(8, v_1)$ and $(1, v_0)$ to U_3 and $\mathcal{R}_3$. When these descriptors are removed from $\mathcal{R}_3$ the elements $(9, v_4)$, $(9, v_3)$, and $(9, v_1)$ are added to U_4 and $\mathcal{R}_4$.

At step 5 we add $(8, v_4)$, $(8, v_3)$, $(8, v_1)$ and $(1, v_0)$ to U_4 and $\mathcal{R}_4$. Then applying the action read a from the descriptor $(1, v_0)$, we add $(2, v_0)$ to U_5

and $\mathcal{R}_5$. At the final step no actions can be carried out, but U_5 contains the descriptor $(2, v_0)$, and 2 is a SRCA accepting state so the string is accepted.

2 The GLL algorithm - an example

We combine the SRCA approach described in the previous section with 'recursive descent' style parsing to get our GLL algorithm, which uses a GSS and sets U_i of descriptors to manage multiple options in the recursive descent functions for non-LL(1) grammars. In this section we give an explicit example using Γ_1, illustrating the relationship between its GLL recogniser and the recursive descent parse functions. A formal construction procedure for a GLL recogniser for a general CFG is given in Section 3.

A traditional recursive descent parser for Γ_1 is composed of parse functions $p_S()$, $p_B()$, $p_C()$ and a main function. The input is held in a global array I.

```
main() { i := 0
         if (I[i] ∈ FIRST(S$))  p_S()  else error()
         if I[i] = $   report success
         else  report failure }

p_S() {  if (I[i] ∈ FIRST(Ca)) {
                    p_C()
                    if (I[i] = a) { i := i + 1 }  else error() }
         else {  if (I[i] = d) { i := i + 1 }  else error() } }

p_B() {  if (I[i] = a) { i := i + 1 } }

p_C() {  if (I[i] = b) { i := i + 1 }
         else {  if (I[i] ∈ FIRST(BCb)) {
                          p_B()
                          if (I[i] ∈ FIRST(Cb)) p_C()  else error()
                          if (I[i] = b) { i := i + 1 }  else error() }
         else {  if (I[i] = b) {
                          i := i + 1
                          if (I[i] = b) { i := i + 1 }  else error() }
         else error() }  }
```

(Here *error*() is a function that terminates the algorithm and reports failure.)

Of course, the example grammar is not LL(1) so this parsing algorithm will not behave correctly. We can convert the function calls into explicit goto statements, labelling the appropriate return lines in the algorithm, and label portions of the code that correspond to different alternates of grammar rules that are not LL(1). We can then use a GSS instead of the standard call stack to manage the algorithm's flow of control.

When the algorithm reaches a goto statement, a descriptor is created containing the label in the goto statement and the current stack-top node in the GSS. In this version of the GLL algorithm descriptors yet to be processed

are held in a single set $\mathcal{R}$, rather than one subset $\mathcal{R}_j$ for each input symbol. However, the algorithm can be modified to use subsets if ordered descriptor processing is required to improve run-time space usage. The outer loop of our algorithm iterates over the set $\mathcal{R}$ until all possible options have been processed.

The algorithm uses the functions *add*(), *create*() and *pop*() that are formally defined in Section 3. Informally, $add(L, u, j)$ checks if there is a descriptor (L, u) in U_j and if not it adds it to U_j and $\mathcal{R}$. The function $create(L, u, j, \mathcal{P})$ checks to see if there is a GSS node $v = (L, j)$ with child u and if not creates one, and then returns v. If $(v, k) \in \mathcal{P}$ then $add(L, u, k)$ is called. The function $pop(u, j, \mathcal{P})$ performs a pop action on the node u and adds (u, j) to $\mathcal{P}$.

We replace a call to *error*() with a jump to the main loop, at L_{S_0}, to process the next descriptor.

There are three types of SPPF nodes: symbol nodes, with labels of the form (x, j, i) where x is a terminal or nonterminal, intermediate nodes, with labels of the form (A, α, j, i) and packed nodes, with labels for the form (r, k). Terminal symbol nodes have no children. A nonterminal symbol node, (A, j, i), has packed node children with labels of the form $(A ::= \gamma, k)$, where $j \leq k \leq i$. A packed node has one or two children, the rightmost is a symbol node (x, k, i), and the left most is a symbol or intermediate node, $(r, j, k-1)$. An intermediate node (t, j, k, i) has two children, the rightmost is a symbol node (x, k, i), and the left most is a symbol or intermediate node, $(r, j, k-1)$.

