GLL parse-tree generation

Elizabeth Scott and Adrian Johnstone

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

Abstract

Backtracking techniques which are often used to extend recursive descent (RD) parsers can have explosive run-times and cannot deal with grammars with left recursion. GLL parsers are fully general, worst-case cubic parsers which have the recursive descent-like property that they are easy to write and to use for grammar debugging. They have the direct relationship with the grammar that an RD parser has. In this paper we give an algorithm for generating GLL parsers which build an SPPF representation of the derivations of the input, complementing our existing GLL recognition algorithm, and we show that such parsers and recognisers are worst-case cubic.

Key words: generalised parsing, recursive descent, GLL parsing, RNGLR and RIGLR parsing, context free languages

In this paper we present the GLL (Generalised LL) parsing algorithm which handles any context free grammar, including those with ambiguity, and which we shall prove has worst-case cubic performance. The algorithm has the 'recursively decent' property: parsers may be constructed straightforwardly by hand directly from the grammar, and the resulting parsers may easily be traced using a conventional language debugger because they have a straightforward relationship with the grammar itself.

This simple mapping between grammar rules and parser fragments allows programmers to think operationally about grammars whilst retaining the declarative nature of the grammar specification. In this respect GLL parsers are significantly different to approaches based on backtracking and lookahead extensions of deterministic parsers [13,14,1,24,9,5] including algorithms such as

Email address: `e.scott@rhul.ac.uk, a.johnstone@rhul.ac.uk` (Elizabeth Scott and Adrian Johnstone).

Aho and Ullmann's TDPL and GTDPL [3] (more recently popularised as Parsing Expression Grammars [8] and their associated Packrat parsers [7]).

The problem with TDPL and greedy backtracking strategies generally is that although the rules from which they are constructed look like context free grammar rules, the parsers may not fully explore the derivation space of the grammar, and so the language accepted may not be the same as the language specified by the typographically identical context free grammar. Software engineers may get nasty surprises when a perfectly sensible looking input is rejected by such a parser.

For users whose grammars are near-deterministic, recursive descent (RD) style parsing continues to be appealing. This is in large part due to the fact that an RD parser is essentially a procedural version of the grammar itself, and hence, in the language design phase, semantic action insertion is easy and grammar debugging is more tractable. In addition, for LL(1) grammars, RD parsers are linear in the size both of the grammar and the input string, while even LR(0) parse tables can be exponential in the size of the grammar [21]. However, few 'real' grammars are LL(1) and devotees of the alternative bottom-up stack-based table driven approach, pioneered by Knuth in his seminal 1960's paper [11], often cite the restrictive nature of the LL(1) conditions, particularly the need to avoid left recursion, as justification for their preference. In practice, grammars for real languages such as Java and C are also not LR(1), and no grammar with *hidden* left recursion (see Section 1) is LR(1).

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 [25], Earley [6] and Tomita style GLR parsers [20,12,16]. 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 been taken up for language re-engineering applications. It is used, for example, in ASF+SDF [22] and Stratego [23], and even Bison has a partial GLR mode [2]. We have developed [16,19] more efficient variants of GLR-parsing including a worst-case cubic time algorithm, BRNGLR, based on binarised derivation forests. However, at their heart all GLR parsers have a shift-reduce automaton (usually table based) and considerable book keeping to handle the sequentialisation of the naturally parallel partial parses. As a result, nobody could accuse a GLR parser for, say, C++, of being easy to read, and by extension easy to use for grammar debugging.

In our preliminary conference paper [17] we introduced a new algorithm, *Generalised LL* (GLL) recognition, which handles all (including left recursive) context free grammars and runs in worst-case cubic time. In that paper we also reported on experimental results for GLL recognisers (but not parsers) for hard highly ambiguous grammars, as well as for the standard grammars for ANSI-C and Pascal. The construction is so straightforward that implementation by hand is feasible: indeed we produced a hand constructed GLL recogniser for ANSI C.

In this paper we extend the GLL algorithm to a full parser and prove that the parsers have worst-case cubic complexity for any context free grammar. We show that run times are commensurate with our BRNGLR algorithm, and entirely practical for traditional parser applications such as language front ends.

1 Background

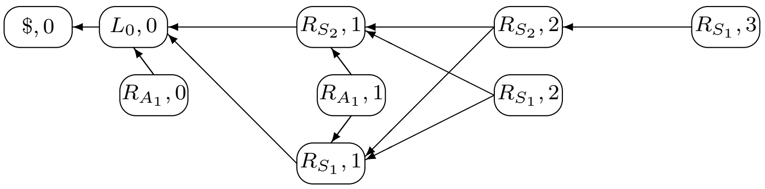
A *context free grammar* (CFG) consists of a set $\mathbf{N}$ of nonterminal symbols, a set $\mathbf{T}$ of terminal symbols, an element $S \in \mathbf{N}$ called the start symbol, and a set of grammar rules of the form $A ::= \alpha$ where $A \in \mathbf{N}$ and α is a string in $(\mathbf{T} \cup \mathbf{N})^*$. 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, $A ::= \alpha_1 \mid \ldots \mid \alpha_p$. 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 $\sigma \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \beta_n = \tau$, also written $\sigma \overset{*}{\Rightarrow} \tau$ or, if $n > 0$, $\sigma \overset{+}{\Rightarrow} \tau$.

