Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2010) 177–189
www.elsevier.com/locate/entcs

GLL Parsing

Elizabeth Scott and Adrian Johnstone[1]

Department of Computer Science
Royal Holloway, University of London
Egham, Surrey, United Kingdom

Abstract

Recursive Descent (RD) parsers are popular because their control flow follows the structure of the grammar and hence they are easy to write and to debug. However, the class of grammars which admit RD parsers is very limited. Backtracking techniques may be used to extend this class, but can have explosive runtimes and cannot deal with grammars with left recursion. Tomita-style RNGLR parsers are fully general but are based on LR techniques and do not have the direct relationship with the grammar that an RD parser has. We develop the fully general GLL parsing technique which is recursive descent-like, and has the property that the parse follows closely the structure of the grammar rules, but uses RNGLR-like machinery to handle non-determinism. The resulting recognisers run in worst-case cubic time and can be built even for left recursive grammars.

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

Parser users tend to separate themselves into bottom-up and top-down tribes. Top-down users value the readability of recursive descent (RD) implementations of LL parsing along with the ease of semantic action incorporation. Bottom-up users value the extended parsing power of LR parsers, in particular the admissibility of left recursive grammars, although LR parsers cannot cope with *hidden* left recursion and even LR(0) parse tables can be exponential in the size of the grammar, while an LL parser is linear in the size of the grammar. Both tribes suffer from the need to coerce their grammars into forms which are deterministic, or at least near-deterministic for their chosen parsing technology. There are many examples of parser generators which extend deterministic algorithms with backtracking and lookahead[10,11,1,18,7,5], although such extensions can trap the unwary. A more formal approach to backtracking is represented by Aho and Ullman's TDPL language (recently repopularised as Parsing Expression Grammars and their associated memoized Packrat parsers). These techniques are superficially attractive because, by definition, there is at most one derivation for each string in the language of a

[1] Email:e.scott@rhul.ac.uk, a.johnstone@rhul.ac.uk

doi:10.1016/j.entcs.2010.08.041

PEG, but, of course, PEG's are not context-free grammars, and as Aho and Ullman said [3, p466]

"...it can be quite difficult to determine what language is defined by a TDPL program."

The current interest in PEG's is another manifestation of users' need for parsers which are human readable.

The Natural Language Processing (NLP) community has always had to cope with the full expressive power of context free grammars. A variety of approaches have been developed and remain popular including CYK [19], Earley [6] and Tomita style GLR parsers [15,9,13]. Although GLR parsing has not been universally adopted by the NLP community — perhaps because of its complexity compared to the easier to visualise CYK and Earley methods — GLR has the attractive property for computer science applications that it achieves linear performance on LR-deterministic grammars whilst gracefully coping with fully general grammars. Since most computing applications involve near-deterministic grammars, GLR has seen significant takeup for language re-engineering applications. It is used, for example, in ASF+SDF [16] and Stratego [17], and even Bison has a partial GLR mode [2]. We have developed [14] cubic worst-case GLR algorithms which smoothly improve their performance to linear time algorithms when processing LR grammars, but this does not address the desiderata of the top down cohort. Nobody could accuse a GLR implementation of a parser for, say, C++, of being easy to read, and by extension easy to debug.

This paper introduces a new algorithm, *Generalised LL* (GLL) parsing, which handles all (including left recursive) context free grammars; runs in worst case cubic time; runs in linear time on LL grammars and which also allows grammar rule factorisation, with consequential speed up. Most importantly, the construction is so straightforward that implementation by hand is feasible: indeed we report on the performance of a hand constructed GLL parser for ANSI C. The resulting code has the RD-property that it is essentially in one-to-one correspondence with the grammar, so parsers may be debugged by stepping through the generated code with a conventional debugger. We believe that GLL will become the generalised parsing algorithm of choice.