$findNodeT(a, j, i)$ {
  if there is no SPPF node labelled (a, j, i) create one
  return the SPPF node labelled (a, j, i) }

$findNodeP(r, z, w)$ {
  suppose that r is $X ::= \alpha$ and w has label (x, k, i)
  if $(z \neq \$)$ {
    suppose that z has label $(s, j, k-1)$
    if there does not exist an SPPF node y labelled (X, j, i) create one
    if y does not have a child labelled (r, k)
        create one with left child z and right child w }
  else {
    if there does not exist an SPPF node y labelled (X, k, i) create one
    if y does not have a child labelled (r, k) create one with child w }
  return y }

$findNodeI(x_1 \ldots x_h, z, w)$ {
  suppose that w has label (x, k, i) and z has label $(s, j, k-1)$
  if there does not exist an SPPF node y labelled $(x_1 \ldots x_h, j, i)$ {
    create a node labelled $(x_1 \ldots x_h, j, i)$ with left child z and right child w}
  return y }

read the input into I and set $I[m] := \$$

create GSS nodes $u_1 = (L_{S_0}, 0)$, $u_0 = (\$, 0)$ and an edge $(u_0, \$, u_1)$
create an SPPF node w_0 labelled $(S, 0, m)$
$c_u := u_1$, $i := 0$, $c_L = \$$, $c_R = \$$
for $0 \leq j \leq m$ { $U_j := \emptyset$ }
$\mathcal{R} := \emptyset$, $\mathcal{P} := \emptyset$
if($I[0] \in$ FIRST($S\$$)) { **goto** L_S } **else** { report failure }

L_{S_0}: **if** ($\mathcal{R} \neq \emptyset$) {
remove (L, u, j, w, z) from $\mathcal{R}$
$c_u := u$, $i := j$, $c_L = w$, $c_R = z$
goto L }
else if (there exists an SPFF node labelled $(S, 0, m)$) { report success }
else { report failure }

L_S: **if** ($I[i] \in$ FIRST(Ca)) { **goto** L_{S_1} }
else { **if** ($I[i] = d$) { **goto** L_{S_2} }
L_{S_1}: $c_u := create(R_{C_1}, c_u, i, \$)$, **goto** L_C
R_{C_1}: $c_L := c_R$
if ($I[i] = a$) {
$c_R := findNodeT(a, i, i+1)$, $i := i+1$ }
else { **goto** L_{S_0} }
$c_L := findNodeP(r_1, c_L, c_R)$
$pop(c_u, i, c_L)$, **goto** L_{S_0}
L_{S_2}: $c_R := findNodeT(d, i, i+1)$, $i := i+1$
$c_L := findNodeP(r_1, \$, c_R)$
$pop(c_u, i, c_L)$, **goto** L_{S_0}

L_B: **if** ($I(i) \in$ FOLLOW(B)) { add($L_{B_1}, c_u, i, \$, \$$) }
if ($I(i) \in$ FIRST(b)) { add($L_{B_2}, c_u, i, \$, \$$) }
goto L_{S_0}
L_{B_1}: $c_R := findNodeT(\epsilon, i, i)$
$c_L := findNodeP(r_3, \$, c_R)$
$pop(c_u, i, c_L)$, **goto** L_{S_0}
L_{B_2}: $c_R := findNodeT(a, i, i+1)$, $i := i+1$
$c_L := findNodeP(r_4, \$, c_R)$
$pop(c_u, i, c_L)$, **goto** L_{S_0}