A nonterminal A is *left recursive* if there is a string $\mu \in (\mathbf{T} \cup \mathbf{N})^*$ such that $A \overset{+}{\Rightarrow} A\mu$, and *nullable* if $A \overset{*}{\Rightarrow} \epsilon$. If $A \overset{*}{\Rightarrow} \beta A \mu$ where $\beta \overset{+}{\Rightarrow} \epsilon$ we say A has *hidden* left recursion.

An LL(1) recursive descent parser consists of a collection of parse functions, one for each nonterminal 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.

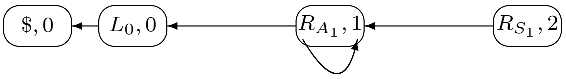
Traditional LR-table based parsers [11,16] use a stack to record the traversal of a finite state automaton rendition of the grammar. GLR parsers extend LR-parsers to deal with non-determinism by spawning parallel processes, each with their own stack. This approach is made practical by combining the stacks into a Tomita-style *graph structured stack* (GSS) which shares common initial substacks and recombines stacks when their associated processes converge. A GSS is simply a directed graph whose nodes are grouped into 'levels' by the value of the input pointer position at the time the node was created. We write (L, i) to indicate a node labelled L at level i. It is conventional to have an arc from an element on the stack to the element below it, and for the bottom element, \$, of the stack to be drawn on the left. For example, the GSS (which relates to the example in Section 2.3)



has four levels and represents the stacks

$$[\$, L_0, R_{A_1}] \quad [\$, L_0, R_{S_2}, R_{A_1}] \quad [\$, L_0, R_{S_1}, R_{A_1}] \quad [\$, L_0, R_{S_2}, R_{S_1}]$$
$$[\$, L_0, R_{S_1}, R_{S_1}] \quad [\$, L_0, R_{S_2}, R_{S_2}, R_{S_1}] \quad [\$, L_0, R_{S_1}, R_{S_2}, R_{S_1}]$$

Direct left recursion is not a problem for LR parsers, but hidden left recursion can result in non-termination essentially because infinitely many stacks are constructed. Our Tomita-style RNGLR algorithm [16] and our Aycock and Horspool-style RIGLR algorithm [15] both use a modified type of GSS which can include cycles, allowing application to all context free grammars. For example, the GSS below represents the infinitely many stacks $[\$, L_0, R_{A_1}, \ldots, R_{A_1}, R_{S_1}]$



In the RD-based GLL algorithm presented in this paper we use RIGLR-style 'descriptors' (see next section) to represent the multiple process configurations which result from non-determinism, and a GSS to explicitly manage the parse function call stacks in a way that copes with left recursion.

2 Generalising recursive descent

In [17] we gave a presentation of the GLL recognition algorithm, which follows the recursive descent approach but is applicable to all grammars. The insight behind GLL comes in part from our work on Aycock and Horspool style RIGLR parsers [15]. 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 closer in principle to a generalised LL parser. It is this observation that led us to apply the techniques that we developed for RNGLR and RIGLR parsing to give a general recursive descent-style algorithm. We refer the reader to [17] for further motivational discussion, and recommend this paper and our more implementation oriented discussion [10] to readers who are not already familiar with generalised parsing. In this section we describe the GLL recogniser informally in terms of recursive descent. We shall use the terminology from [17] so that the reader can refer to this if they wish. Some of the functions introduced here will be modified in the later formal description to incorporate derivation tree construction, which was not discussed in [17].

2.1 Algorithm control flow

As we have said, an RD parser consists of a collection of parse functions associated with the grammar nonterminals. There is a section in the parse function $parseX()$ for each alternate $x_1 \dots x_k$ of the grammar rule for X, and there is a line in the section for each symbol x_i in the alternate. If x_i is a terminal it is matched to the current input symbol and if it is a nonterminal then the corresponding parse function is called.

For example, consider the grammar

$$S \quad ::= \quad a\ S \mid A\ S\ d \mid \epsilon \qquad\qquad A \quad ::= \quad b$$

We assume that the input string is held in an array I, that i denotes the index of the current input symbol and that $error()$ is a function which terminates the whole parser and returns a suitable error message. The symbol \$ denotes the end-of-string symbol and the last entry in I is set to \$.

All strings of terminals derivable from ASd have the form bu, thus we guard the selection of the corresponding code with a test $I[i] = b$. For a one-symbol lookahead RD parser all the alternate choices are uniformally guarded with tests of this form. Then we have the following parse functions for the above grammar.

```
$parseS()$ {
   if $(I[i] = a)$ {
         if$(I[i] = a)$ { $i := i + 1;$}  else $error()$
         $parseS()$;   return }
   if $(I[i] = b)$ {
         $parseA()$; $parseS()$
```

```
        if($I[i] = d$) { $i := i + 1$ }  else $error()$;  return } }

$parseA()$ {
  if ($I[i] = b$) {
      if($I[i] = b$) { $i := i + 1$ }  else $error()$;  return }
  $error()$ }
```

We think of an RD parser as 'walking' the grammar using a given input string. The function $parseX()$ selects an alternate, α, of the rule for X, according to the current symbol in the input string being parsed, and then calls the parse functions associated with the symbols in α.