The insight behind GLL comes in part from our work on Aycock and Horspool style RIGLR parsers [12]. Aycock and Horspool [4] developed an approach designed to reduce the amount of stack activity in a GLR parser. Their algorithm does not admit grammars with hidden left recursion, but we have given a modified version, the RIGLR algorithm, which is general. In their original paper, Aycock and Horspool described their automata based algorithm as a faster GLR parser but it is our view that the algorithm is in closer in principle to a generalised LL parser. The RIGLR automata are derived from the grammar rules by 'terminalising' certain instances of nonterminals in a way that removes embedded recursion. When an RIGLR traverser encounters a terminalised nonterminal, it is required to make a call to another automaton. Normally, we seek to minimise call stack activity by finding a small set of terminalisations which are complete, in the sense of eliminating all embedded recursion. However, in [12] we noted that

if we 'terminalise' all but the topmost instance of each nonterminal, we get a parser whose stack activity mimics that of a recursive descent parser, except that left recursion is allowable!

It is this observation that lead us to apply the techniques that we developed for RNGLR and RIGLR parsing to give a general recursive descent-style algorithm. In fact we can organise the algorithm so that the parsing schedule either mimics a depth first backtracking recursive descent parser (except that recursive calls are terminated early) or so that all putative parses are synchronised with respect to reading the input. The latter synchronisation is more GLR like and causes the call stacks to be constructed in levels, and that allows a memory efficient approach to the construction of both the stacks and the associated parse trees in a full parser implementation. In this paper we focus on the former organisation.

1 The general approach

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 string in $(\mathbf{T} \cup \mathbf{N})^*$. 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 $\gamma, \beta \in (\mathbf{T} \cup \mathbf{N})^*$ and $A ::= \alpha$ is a grammar rule. A *derivation* of τ from σ is a sequence $\sigma \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \tau$, also written $\sigma \stackrel{*}{\Rightarrow} \tau$ or, if $n > 0$, $\sigma \stackrel{+}{\Rightarrow} \tau$.

A non-terminal A is *left recursive* if there is a string μ such that $A \stackrel{+}{\Rightarrow} A\mu$.

A recursive descent parser consists of a collection of parse functions, $p_A()$, one for each non-terminal A in the grammar. The function selects an alternate, α, of the rule for A, according to the current symbol in the input string being parsed, and then calls the parse functions associated with the symbols in α. It is possible that the current input symbol will not uniquely determine the alternate to be chosen, and if A is left recursive the parse function can go into an infinite loop.

GLR parsers extend LR parsers to deal with non-determinism by spawning parallel processes, each with their own stack. This approach is made practical by combining the stacks into a Tomita-style *graph structured stack* (GSS) which recombines stacks when their associated processes converge. Direct left recursion is not a problem for LR parsers, but hidden left recursion ($A \stackrel{*}{\Rightarrow} \beta A\mu$ where $\beta \stackrel{+}{\Rightarrow} \epsilon$) can result in non-termination.

We have used a modified type of GSS to give a Tomita-style RNGLR algorithm [13] and an Aycock and Horspool-style RIGLR algorithm [12], both of which result in parsers that can be applied to all context free grammars, including those with hidden left recursion. In the RD-based GLL algorithm, introduced in this paper, we shall use RIGLR-style 'descriptors' (see next section) to represent the multiple process configurations which result from non-determinism, and a modified GSS to explicitly manage the parse function call stacks in a way that copes with left recursion.

2 Call stacks and elementary descriptors

We begin by describing the basic approach using the grammar Γ_0

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

(Note that this approach will need modification to become general, as we shall discuss in Section 3.)

A traditional recursive descent parser for Γ_0 is composed of parse functions $p_S()$, $p_A()$, $p_B()$ and a main function. The parse function contains code corresponding to each alternate, α, and these code sections are guarded by a test which checks whether the current input symbol belongs to FIRST(α), or FOLLOW(α) if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, see Section 4.1. We suppose that the input is held in a global array I of length $m+1$, and that $I[m] = \$$, the end-of-string symbol.

main() { $i := 0$
if $(I[i] \in \{a, b, c, d, \$\})$ $p_S()$ **else** $error()$
if $I[i] = \$$ report success **else** $error()$ }

$p_S()$ { **if** $(I[i] \in \{a, c\})$ { $p_A(); p_S();$ **if** $(I[i] = d)$ { $i := i + 1$ } **else** error() }
else { **if** $(I[i] \in \{a, b\})$ { $p_B(); p_S()$ } }