L_C: **if** ($I[i] \in$ FIRST(b)) { $add(L_{C_1}, c_u, i, \$, \$)$ }
if ($I[i] \in$ FIRST(BCb)) { $add(L_{C_2}, c_u, i, \$, \$)$ }
if ($I[i] \in$ FIRST(bb)) { $add(L_{C_3}, c_u, i, \$, \$)$ }
goto L_{S_0}
L_{C_1}: $c_R := findNodeT(b, i, i+1)$, $i := i+1$
$c_L := findNodeP(r_5, \$, c_R)$
$pop(c_u, i, c_L)$, **goto** L_{S_0}
L_{C_2}: $c_u := create(R_{B_1}, c_u, i, \$)$ **goto** L_B
R_{B_1}: $c_L := c_R$
if ($I[i] \in$ FIRST(Cb)) {
$c_u := create(R_{C_2}, c_u, i, c_L)$, **goto** L_C }

```
                  else { goto L_{S_0} }
R_{C_2}:          c_L := findNodeI(BC, c_L, c_R)
                  if (I[i] = b) {
                  c_R := findNodeT(b, i, i + 1)
                  c_L := findNodeP(r_6, c_L, c_R)
                  i := i + 1
                  } else { goto L_{S_0} }
                  pop(c_u, i, c_L), goto L_{S_0}
L_{C_3}:          c_L := findNodeT(b, i, i + 1)
                  i := i + 1
                  if (I[i] = b) {
                    c_R := findNodeT(b, i, i + 1)
                    c_L := findNodeP(r_7, c_L, c_R)
                    i := i + 1 }
                  else { goto L_{S_0} }
                  pop(c_u, i, c_L), goto L_{S_0}
```

Note It is not obvious how to implement the algorithm as written because few programming languages include an unrestricted goto statement that can take a non-statically visible value, which is what is implied in the **if** statement at label L_{S_0} in the above algorithm. However, in an implementation this can be replaced by a Hoare style 'case' statement (a **switch** statement in C), as we shall discuss below.

3 Formal definition of the GLL approach

3.1 Basic definitions

A *context free grammar* (CFG) consists of a set $\mathbf{N}$ of non-terminal symbols, a set $\mathbf{T}$ of terminal symbols, an element $S \in \mathbf{N}$ called the start symbol, and a set of grammar rules of the form $A ::= \alpha$ where $A \in \mathbf{N}$ and α is a (possibly empty) string of terminals and non-terminals. The symbol ϵ denotes the empty string.

We often compose rules with the same left hand sides into a single rule using the alternation symbol, $A ::= \alpha_1 \mid \ldots \mid \alpha_t$. We refer to the strings α_j as the *alternates of A*.

A *derivation step* is an expansion $\gamma A\beta \Rightarrow \gamma\alpha\beta$ where γ and β are strings of terminals and non-terminals and $A ::= \alpha$ is a grammar rule. A *derivation* of τ from σ is a sequence of derivation steps $\sigma \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \tau$, also written $\sigma \stackrel{*}{\Rightarrow} \tau$.

A *sentential form* is any string α such that $S \stackrel{*}{\Rightarrow} \alpha$ and a *sentence* is a sentential form which contains only elements of $\mathbf{T}$.

We say A is *nullable* if $A \stackrel{*}{\Rightarrow} \epsilon$. We define $\textsc{first}_{\mathbf{T}}(A) = \{t \in \mathbf{T} | \exists \alpha (A \stackrel{*}{\Rightarrow} t\alpha)\}$ and $\textsc{follow}_{\mathbf{T}}(A) = \{t \in \mathbf{T} \mid \exists \alpha, \beta (S \stackrel{*}{\Rightarrow} \alpha A t \beta)\}$. If A is nullable we define $\textsc{first}(A) = \textsc{first}_{\mathbf{T}}(A) \cup \{\epsilon\}$ and $\textsc{follow}(A) = \textsc{follow}_{\mathbf{T}}(A) \cup \{\$\}$. Otherwise we define $\textsc{first}(A) = \textsc{first}_{\mathbf{T}}(A)$ and $\textsc{follow}(A) = \textsc{follow}_{\mathbf{T}}(A)$.

We say that a nonterminal A is $LL(1)$ if (i) $A ::= \alpha$, $A ::= \beta$ imply $\textsc{first}(\alpha) \cap \textsc{first}(\beta) = \emptyset$, and (ii) if $A \stackrel{*}{\Rightarrow} \epsilon$ then $\textsc{first}(A) \cap \textsc{follow}(A) = \emptyset$.

3.2 Initial machinery

A GLL recogniser includes labelled lines, and the labels are of three types: return, nonterminal and alternate. Return labels, R_{X_i}, are used to label the main loop of the algorithm and what would be parse function call return lines in a recursive descent parser. Nonterminal labels, L_X, are used to label the first line of what would be the code for the parse function for X in a recursive descent parser. Alternate labels, L_{X_i}, are used to label the first line of what would be the code corresponding to the ith-alternate, α_i say, of X.