If the parser is implemented in a typical high level programming language, a call to a parse function $parseX()$ will cause the address of the action immediately after the call (the return address) to be written to the function call stack and the algorithm execution will jump to the start of the code for $parseX()$. When the end of the function is reached the return address is popped off the stack and the algorithm execution jumps to that address.

The problem is that the choice of alternate is not always uniquely determined by the current input symbol, and this can cause a naïve RD parser to incorrectly reject an input.

We shall use the grammar Γ_0, a modification of the grammar above, as a running example.

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

We have the following parse functions for Γ_0.

```
$parseS()$ {
  if ($I[i] = a$) {
        if($I[i] = a$) { $i := i + 1;$}  else $error()$
        $parseS()$;  return }
  if ($I[i] = a$) {
        $parseA()$; $parseS()$
        if($I[i] = d$) { $i := i + 1$ }  else $error()$;  return } }

$parseA()$ {
  if ($I[i] = a$) {
      if($I[i] = a$) { $i := i + 1$ }  else $error()$;  return }
  $error()$ }
```

In this case the RD parser will not work correctly because the second alternate will never be reached. The GLL algorithm explores all the execution paths through the algorithm, but to do this it needs to explicitly manage the function call and return mechanism in order to support multiple call stacks.

We replace the parse functions for Γ_0 with labels and goto statements and an explicit stack. We also add a final statement labelled L_0 which checks whether the parser has read all the input. The length of the input is denoted by m. We assume that the stack is initialised with L_0 on the bottom, and that there is a function $push(R)$ which pushes its argument on to the top of the stack, and a function $pop()$ which takes the top element off the stack and makes it the current label, denoted by L.

```
$L_S$:  if $(I[i] = a)$ {
           if$(I[i] = a)$ { $i := i + 1$ }  else $error()$
           $push(R_{S_1})$;  goto $L_S$
$R_{S_1}$:    $pop()$;  goto $L$  }

        if $(I[i] = a)$ {
           $push(R_{A_1})$;  goto $L_A$
$R_{A_1}$:    $push(R_{S_2})$;  goto $L_S$
$R_{S_2}$:    if$(I[i] = d)$ { $i := i + 1$ }  else $error()$
           $pop()$; goto $L$  }
        else $pop()$; goto $L$

$L_A$: if $(I[i] = a)$ {
          if$(I[i] = a)$ { $i := i + 1$ }  else $error()$
          $pop()$;  goto $L$ }
        else $error()$

$L_0$:  if $(i = m)$  return success  else $error()$
```

Of course, if we run this algorithm with input ad it reports an error at L_0.

2.2 Dealing with nondeterminism

The problem with the parser given at the end of the previous section is that two alternates of the rule for S can be chosen when the first input symbol a has been seen, and from the structure of the algorithm, it will always be the first one aS.

The GLL algorithm addresses this by creating a 'descriptor' for each valid choice and then pursuing all of the corresponding execution threads in turn. A set $\mathcal{R}$ of pending descriptors is maintained and the GLL algorithm has an outer loop, labelled L_0, which removes a descriptor from $\mathcal{R}$ and resumes the corresponding execution thread.

Execution forks occur at the start of a parse function, when the alternate is chosen. The start of each alternate has to be recorded, so these are labelled. Each thread has an associated current stack but it would be impractical to keep a separate copy of each stack for each thread. The stacks are merged together into a single structure, the GSS, and the GSS node which corresponds to the top of the stack(s) associated with a given thread is included in the descriptor.

Thus, in this section, descriptors are triples which contain a label, a stack node and an input position (L, u, i). (In our later formal description of the parser, descriptors will be extended to include tree nodes.) When a descriptor is removed from $\mathcal{R}$ the process execution is resumed at position L, with input symbol $I[i]$ and stack whose stack top is u.

Because the stacks are merged, it is possible for a descriptor to represent several threads with different stacks. When a stack node is popped it may create several stacks which had previously been merged. Thus the *pop*() function creates descriptors, one for each of these stacks, which are added to $\mathcal{R}$, and a call to *pop*() is immediately followed by a return to the outer loop of the algorithm and the selection of the next element from $\mathcal{R}$.

We use c_U to denote the current stack top, a GSS node, and c_I to denote the current input position, an integer. The function *add*() creates a new descriptor and adds it to $\mathcal{R}$. To avoid repeated processing of threads, *add*() only creates a descriptor if it has not already been created. We replace the *push*() function in the naïve RD parser with a function *create*() which creates a new GSS node as a parent of the current stack top, if a suitable node does not already exist, and then returns this node. We will not go into further details of the functions *pop*(), *add*() and *create*() here. Formal descriptions of the versions of these functions needed for the parser version of GLL are given in Section 3.3 and the detailed behaviour of the recogniser versions of these functions, as they are used in this section, are given in [17].