$p_A()$ { **if** $(I[i] = a)$ { $i := i + 1$ }
else if $(I[i] = c)$ { $i := i + 1$ } **else** error() }

$p_B()$ { **if** $(I[i] = a)$ { $i := i + 1$ }
else if $(I[i] = b)$ { $i := i + 1$ } **else** error() }

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

Of course, Γ_0 is not LL(1) so this algorithm will not behave correctly without some additional mechanism for dealing with non-determinism. We address this by converting the function calls into explicit call stack operations using stack *push* and *goto* statements in the usual way. We also partition the body of those functions whose corresponding nonterminal is not LL(1) and separately label each partition. In practice, then, some *goto* statements will have several target labels, corresponding to these multiple partitions: for example, this will be the case for the nonterminal S in Γ_0. We use descriptors to record each possible choice, and replace termination in the RD algorithm with execution re-start from the point recorded in the next descriptor. Instead of calls to the error function, the algorithm simply processes the next descriptor and it terminates when there are no further descriptors to be processed.

In detail, an *elementary descriptor* is a triple (L, s, j) where L is a line label, s is a stack and j is a position in the input array I. We maintain a set $\mathcal{R}$ of current descriptors. At the end of a parse function and at points of non-determinism in

the grammar we create a new descriptor using the label at the top of the current stack. When a particular execution of the algorithm stops, at input $I[i]$ say, the top element L is popped from the stack $s = [s', L]$ and (L, s', i) is added to $\mathcal{R}$ (if it has not already been added). We use $\mathcal{POP}(s, i, \mathcal{R})$ to denote this action. Then the next descriptor (L', t, j) is removed from $\mathcal{R}$ and execution starts at line L' with call stack t and input symbol $I[j]$. The overall execution terminates when the set $\mathcal{R}$ is empty. In order to allow us, later, to combine the stacks we record both the line label L and the current input buffer index k on the stack using the notation L^k. At this interim stage we treat the stack as a bracketed list, [ ] denotes the empty stack, and we assume that we have a function $\mathcal{USH}(s, L^k)$ which simply updates the stack s by pushing on the element L^k. In the final version of the algorithm this will be replaced by a function *create*() which builds the GSS.

$i := 0$; $\mathcal{R} := \emptyset$; $s := [L_0^0]$
L_S: **if** $(I[i] \in \{a, c\})$ add (L_{S_1}, s, i) to $\mathcal{R}$
if $(I[i] \in \{a, b\})$ add (L_{S_2}, s, i) to $\mathcal{R}$
if $(I[i] \in \{d, \$\})$ add (L_{S_3}, s, i) to $\mathcal{R}$
L_0: **if** $(\mathcal{R} \neq \emptyset)$ { remove (L, s_1, j) from $\mathcal{R}$
if $(L = L_0$ and $s_1 = [\,]$ and $j = |I|)$ report success
else { $s := s_1$; $i := j$; **goto** L }
else report failure

L_{S_1}: $\mathcal{USH}(s, L_1^i)$; **goto** L_A
L_1: $\mathcal{USH}(s, L_2^i)$; **goto** L_S
L_2: **if** $(I[i] = d)$ { $i := i + 1$; $\mathcal{POP}(s, i, \mathcal{R})$ }; **goto** L_0
L_{S_2}: $\mathcal{USH}(s, L_3^i)$; **goto** L_B
L_3: $\mathcal{USH}(s, L_4^i)$; **goto** L_S
L_4: $\mathcal{POP}(s, i, \mathcal{R})$; **goto** L_0
L_{S_3}: $\mathcal{POP}(s, i, \mathcal{R})$; **goto** L_0
L_A: **if** $(I[i] = a)$ { $i := i + 1$; $\mathcal{POP}(s, i, \mathcal{R})$; **goto** L_0 }
else{ **if** $(I[i] = c)$ { $i := i + 1$; $\mathcal{POP}(s, i, \mathcal{R})$ }
goto L_0 }
L_B: **if** $(I[i] = a)$ { $i := i + 1$; $\mathcal{POP}(s, i, \mathcal{R})$; **goto** L_0 }
else{ **if** $(I[i] = b)$ { $i := i + 1$; $\mathcal{POP}(s, i, \mathcal{R})$ }
goto L_0 }