The algorithm also employs three functions *add*(), *create*() and *pop*() which build the GSS and create and store processes for subsequent execution, and a function *test*() which checks the current input symbol against the current nonterminal and alternate. These functions are defined as follows.

test(x, A, α) {
  if ($x \in$ FIRST(α)) or ($\epsilon \in$ FIRST(α) and $x \in$ FOLLOW(A)) { return true }
  else { return false } }

add(L, u, j, w, z) {
  if $((L, u, w, z) \notin U_j$ { add (L, u, w, z) to U_j, add (L, u, j, w, z) to $\mathcal{R}$ } }

pop$(u, j, \mathcal{P}, z)$ {
  if $(c_u \neq u_0)$ {
  let (L, k) be the label of u
  add (u, j, z) to $\mathcal{P}$
  for each edge (u, w, v) {
   add(L, v, j, w, z) } } }

create$(L, u, j, \mathcal{P}, w)$ {
  if there is not already a GSS node labelled (L, j) create one
  let v be the GSS node labelled (L, j)
  if there is not an edge from v to u labelled w {
        create an edge from v to u labelled w
        for all $((v, k, z) \in \mathcal{P})$ { *add*(L, u, k, w, z) } }
  return v }

3.3 Dealing with alternates

We begin by defining the part of the algorithm which is generated for an alternate α of a grammar rule for A. We name the corresponding lines of the algorithm *code*$(A ::= \alpha)$.

Each nonterminal instance on the right hand side of the grammar rules is given a unique instance number. We write A_k to indicate the kth instance of nonterminal A. Each alternate of the grammar rule for a nonterminal is also given an instance number. We write $A ::= \alpha_k$ to indicate the kth alternate of the grammar rule for A.

For a terminal a we define

$$\begin{array}{lll} code(a\alpha, j, X) & = & \textbf{if}(I[j] = a)\ \{\ c_R := findNodeT(a, j, j+1), j := j+1\ \} \\ & & \textbf{else}\ \{\ \textbf{goto}\ L_{S_0}\ \} \end{array}$$

For a nonterminal instance A_k we define

$$\begin{array}{lll} code(A_k\alpha, j, X) & = & \textbf{if}(test(I[j], X, A_k\alpha)\ \{ \\ & & \quad c_u := create(R_{A_k}, c_u, j, c_L),\ \textbf{goto}\ L_A\ \} \\ & & \textbf{else}\ \{\ \textbf{goto}\ L_{S_0}\ \} \\ & R_{A_k}: & \end{array}$$

For each production $A ::= \alpha_k$ we define $code(A ::= \alpha_k, j)$ as follows. Let $\alpha_k = x_1 x_2 \ldots x_f$, where each x_l, $1 \leq l \leq f$, is either a terminal or a nonterminal instance of the form X_k.

If $f = 0$ then $\alpha_k = \epsilon$ and

$$code(A ::= \epsilon, j) \ = \ pop(c_u, j),\ \ \textbf{goto}\ L_{S_0}$$

If $f = 1$ and x_1 is a terminal then

$$\begin{array}{lll} code(A ::= x_1, j) & = & c_R := findNodeT(x_1, j, j+1) \\ & & j := j+1 \\ & & c_L := findNodeP(A ::= x_1, \$, c_R) \\ & & pop(c_u, j, c_L),\ \ \textbf{goto}\ L_{S_0} \end{array}$$

If $f \geq 2$ and x_1 is a terminal then

$$\begin{array}{lll} code(A ::= \alpha_k, j) & = & c_L := findNodeT(x_1, j, j+1) \\ & & j := j+1 \\ & & code(x_2 \ldots x_f, j, A) \\ & & c_L := findNodeI(x_1x_2, c_L, c_R) \\ & & code(x_3 \ldots x_f, j, A) \\ & & c_L := findNodeI(x_1x_2x_3, c_L, c_R) \\ & & \ldots \\ & & code(x_{f-1}x_f, j, A) \\ & & c_L := findNodeI(x_1 \ldots x_{f-1}, c_L, c_R) \\ & & code(x_f, j, A) \\ & & c_L := findNodeP(A ::= \alpha_k, c_L, c_R) \\ & & pop(c_u, j, c_L),\ \ \textbf{goto}\ L_{S_0} \end{array}$$