The following is a sketch version of the GLL recognition algorithm for Γ_0 which shows the 'shape' of the control flow. The bookkeeping details for the full algorithm are discussed in the next section.

```
$L_0$: if $\mathcal{R} \neq \emptyset$ {
            remove $(L, u, i)$ from $\mathcal{R}$;  goto $L$ }
    if $(c_I = m)$  return success  else $error()$

$L_S$: if $(I[c_I] = a)$ $add(L_{S_1}, c_U, c_I)$
    if $(I[c_I] = a)$ $add(L_{S_2}, c_U, c_I)$
    $add(L_{S_3}, c_U, c_I)$;  goto $L_0$

$L_{S_1}$: $c_I := c_I + 1$
     $c_U := create(R_{S_1}, c_U, c_I)$;  goto $L_S$
```

```
$R_{S_1}$: $pop(c_U, c_I)$;  goto $L_0$

$L_{S_2}$: $c_U := create(R_{A_1}, c_U, c_I)$;  goto $L_A$
$R_{A_1}$: $c_U := create(R_{S_2}, c_U, c_I)$;  goto $L_S$
$R_{S_2}$: if($I[c_I] = d$) { $c_I := c_I + 1$ }  else $error()$
       $pop(c_U, c_I)$;  goto $L_0$

$L_{S_3}$: $pop(c_U, c_I)$;  goto $L_0$

$L_A$: if ($I[c_I] = a$) $add(L_{A_1}, c_U, c_I)$
      goto $L_0$
$L_{A_1}$: $c_I := c_I + 1$
       $pop(c_U, c_I)$;  goto $L_0$
```

Note, a simple RD parser implements an ϵ-alternate by omitting the final *error*() call at the end of the parse function, thus causing the parse function to return normally having done nothing. In the general case it maybe that both the ϵ-alternate and other alternates form valid execution threads so we have to explicitly create a descriptor for this case.

2.3 A GLL recogniser for Γ_0

In summary then, a GLL recogniser includes labelled lines of three types: return, nonterminal and alternate. Return labels, R_{X_i}, are used to label 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_j}, are used to label the first line of what would be the code corresponding to the j^{th}-alternate, α_j say, of X. The outer loop of the algorithm is labelled L_0.

As we described above, the return labels are stored in a GSS whose nodes contain a return label and an input string position. For the GSS node, c_U, corresponding to the top of the current stack, the function $create(L, u, i)$ finds or creates, and then returns, a GSS node labelled (L, i) which has child node u. The GSS is initialised to have L_0 at the bottom and when this node is popped the control flow moves to the final test for the end of string symbol. To avoid an empty GSS at this point, a dummy base node labelled \$ is created as a child of the first node $(L_0, 0)$.

Process descriptors of the form (L, u, i), created by $add(L, u, i)$, are held in a set $\mathcal{R}$. To avoid creating the same descriptor twice we maintain sets

$$\mathcal{U}_i = \{(L, u) \mid (L, u, i) \text{ has been added to } \mathcal{R}\}$$

and a descriptor (L, u, i) is only added to $\mathcal{R}$ if (L, u) is not already in $\mathcal{U}_i$. To reduce the space requirements each $\mathcal{U}_k$ may be deleted once all descriptors (L, u, j) such that $j \leq k$ have been processed.

(For a GLL parser the edges of the GSS are labelled and *create*() and *add*() have different input parameters in this case, see Section 3.3.)

We assume that the input has length m and is stored in an array I whose entries are indexed from 0 to m. The last entry in I, $I[m]$, is set to end-of-string symbol \$. The c_I denotes the current input index.

The following is a full GLL recognition algorithm for Γ_0 and the first example GSS in Section 1 is the GSS produced when this algorithm is executed on the input *aad*\$.

```
     create GSS nodes $u_1 := (L_0, 0)$, $u_0 := \$$ and an edge $(u_0, u_1)$
     $c_I := 0$; $\mathcal{R} := \emptyset$; $c_U := u_1$
     for $0 \leq j \leq m$ { $\mathcal{U}_j := \emptyset$ }
     goto $L_S$
$L_0$: if $(\mathcal{R} \neq \emptyset)$ { remove $(L, u, j)$ from $\mathcal{R}$
                        $c_U := u$; $c_I := j$;  goto $L$ }
     else  if $((L_0, u_0, m) \in \mathcal{U}_m)$ report success  else report failure

$L_S$: if $(I[c_I] = a)$ { $add(L_{S_1}, c_U, c_I)$; $add(L_{S_2}, c_U, c_I)$ }
     if $(I[c_I] \in \{d, \$\})$  $add(L_{S_3}, c_U, c_I)$
     goto $L_0$
$L_{S_1}$: $c_I := c_I + 1$
     if $(I[c_I] \in \{a, d, \$\})$ { $c_U := create(R_{S_1}, c_U, c_I)$;  goto $L_S$ }
     else  goto $L_0$
$R_{S_1}$: $pop(c_U, c_I)$;  goto $L_0$
$L_{S_2}$: $c_U := create(R_{A_1}, c_U, c_I)$;  goto $L_A$
$R_{A_1}$: if $(I[c_I] \in \{a, d\})$ { $c_U := create(R_{S_2}, c_U, c_I)$;  goto $L_S$ }
     else  goto $L_0$
$R_{S_2}$: if$(I[c_I] = d)${ $c_I := c_I + 1$; $pop(c_U, c_I)$ };   goto $L_0$
$L_{S_3}$: $pop(c_U, c_I)$;  goto $L_0$
$L_A$:  if $(I[c_I] = a)$ $add(L_{A_1}, c_U, c_I)$;  goto $L_0$
$L_{A_1}$:  $c_I := c_I + 1$; $pop(c_U, c_I)$;  goto $L_0$
```