As an example we execute the above algorithm with input *aad*\$. We begin by adding $(L_{S_1}, [L_0^0], 0)$ and then $(L_{S_2}, [L_0^0], 0)$ to $\mathcal{R}$ and then go to line L_0. We remove $(L_{S_1}, [L_0^0], 0)$ from $\mathcal{R}$ and go to line L_{S_1}. The push action sets s to $[L_0^0, L_1^0]$ and we go to L_A. The pop action adds $(L_1, [L_0^0], 1)$ to $\mathcal{R}$ and then we go back to L_0. In the same way, processing $(L_{s_2}, [L_0^0], 0)$ from $\mathcal{R}$ eventually results in $(L_3, [L_0^0], 1)$ being added to $\mathcal{R}$.

$$\mathcal{R} = \{(L_1, [L_0^0], 1), (L_3, [L_0^0], 1)\}$$

Next $(L_1, [L_0^0], 1)$ is processed. At L_1 the push action sets s to $[L_0^0, L_2^1]$ and then at L_S we add $(L_{S_1}, [L_0^0, L_2^1], 1)$ and $(L_{S_2}, [L_0^0, L_2^1], 1)$ to $\mathcal{R}$. Similarly, processing

$(L_3, [L_0^0], 1)$ gives

$$\mathcal{R} = \{(L_{S_1}, [L_0^0, L_2^1], 1), (L_{S_2}, [L_0^0, L_2^1], 1), (L_{S_1}, [L_0^0, L_4^1], 1), (L_{S_2}, [L_0^0, L_4^1], 1)\}$$

Processing each of these elements in turn results in

$$\mathcal{R} = \{(L_1, [L_0^0, L_2^1], 2), (L_3, [L_0^0, L_2^1], 2), (L_1, [L_0^0, L_4^1], 2), (L_3, [L_0^0, L_4^1], 2)\}$$

Then, as $I[2] = d$, processing each of these results in

$$\mathcal{R} = \{(L_{S_3}, [L_0^0, L_2^1, L_2^2], 2), (L_{S_3}, [L_0^0, L_2^1, L_4^2], 2), (L_{S_3}, [L_0^0, L_4^1, L_2^2], 2), (L_{S_3}, [L_0^0, L_4^1, L_4^2], 2)\}$$

From this set we get

$$\mathcal{R} = \{(L_2, [L_0^0, L_2^1], 2), (L_4, [L_0^0, L_2^1], 2), (L_2, [L_0^0, L_4^1], 2), (L_4, [L_0^0, L_4^1], 2)\}$$

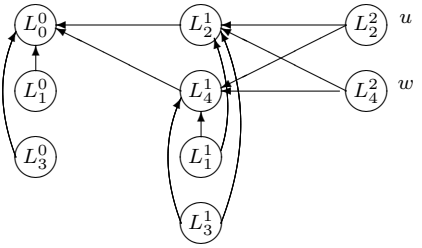
Processing these elements gives

$$\mathcal{R} = \{(L_2, [L_0^0], 3), (L_2, [L_0^0], 2), (L_4, [L_0^0], 3), (L_4, [L_0^0], 2)\}$$

Since $I[3] = \$$, processing these results in $(L_0, [\,], 3)$ and $(L_0, [\,], 2)$ being added to $\mathcal{R}$ and finally algorithm terminates and correctly reports success.