If $f = 1$ and x_1 is a nonterminal instance X_l then

$$\begin{array}{lll} code(A ::= X_l, j) & = & c_u := create(R_{X_l}, c_u, j, \$),\ \textbf{goto}\ L_X \\ & & R_{X_l}: c_L := findNodeP(A ::= x_1, \$, c_R) \\ & & \quad pop(c_u, j, c_L),\ \ \textbf{goto}\ L_{S_0} \end{array}$$

If $f \geq 2$ and x_1 is a nonterminal instance X_l then

$$\begin{array}{lll} code(A ::= \alpha_k, j) & = & c_u := create(R_{X_l}, c_u, j, \$),\ \textbf{goto}\ L_X \\ & & R_{X_l} : c_L := c_R \\ & & \quad code(x_2 \ldots x_f, j, A) \\ & & \quad c_L := findNodeI(x_1x_2, c_L, c_R) \\ & & \quad code(x_3 \ldots x_f, j, A) \\ & & \quad c_L := findNodeI(x_1x_2x_3, c_L, c_R) \\ & & \quad \ldots \\ & & \quad code(x_{f-1}x_f, j, A) \\ & & \quad c_L := findNodeI(x_1 \ldots x_{f-1}, c_L, c_R) \\ & & \quad code(x_f, j, A) \\ & & \quad c_L := findNodeP(A ::= \alpha_k, c_L, c_R) \\ & & \quad pop(c_u, j, c_L),\ \ \textbf{goto}\ L_{S_0} \end{array}$$

3.4 Dealing with rules

Consider the grammar rule $A ::= \alpha_1 \mid \ldots \mid \alpha_t$. We define $code(A, j)$ as follows. If A is an LL(1) nonterminal then

$$\begin{array}{lll} code(A, j) & = & \textbf{if}(test(I[j], A, \alpha_1))\ \{\ \textbf{goto}\ L_{A_1})\ \} \\ & & \ldots \\ & & \textbf{else if}(test(I[j], A, \alpha_t))\ \{\ \textbf{goto}\ L_{A_t}\ \} \\ & & L_{A_1} : code(A ::= \alpha_1, j) \\ & & \ldots \\ & & L_{A_t} : code(A ::= \alpha_t, j) \end{array}$$

If A is not an LL(1) nonterminal then

$$\begin{array}{lll} code(A, j) & = & \textbf{if}(test(I[j], A, \alpha_1))\ \{\ add(L_{A_1}, c_u, j)\ \} \\ & & \ldots \\ & & \textbf{if}(test(I[j], A, \alpha_t))\ \{\ add(L_{A_t}, c_u, j)\ \} \\ & & \textbf{goto}\ L_{S_0} \\ & & L_{A_1} : code(A ::= \alpha_1, j) \\ & & \ldots \\ & & L_{A_t} : code(A ::= \alpha_t, j) \end{array}$$

3.5 Building a GLL recogniser for a general CFG

We suppose that the nonterminals of the grammar Γ are $A, \ldots, X$. Then the GLL recognition algorithm for Γ is given by:

m is a constant integer whose value is the length of the input
I is a constant integer array of size $m + 1$
i is an integer variable
GSS is a labelled digraph whose nodes a labelled with pairs of the form (L, j)
c_u is a GSS node variable
$\mathcal{P}$ is a set of GSS node, integer and SPPF node triples

$\mathcal{R}$ is a set of descriptors

read the input into I and set $I[m] := \$$, $i := 0$
create GSS nodes $u_1 = (L_{S_0}, 0)$, $u_0 = (\$, 0)$ and an edge (u_0, u_1)
$c_u := u_1$, $i := 0$, $c_L := \$$, $c_R := \$$
for $0 \leq j \leq m$ { $U_j := \emptyset$}
$\mathcal{R} := \emptyset$, $\mathcal{P} := \emptyset$
if($I[0] \in$ FIRST($S\$$)) { **goto** L_S } **else** { report failure }
L_{S_0}: **if** $\mathcal{R} \neq \emptyset$ {
remove a descriptor, (L, u, j, w, z) say, from $\mathcal{R}$
$c_u := u$, $i := j$, $c_L := w$, $c_R := z$, **goto** L }
else if (there exists an SPPF node labelled $(S, 0, m)$) { report success }
else { report failure }

L_A: *code*(A, i)
...
L_X: *code*(X, i)