3 Building derivation trees

Ultimately we want a translator to generate target code and for this the semantics of the input must be established. Simply knowing that the input is syntactically correct is not adequate. The semantics of a language are usually reflected in the syntactic structure and thus the semantic analyser can use the derivation of the input as a starting point. Parsing algorithms differ from recognisers in that they also produce some form of the derivation of the input.

For LL(1) grammars, that is grammars for which the choice of alternate in a parse function is always uniquely determined by the input symbol, the extension of an RD parser to include derivation tree construction is trivial. Thus the distinction between a recogniser and a parser is often blurred. However, in general the extension of a recogniser algorithm to one which constructs derivations is not straightforward, see [18] for an illustration of this based on Earley's algorithm. In the rest of this paper we describe the GLL parsing algorithm.

3.1 Shared packed parse forests (SPPF) and parser descriptors

A *derivation tree* is an ordered tree whose root is labelled with the start symbol, leaf nodes are labelled with a terminal or ϵ and interior nodes are labelled with a nonterminal, A say, and have a sequence of children corresponding to the symbols on the right hand side of a rule for A.

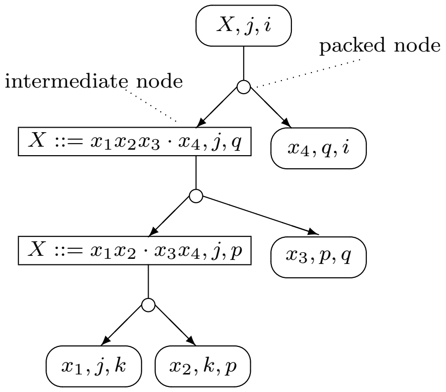
A grammar is *ambiguous* if there is some string which has two or more derivation trees. For some grammars some strings have infinitely many derivation trees and thus general parsers cannot simply construct all the possible derivation trees.

The complexity of parsing algorithms continues to be an open problem. There is no known linear time general parsing algorithm; the best practical general algorithms have cubic complexity. In their recogniser forms the CYK algorithm [25] is worst-case cubic order on grammars in Chomsky normal form and Earley's algorithm [6] is worst-case order cubic on general context free grammars and worst-case quadratic order on non-ambiguous grammars. For the parser versions the situation is even more complicated. Earley's own sketch of a parsing version of his algorithm is incorrect and Tomita's GLR algorithm [20], which was designed with tree building in mind, is unbounded polynomial order even on some ϵ-free grammars.

GLR parsers represent the complete set of derivation trees for a string using a *shared packed parse forest* (SPPF) which is designed to reduce the space required to represent multiple derivation trees. In an SPPF, nodes which have the same tree below them are shared and nodes which correspond to different derivations of the same substring from the same nonterminal are combined by creating a packed node for each family of children. In order to create a cubic Earley parser [18] and the worst-case cubic BRNGLR parser [19] we used a binarised form of SPPF which contains additional *intermediate* nodes. As we shall show, GLL recognisers are also worst-case cubic complexity and we can also turn them in to worst-case cubic parsers using binarised SPPFs.

In binarised SPPFs there are nodes labelled with positions on the right hand sides of grammar rules, as well as nodes labelled with grammar symbols. A *grammar slot* is any position immediately before or after any symbol in any alternate. We write grammar slots in the same way as LR(0) items are written, so $X ::= \alpha \cdot \beta$ is the grammar position immediately after the last symbol in α. Then, a binarised SPPF has three types of SPPF nodes: symbol nodes, with labels of the form (x, j, i) where x is a terminal, nonterminal or ϵ and $0 \leq j \leq i \leq m$; intermediate nodes, with labels of the form (t, j, i); and packed nodes, with labels for the form (t, k), where $0 \leq k \leq m$ and t is a grammar slot, $X ::= \alpha \cdot \beta$. We shall call (j, i) the *extent* (j, i are the left and right extents respectively) of the SPPF symbol or intermediate node and k the *pivot* of the packed node. The packed nodes allow the SPPF to represent multiple derivation trees and the intermediate nodes ensure that the SPPF has at worst-cubic size.