3 The GSS and the sets U_i and $\mathcal{P}$

The problem with the approach as it is described above is that for some grammars the number of descriptors created can be exponential in the size of input and the process does not work correctly for grammars with left recursion. We deal with these issues by combining the stacks into a single, global graph structure, a GSS, recording only the corresponding stack top node in the descriptor, and using loops in the GSS when left recursion is encountered. The GSS will be built by the GLL algorithm as illustrated in the modified Γ_0-recogniser described below. The GSS combining all the stacks constructed in the example for Γ_0 previous section is



A *descriptor* is a triple (L, u, i) where L is a label, u is a GSS node and i is an integer. For example, the four elementary descriptors $\{(L_{S_3}, [L_0^0, L_2^1, L_2^2], 2)$, $(L_{S_3}, [L_0^0, L_2^1, L_4^2], 2)$, $(L_{S_3}, [L_0^0, L_4^1, L_2^2], 2), (L_{S_3}, [L_0^0, L_4^1, L_4^2], 2)\}$ in the above example are replaced by two descriptors $\{(L_{S_3}, u, 2), (L_{S_3}, w, 2)\}$. As a result of this definition actions $_{\mathcal{POP}}(s, i, \mathcal{R})$ are replaced by actions which add (L, v, i) to $\mathcal{R}$ for all children v of node corresponding to the top of s.

In order to avoid creating the same descriptor twice we maintain sets $U_i = \{(L,u) \mid (L,u,i) \text{ has been added to } \mathcal{R}\}$. A problem arises in the case when an additional child, w, is added to u after a pop statement has been executed because the pop action needs to be applied to this child. To address this we use a set $\mathcal{P}$ which contains pairs (u,k) for which a 'pop' line has been executed. When a new child node w is added to u, for all $(u,k) \in \mathcal{P}$ if $(L_u,w) \notin U_k$ then (L_u,u,k) is added to $\mathcal{R}$, where L_u is the label of u.

These techniques are implemented via functions *add*(), *create*() and *pop*() that are formally defined in Section 4. 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 $create(L,u,j)$ creates a GSS node $v = L^j$ with child u if one does not already exist, and then returns v. If $(v,k) \in \mathcal{P}$ then $add(L,u,k)$ is called. The function $pop(u,j)$ calls $add(L_u,v,j)$ for all children v of u, and adds (u,j) to $\mathcal{P}$.

We can rewrite the algorithm from Section 2 as follows. The variable c_u holds the current GSS node, i holds the current input index and $m = |I| + 1$.

create GSS nodes $u_1 := L_0^0$, $u_0 :=$ \$ and an edge (u_0,u_1)
$i := 0$; $\mathcal{R} := \emptyset$; $c_u := u_1$
for $0 \le j \le m$ { $U_j = \emptyset$ }
L_S: **if** $(I[i] \in \{a,c\})$ $add(L_{S_1},c_u,i)$
if $(I[i] \in \{a,b\})$ $add(L_{S_2},c_u,i)$
if $(I[i] \in \{d,\$\})$ $add(L_{S_3},c_u,i)$
L_0: **if** $(\mathcal{R} \neq \emptyset)$ { remove (L,u,j) from $\mathcal{R}$
$c_u := u$; $i := j$; **goto** L }
else if $((L_0,u_0,m) \in U_m)$ report success **else** report failure

L_{S_1}: $c_u := create(L_1,c_u,i)$; **goto** L_A
L_1: $c_u := create(L_2,c_u,i)$; **goto** L_S
L_2: **if**$(I[i] = d)$$\{$ $i := i+1$; $pop(c_u,i)$ $\}$; **goto** L_0
L_{S_2}: $c_u := create(L_3,c_u,i)$; **goto** L_B
L_3: $c_u := create(L_4,c_u,i)$; **goto** L_S
L_4: $pop(c_u,i)$; **goto** L_0
L_{S_3}: $pop(c_u,i)$; **goto** L_0
L_A: **if** $(I[i] = a)$ { $i := i+1$; $pop(c_u,i)$; **goto** L_0 }
else$\{$ **if**$(I[i] = c)$$\{$ $i := i+1$; $pop(c_u,i)$ $\}$; **goto** L_0 }
L_B: **if** $(I[i] = a)$ { $i := i+1$; $pop(c_u,i)$; **goto** L_0 }
else$\{$ **if**$(I[i] = b)$$\{$ $i := i+1$; $pop(c_u,i)$ $\}$; **goto** L_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_0 in the above algorithm. We discuss this in Section 5.