4 Implementation and experimental results

As we mentioned above, to implement a GLL algorithm in a standard programming language the **goto** statement in the main **for** loop can be replaced with a Hoare style case statement. We associate a unique integer, NR_{X_j} or NL_{X_j}, with each label and use that integer in the descriptors (so L becomes an integer variable). For Γ_1 we have

L_{S_0}: **if** $(\mathcal{R} \neq \emptyset)$ { remove an element, (L, u, j) say, from $\mathcal{R}$
$c_u := u$, $i := j$
switch(L) { **case** NR_{C_1}: **goto** R_{C_1}
case NL_{C_1}: **goto** L_{C_1}
case NL_{C_2}: **goto** L_{C_2}
case NR_{C_2}: **goto** R_{C_2}
case NR_{B_1}: **goto** R_{B_1}
case NL_{C_3}: **goto** L_{C_3} } }
else if $((L_{S_0}, u_0, d) \in U_m)$ { report success } **else** { report failure }

Of course, we could substitute the appropriate lines of the algorithm in the case statements if we wished, removing the goto statements completely with the use of break statements.

Elements are only added to $\mathcal{R}$ once so $\mathcal{R}$ can be implemented as a stack or a queue. As written in the algorithm $\mathcal{R}$ is a set so there is no specified order in which its elements are processed. If, however, $\mathcal{R}$ is implemented as a stack then the effect will be a depth-first parse trace, modulo the fact that left recursive calls are terminated at the start of the second iteration. Thus the flow of the algorithm will be essentially that of a recursive descent parser.

On the other hand, $\mathcal{R}$ could be implemented as a set of subsets $\mathcal{R}_j$ which contain the elements of the form (L, u, j). In this case, if the elements of $\mathcal{R}_j$

	Grammar	Input	GSS nodes	GSS edges	$\|U\|$	10^{-3}s
GLL	C	4,291	60,627	219,204	509,484	1,510
SRCA	C	4,291	44,510	78,519	180,114	1,436
GLL	C	36,827	564,164	2,042,019	4,737,207	13,750
SRCA	C	36,827	406,008	739,057	1,717,883	17,330
GLL	Pascal	4,425				
SRCA	Pascal	4,425	21,086	29,369	79,885	1,770
GLL	Γ_1	$a^{20}b^{150}a$	45	67	3,330	10
SRCA	Γ_1	$a^{20}b^{150}a$	44	66	9,514	16
GLL	Γ_2	b^{300}	1,498	671,565	1,123,063	28,595
SRCA	Γ_2	b^{300}	1,496	446,117	896,718	16,550
GLL	Γ_2^*	b^{300}	1,198	357,907	583,654	8,060
SRCA	Γ_2^*	b^{300}	1,796	359,400	899,405	12,930

Table 1:

are processed before any of those in R_{j+1}, $0 \leq j < m$, then the sets U_j and the GSS nodes (u, j) will be constructed in corresponding order, with no elements of U_j created once $\mathcal{R}_j = \emptyset$. This can allow U_j to be deleted once $\mathcal{R}_j = \emptyset$.

To demonstrate practicality we have written GLL-recognisers for grammars for C and Pascal, for the example grammar, Γ_1, used in this paper, and for the grammar, Γ_2,

$$S \ ::= \ b \mid S \ S \mid S \ S \ S$$

on which standard GLR parsers are $O(n^4)$. The GLL-recognisers for C, Γ_1 and Γ_2 were written by hand, demonstrating the relative simplicity of GLL implementation. For C, the GTB tool [JS07] was used to generate the FIRST sets and implementation was made easier by the fact that the grammar is ϵ-free. For Pascal, the recogniser was generated by the newly created GLL-parser generator algorithm that has been added to GTB. We have also built the corresponding SRCA based RIGLR recognisers using GTB.

The input strings for C are a Quine-McCluskey Boolean minimiser, 4,291 tokens, and the source code for GTB itself, 36,827 tokens. The input string for Pascal is a program that performs elementary tree construction and visualisation, 4,425 tokens. (In all cases the input has already been tokenised so no lexical analysis needed to be performed.) The results are shown in Table 1.