Terminal symbol nodes have no children. Nonterminal symbol nodes, (A, j, i), have packed node children of the form $(A ::= \gamma\cdot, k)$ and intermediate nodes, (t, j, i), have packed node children with labels of the form (t, k), where $j \leq k \leq i$. While symbol and intermediate nodes may have several children, a packed node has one or two children, the right child is a symbol node (x, k, i), and the left child (if it exists) is a symbol or intermediate node, (s, j, k). For example, for the rule $X ::= x_1x_2x_3x_4$ we have SPPF fragment



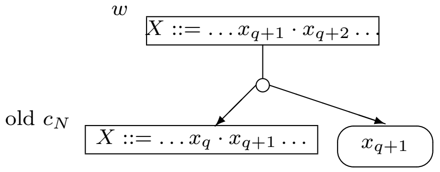
Note: of course the SPPFs have unnecessary packed nodes in the (very common) case that a nonterminal does not generate any ambiguity. Once the parse is complete the SPPF can be walked and the packed nodes which are 'only children' can be removed. This can also be done as the SPFF is constructed by only constructing packed nodes when a second family of children is added to a node. In this case some care must be taken in the implementation because the label of the potential first packed node will have to be stored in case it is needed. We shall not consider this optimisation further in this paper.

Grammar slots can be used as the code labels. The label L_{X_i} corresponding to the alternate $X ::= \alpha_i$ is euqivalent to the slot $X ::= \cdot\alpha_i$ and the return label R_{X_l} is equivalent to $Y ::= \tau X \cdot \mu$, where this is the slot immediately after the lth instance of X. Thus, throughout the remainder of this paper, we shall take the grammar slots together with the labels L_X, $X \in \mathbf{N}$, and L_0 to be the set of code labels.

In each descriptor, as well as recording the GSS node, return label and input position, we also need to record the current SPPF node. Just before the descriptor is constructed a new SPPF is created and this is then recorded. Thus *parser descriptors* are 4-tuples of the form (L, u, i, w) where w is an SPPF node, and these are divided into disjoint sets of triples $\mathcal{U}_i$. Descriptors in which w is not the dummy node will be constructed as the result of a *pop* action. The right child of the new node w will be the existing current SPPF node, and hence will be available, but the left child will need to be retrieved. For an ambiguous grammar there may be several left children so we record each one as the label of an edge from the GSS node that will be popped. The left child is retrieved from the labelled GSS edge when the source node of the edge is popped. The label of the new SPPF node is the grammar slot which is also the return label held in the GSS node. The GSS is thus a labelled directed graph in which the edges are labelled with either an SPPF node or with the 'dummy' node \$ and the nodes are labelled with a unique pair of the form $(A ::= \alpha \cdot \beta, i)$. We call i the *index* of the GSS node.

We can imagine a GLL parser as having a pointer into the grammar, $X ::= x_1 \ldots x_q \uparrow x_{q+1} \ldots x_n$, and a current input pointer. The grammar pointers point to slots, $X ::= x_1 \ldots x_q \cdot x_{q+1} \ldots x_n$.[1] At each point in its execution a GLL parser has a current SPPF node, denoted by c_N, corresponding to the portion of the rule to the left of the current grammar pointer, $x_1 \ldots x_q$.

As the pointer moves past x_{q+1} the parser creates a new node, w, as we have discussed, with a packed node whose left child is c_N and whose right child is the node corresponding to x_{q+1}. Then w becomes the new c_N.


w
$X ::= \ldots x_{q+1} \cdot x_{q+2} \ldots$
old c_N
$X ::= \ldots x_q \cdot x_{q+1} \ldots$
x_{q+1}

If x_{q+1} is a terminal then a corresponding SPPF node $(x_{q+1}, c_I, c_I + 1)$, where c_I denotes the current input position, is found or constructed. This node is denoted by c_R. If x_{q+1} is a nonterminal then before moving past x_{q+1}, the

[1] Note, these are grammar *positions* thus if the grammar has repeated rules $A ::= a|a$ say then there are two distinct slots $A ::= \cdot a$.

grammar pointer moves to the start of the rules for x_{q+1} and descriptors are created to allow for non-deterministic alternate choice.

If x_{q+1} is a nonterminal, a GSS node, v, labelled with the grammar return position, $X ::= x_1 \ldots x_{q+1} \cdot x_{q+2} \ldots x_n$, is created and the edge from this node is labelled with c_N. At this point the construction of an SPPF subgraph rooted at x_{q+1} begins and there is no current SPPF node. So the descriptors are padded with a dummy node denoted by \$. Similarly, if $x_{q+1} = x_1$ there is no left portion of the rule and the dummy node \$ is used to label the corresponding GSS edge.

When the parser finishes 'matching' x_{q+1} the GSS node v is popped and the return position and SPPF node are recovered, allowing the next SPPF node to be constructed. In detail there may be several return configurations associated with a given pop action. For each edge (c_U, z, u) in the GSS, where c_U denotes the current stack top, $getNodeP()$ is called to construct an intermediate or symbol node, w, with a packed node whose left child is the node z retrieved from the GSS edge between c_U and u and whose right child is the SPPF node corresponding to x_{q+1}. A descriptor of the form $(R_{x_{q+1}}, u, c_I, w)$ is then created. When such a descriptor is removed from $\mathcal{R}$ execution continues with the input pointer at position c_I, the grammar pointer after x_{q+1}, and $c_U := u$ and $c_N := w$.