4 Formal definition of the GLL approach

4.1 Initial machinery

We say A is *nullable* if $A \stackrel{*}{\Rightarrow} \epsilon$. We define $\text{FIRST}_{\mathbf{T}}(A) = \{t \in \mathbf{T} | \exists \alpha (A \stackrel{*}{\Rightarrow} t\alpha)\}$ and $\text{FOLLOW}_{\mathbf{T}}(A) = \{t \in \mathbf{T} \mid \exists \alpha, \beta (S \stackrel{*}{\Rightarrow} \alpha A t \beta)\}$. If A is nullable we define $\text{FIRST}(A) = \text{FIRST}_{\mathbf{T}}(A) \cup \{\epsilon\}$ and $\text{FOLLOW}(A) = \text{FOLLOW}_{\mathbf{T}}(A) \cup \{\$\}$. Otherwise we define $\text{FIRST}(A) = \text{FIRST}_{\mathbf{T}}(A)$ and $\text{FOLLOW}(A) = \text{FOLLOW}_{\mathbf{T}}(A)$. We say that a nonterminal A is $LL(1)$ if (i) $A ::= \alpha$, $A ::= \beta$ imply $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$, and (ii) if $A \stackrel{*}{\Rightarrow} \epsilon$ then $\text{FIRST}(A) \cap \text{FOLLOW}(A) = \emptyset$.

We use L_u to denote the line label corresponding to a GSS node u.

A GLL recogniser includes labelled lines 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 i^{th}-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 $(x \in \text{FIRST}(\alpha))$ or $(\epsilon \in \text{FIRST}(\alpha)$ and $x \in \text{FOLLOW}(A))$ { return true }
  else { return false } }

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

pop(u, j) { **if** $(u \neq u_0)$ { add (u, j) to $\mathcal{P}$
  for each child v of u { *add*(L_u, v, j) } } }

create(L, u, j) { **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 {
    create an edge from v to u
    for all $((v, k) \in \mathcal{P})$ { *add*(L, u, k) } }
  return v }

4.2 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 sides of the grammar rules is given an 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

$$code(a\alpha, j, X) \;=\; \textbf{if}(I[j] = a)\ \{\ j := j + 1\ \}\ \ \textbf{else}\ \{\ \textbf{goto}\ L_0\ \}$$

For a nonterminal instance A_k we define

$$\begin{array}{rcl} code(A_k\alpha, j, X) & = & \textbf{if}(test(I[j], X, A_k\alpha)\ \{ \\ & & \quad c_u := create(R_{A_k}, c_u, j),\ \textbf{goto}\ L_A\ \} \\ & & \textbf{else}\ \{\ \textbf{goto}\ L_0\ \} \\ & R_{A_k} : & \end{array}$$

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_p, $1 \leq p \leq f$, is either a terminal or a nonterminal instance of the form X_l.

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

$$code(A ::= \epsilon, j) \;=\; pop(c_u, j),\ \ \textbf{goto}\ L_0$$

If x_1 is a terminal then

$$\begin{array}{rcl} code(A ::= \alpha_k, j) & = & j := j + 1 \\ & & code(x_2 \ldots x_f, j, A) \\ & & code(x_3 \ldots x_f, j, A) \\ & & \ldots \\ & & code(x_f, j, A) \\ & & pop(c_u, j),\ \ \textbf{goto}\ L_0 \end{array}$$

If x_1 is a nonterminal instance X_l then

$$\begin{array}{rcl} code(A ::= \alpha_k, j) & = & c_u := create(R_{X_l}, c_u, j),\ \textbf{goto}\ L_X \\ & R_{X_l} : & code(x_2 \ldots x_f, j, A) \\ & & code(x_3 \ldots x_f, j, A) \\ & & \ldots \\ & & code(x_f, j, A) \\ & & pop(c_u, j),\ \ \textbf{goto}\ L_0 \end{array}$$