We can see that, as well as being easy to write, GLL recognisers perform well. The slower times for Γ_2 arise because the RIGLR algorithm factors the grammar as it builds the SRCA automaton. The results for Γ_2^*

$$S \ ::= \ b \mid S \ S \ A \qquad A \ ::= \ S \mid \epsilon$$

in which the grammar is factored, demonstrate the difference. A GLL recogniser for the equivalent EBNF grammar $S \ ::= \ b \mid S \ S \ (S \mid \epsilon)$ runs in 4.20 CUP seconds on b^{300}, indicating that GLL recogniser performance can be made even better by simple grammar factorisation. This advatange is also displayed by

the Pascal data; the Pacsal BNF grammar used was obtained from the EBNF original and hence is also simply factored. In general, such factorisation can be done automatically and will not change the user's view of the algorithm flow.

5 Conclusions and final remarks

We have shown that GLL recognisers are relatively easy to construct and are also practical. They have the desirable properties of recursive descent parsers in that the parser structure matches the grammar structure. It is easy to extend the GLL algorithm to EBNF grammars, making the resulting parsers even more efficient and giving the advantage that iteration can replace recursion.

The version of the GLL algorithm discussed here is only a recogniser, it does not produce any form of derivation. However, all the derivation paths are explored by the algorithm and it is relatively easy to modify the algorithm to produce Tomita-style SPPF representations of all the derivations of an input string. The modification is essentially the same as that made to turn an RIGLR recogniser into a parser (see [SJ05]).

References

[AH99] John Aycock and Nigel Horspool. Faster generalised LR parsing. In *Compiler Construction, 8th Intnl. Conf, CC'99*, volume 1575 of *Lecture Notes in Computer Science*, pages 32 – 46. Springer-Verlag, 1999.

[BB95] Peter T. Breuer and Jonathan P. Bowen. A PREttier Compiler-Compiler: Generating higher-order parsers in C. *Software Practice and Experience*, 25(11):1263–1297, November 1995.

[Bis03] *Gnu Bison home page*. `http://www.gnu.org/software/bison`, 2003.

[Ear70] J Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[JAV00] *JAVACC home page*. `http://javacc.dev.java.net`, 2000.

[JS] Adrian Johnstone and Elizabeth Scott. Backtracking parsers: *caveat emptor. submitted.*

[JS98] Adrian Johnstone and Elizabeth Scott. Generalised recursive descent parsing and follow determinism. In Kai Koskimies, editor, *Proc. 7th Intnl. Conf. Compiler Construction (CC'98), Lecture Notes in Computer Science 1383*, pages 16–30, Berlin, 1998. Springer.

[JS07] Adrian Johnstone and Elizabeth Scott. Proofs and pedagogy; science and systems: the Grammar Tool Box. *Science of Computer Programming*, 2007.

[NF91] Rahman Nozohoor-Farshi. GLR parsing for ϵ-grammars. In Masaru Tomita, editor, *Generalized LR Parsing*, pages 60–75. Kluwer Academic Publishers, The Netherlands, 1991.

[Par96] Terence John Parr. *Language translation using PCCTS and C++*. Automata Publishing Company, 1996.

[Par04] Terence Parr. *ANTLR home page*. `http://www.antlr.org`, Last visited: Dec 2004.

[SJ05] Elizabeth Scott and Adrian Johnstone. Generalised bottom up parsers with reduced stack activity. *The Computer Journal*, 48(5):565–587, 2005.

[SJ06] Elizabeth Scott and Adrian Johnstone. Right nulled GLR parsers. *ACM Transactions on Programming Languages and Systems*, 28(4):577–618, July 2006.

[SJE07] Elizabeth Scott, Adrian Johnstone, and Giorgios Economopoulos. A cubic Tomita style GLR parsing algorithm. *Acta Informatica*, 44:427–461, 2007.

[Tom86] Masaru Tomita. *Efficient parsing for natural language*. Kluwer Academic Publishers, Boston, 1986.

[vdBHKO02] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[WLM03] Albrecht Wöß, Markus Löberbauer, and Hanspeter Mössenböck. Ll(1) conflict resolution in a recursive descent compiler generator. In G.Goos, J. Hartmanis, and J. van Leeuwen, editors, *Modular languages (Joint Modular Languages Conference 2003)*, volume 2789 of *Lecture Notes in Computer Science*, pages 192–201. Springer–Verlag, 2003.

[You67] D H Younger. Recognition of context-free languages in time n^3. *Inform. Control*, 10(2):189–208, February 1967.