3.2 Example

Consider the grammar, Γ_1,

$$S ::= \ a \ S \ b \mid d \mid a \ d \ b$$

whose GLL parser is (note R_{S_1} is $S ::= aS.b$)

```
    read the input into $I$ and set $I[m] := \$$
    create GSS node $u_0 := (L_0, 0)$
    $c_I := 0$; $c_U := u_0$
    $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$
    for $0 \leq j \leq m$ { $\mathcal{U}_j := \emptyset$ }
    goto $L_S$
$L_0$: if $(\mathcal{R} \neq \emptyset)$ { remove $(L, u, i, w)$ from $\mathcal{R}$
                        $c_U := u$; $c_I := i$; $c_N := w$;  goto $L$ }
    else  if (there is an SPPF node $(S, 0, m)$) report success
    else report failure

$L_S$: if $(I[c_I] \in \{a\})$ { $add(L_{S_1}, c_U, c_I, \$)$; $add(L_{S_3}, c_U, c_I, \$)$ }
    if $(I[c_I] \in \{d\})$  $add(L_{S_2}, c_U, c_I, \$)$
```

```
     goto $L_0$
$L_{S_1}$: $c_N := getNodeT(a, c_I)$; $c_I := c_I + 1$
     if $(I[c_I] \in \{a, d\})$ { $c_U := create(R_{S_1}, c_U, c_I, c_N)$;  goto $L_S$ }
     else  goto $L_0$
$R_{S_1}$: if $(I[c_I] = b)$ $c_R := getNodeT(b, c_I)$  else  goto $L_0$
     $c_I := c_I + 1$; $c_N := getNodeP(S ::= aSb., c_N, c_R)$;
     $pop(c_U, c_I, c_N)$;  goto $L_0$
$L_{S_2}$: $c_R := getNodeT(d, c_I)$
     $c_I := c_I + 1$; $c_N := getNodeP(S ::= d., c_N, c_R)$
     $pop(c_U, c_I, c_N)$;  goto $L_0$
$L_{S_3}$: $c_N := getNodeT(a, c_I)$; $c_I := c_I + 1$
      if $(I[c_I] = d)$ $c_R := getNodeT(d, c_I)$  else  goto $L_0$
      $c_I := c_I + 1$;  $c_N := getNodeP(S ::= ad.b, c_N, c_R)$
      if $(I[c_I] = b)$ $c_R := getNodeT(b, c_I)$  else  goto $L_0$
      $c_I := c_I + 1$;  $c_N := getNodeP(S ::= adb., c_N, c_R)$
      $pop(c_U, c_I, c_N)$;  goto $L_0$
```

The base node, u_0, of the GSS is never popped and its label is just a dummy label. We have used L_0 as the dummy label, it is equivalent to the slot $S' ::= S\cdot$ in an ‘augmented’ form of the grammar. However, as this augmented rule is never used there is no need to actually augment the grammar for a GLL parser.

3.3 The GSS, SPPF and descriptor handling functions

We define $\text{FIRST}_{\mathbf{T}}(A) = \{t \in \mathbf{T} | \exists\alpha(A \overset{*}{\Rightarrow} t\alpha)\}$ and $\text{FOLLOW}_{\mathbf{T}}(A) = \{t \in \mathbf{T} \mid \exists\alpha, \beta(S \overset{*}{\Rightarrow} \alpha A t \beta)\}$. If A is nullable we define $\text{FIRST}(A) = \text{FIRST}_{\mathbf{T}}(A) \cup \{\epsilon\}$ and if $S \overset{*}{\Rightarrow} \alpha A$ we define $\text{FOLLOW}(A) = \text{FOLLOW}_{\mathbf{T}}(A) \cup \{\$\}$. 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 $A \overset{*}{\Rightarrow} \epsilon$ then $\text{FIRST}(A) \cap \text{FOLLOW}(A) = \emptyset$.

It is the nature of a GSS that a node u is often the top element of several stacks; the GSS compactness is achieved from node reuse by sharing stack tops. Thus when the top element is popped, by the function *pop*(), this can result in several new stack tops, one for each child of u. New children can be added to u and if this happens after the node has been popped, then the pop has to be applied to the new child. It is not possible, in general, to order the actions so that no pop is applied until after all the children have been added, so this situation is covered in the *create*() function as follows. We keep a record of the pop actions that have been performed so that they can be applied to new children if necessary. We use a set $\mathcal{P}$ which contains pairs (u, z) for which a pop action has been executed. Suppose that (L, j) is the label of u. When a new edge, labelled w say, is added to u, for all $(u, z) \in \mathcal{P}$, the parent node y of w and z is found or created and the corresponding descriptor is added to $\mathcal{R}$. See [17] for further details.