4.3 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

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

If A is not an LL(1) nonterminal then

$$\begin{aligned} code(A,j) \quad = \quad & \textbf{if}(test(I[j], A, \alpha_1)) \ \{ \ add(L_{A_1}, c_u, j) \ \} \\ & \ldots \\ & \textbf{if}(test(I[j], A, \alpha_t)) \ \{ \ add(L_{A_t}, c_u, j) \ \} \\ & \textbf{goto} \ L_0 \\ & L_{A_1} : code(A ::= \alpha_1, j) \\ & \ldots \\ & L_{A_t} : code(A ::= \alpha_t, j) \end{aligned}$$

4.4 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 digraph whose nodes are labelled with elements of the form L^j
c_u is a GSS node variable
$\mathcal{P}$ is a set of GSS node and integer pairs
$\mathcal{R}$ is a set of descriptors

read the input into I and set $I[m] := \$$, $i := 0$
create GSS nodes $u_1 = L_0^0$, $u_0 = \$$ and an edge (u_0, u_1)
$c_u := u_1$, $i := 0$
for $0 \leq j \leq m$ { $U_j := \emptyset$}
$\mathcal{R} := \emptyset$, $\mathcal{P} := \emptyset$
if$(I[0] \in \text{FIRST}(S\$))$ { **goto** L_S } **else** { report failure }
L_0: **if** $\mathcal{R} \neq \emptyset$ {
remove a descriptor, (L, u, j) say, from $\mathcal{R}$
$c_u := u$, $i := j$, **goto** L }
else if $((L_0, u_0, m) \in U_m)$ { report success } **else** { report failure }

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

5 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). Of course, we could also 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 the set $\mathcal{R}$ can be implemented efficiently as a stack or as 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, as we have done, $\mathcal{R}$ 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 $\mathcal{R}_j$ are processed before any of those in R_{j+1}, $0 \leq j < m$, then the sets U_j and the GSS nodes 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 grammar, Γ_1,

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

which contains hidden left recursion, 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 [8] 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.

Of the common generalised parsers, the GLL algorithm most closely resembles the Aycock and Horspool style RIGLR algorithm, mentioned above, in which a set of automata which correspond to grammar non-terminals call each other via a

	Grammar	Input	GSS nodes	GSS edges	$\|U\|$	CPU secs
GLL	C	4,291	60,627	219,204	509,484	1.510
SRIGLR	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
SRIGLR	C	36,827	406,008	739,057	1,717,883	17.330
GLL	Pascal	4,425	19,728	26,264	48,827	0.140
SRIGLR	Pascal	4,425	21,086	29,369	79,885	1.770
GLL	Γ_1	$a^{20}b^{150}a$	45	67	3,330	0.010
SRIGLR	Γ_1	$a^{20}b^{150}a$	44	66	9,514	0.016
GLL	Γ_2	b^{300}	1,498	671,565	1,123,063	28.595
SRIGLR	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
SRIGLR	Γ_2^*	b^{300}	1,796	359,400	899,405	12.930

Table 1

common stack. The RIGLR algorithm can be tuned by selecting which non-terminal instances in the grammar generate an automaton call, trading execution time for automaton space. In the most space efficient version, which we call SRIGLR, all non-terminal instances generate a call. We have used GTB to build SRIGLR recognisers which we have compared to the corresponding GLL recognisers.

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). 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 SRIGLR algorithm factors the grammar as it builds the 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 CPU seconds on b^{300}, indicating that GLL recogniser performances can be made even better by simple grammar factorisation. This advantage 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.

6 Conclusions and Final Remarks

We have shown that GLL recognisers are relatively easy to construct and are also practical. They have the desirable property of recursive descent parsers in that the parser structure matches the grammar structure. It is also possible to extend the GLL algorithm to EBNF grammars, allowing factorisation, and the use of iteration in place of recursion, to make the resulting parsers even more efficient.

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, as described in [12].

References

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

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

[3] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1 — Parsing of *Series in Automatic Computation*. Prentice-Hall, 1972.

[4] 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.

[5] 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.

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

[7] 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.

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

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

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

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

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

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

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

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

[16] 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.

[17] Eelco Visser. Program transformation with Stratego/XT: rules, strategies, tools, and systems in StrategoXT-0.9. In C.Lengauer et. al, editor, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, Berlin, June 2004.

[18] 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.

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