The three functions *add*(), *create*() and *pop*(), mentioned above, which build the GSS and create and store processes for subsequent execution are now defined as follows. (Throughout α and β denote possibly empty strings of terminals and nonterminals, x denotes a single terminal or nonterminal, and L denotes a code label, i.e. a grammar slot or L_X or L_0.)

```
$add(L, u, i, w)$ {
    if (($L, u, w) \notin \mathcal{U}_i$ { add $(L, u, w)$ to $\mathcal{U}_i$, add $(L, u, i, w)$ to $\mathcal{R}$ } }

$pop(u, i, z)$ {
    if ($u \neq u_0$) {
        let $(L, k)$ be the label of $u$
        add $(u, z)$ to $\mathcal{P}$
        for each edge $(u, w, v)$ {
            let $y$ be the node returned by $getNodeP(L, w, z)$
            $add(L, v, i, y)$) } } }

$create(L, u, i, w)$ {
    if there is not already a GSS node labelled $(L, i)$ create one
    let $v$ be the GSS node labelled $(L, i)$
    if there is not an edge from $v$ to $u$ labelled $w$ {
        create an edge from $v$ to $u$ labelled $w$
        for all ($(v, z) \in \mathcal{P}$) {
            let $y$ be the node returned by $getNodeP(L, w, z)$
            $add(L, u, h, y)$ where $h$ is the right extent of $z$ } }
    return $v$ }
```

We also use a function *test*(), which checks the current input symbol against the current nonterminal and alternate, to provide a one-symbol lookahead guard on descriptor creation.

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

The following functions build the SPPF.

```
$getNodeT(x, i)$ {
    if ($x = \epsilon$) $h := i$ else $h := i + 1$
    if there is no SPPF node labelled $(x, i, h)$ create one
```

```
  return the SPPF node labelled $(x, i, h)$ }

$getNodeP(X ::= \alpha \cdot \beta, w, z)$ {
  if ($\alpha$ is a terminal or a non-nullable nontermial and if $\beta \neq \epsilon$)  return $z$
  else {
    if ($\beta = \epsilon$)  $t := X$   else  $t := (X ::= \alpha \cdot \beta)$
    suppose that $z$ has label $(q, k, i)$
    if ($w \neq \$$) {
      suppose that $w$ has label $(s, j, k)$
      if there does not exist an SPPF node $y$ labelled $(t, j, i)$ create one
      if $y$ does not have a child labelled $(X ::= \alpha \cdot \beta, k)$
                        create one with left child $w$ and right child $z$ }
    else {
      if there does not exist an SPPF node $y$ labelled $(t, k, i)$ create one
      if $y$ does not have a child labelled $(X ::= \alpha \cdot \beta, k)$
                        create one with child $z$ }
    return $y$ } }
```

4 Formal definition of the GLL approach

In this section we give the formal templates required to generate a GLL parser from a context free grammar.

Each nonterminal instance on the right hand sides of the grammar rules is given an instance number. We write A_l to indicate the lth 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.

4.1 Dealing with grammar slots

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

For a terminal a we define

$$\begin{aligned} code(A ::= \alpha \cdot a\beta) \quad = \quad & \mathbf{if}(I[c_I] = a)\ c_R := getNodeT(a, c_I)\ \ \mathbf{else}\ \ \mathbf{goto}\ L_0 \\ & c_I := c_I + 1;\ \ c_N := getNodeP(A ::= \alpha a \cdot \beta, c_N, c_R) \end{aligned}$$

For a nonterminal instance X_l we define

```
$\mathit{code}(A ::= \alpha \cdot X_l\beta)$   =   $\textbf{if}(\mathit{test}(I[c_I], A, X\beta)$ {
                                         $c_U := \mathit{create}(R_{X_l}, c_U, c_I, c_N);$   $\textbf{goto } L_X$ }
                                     $\textbf{else goto } L_0$
                            $R_{X_l}$ :
```

4.2 Dealing with alternates

For each production $A ::= \alpha_k$ we define $\mathit{code}(A ::= \alpha_k)$ as follows. Let $\alpha_k = x_1x_2\dots x_f$, 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

```
$\mathit{code}(A ::= \epsilon)$   =   $c_R := \mathit{getNodeT}(\epsilon, c_I);$  $c_N := \mathit{getNodeP}(A ::= \cdot, c_N, c_R)$
                       $\mathit{pop}(c_U, c_I, c_N);$  $\textbf{goto } L_0$
```

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

```
$\mathit{code}(A ::= x_1)$   =   $c_R := \mathit{getNodeT}(x_1, c_I);$   $c_I := c_I + 1$
                       $c_N := \mathit{getNodeP}(X ::= x_1., c_N, c_R)$
                       $\mathit{pop}(c_U, c_I, c_N);$   $\textbf{goto } L_0$
```

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

```
$\mathit{code}(A ::= \alpha_k)$   =   $c_N := \mathit{getNodeT}(x_1, c_I);$   $c_I := c_I + 1$
                        $\mathit{code}(A ::= x_1 \cdot x_2 \dots x_f)$
                        $\dots$
                        $\mathit{code}(A ::= x_1 \dots x_{f-2} \cdot x_{f-1}x_f)$
                        $\mathit{code}(A ::= x_1 \dots x_{f-1} \cdot x_f)$
                        $\mathit{pop}(c_U, c_I, c_N);$   $\textbf{goto } L_0$
```

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

```
$\mathit{code}(A ::= \alpha_k)$   =   $c_U := \mathit{create}(R_{X_l}, c_U, c_I, c_N);$  $\textbf{goto } L_X$
                 $R_{X_l}$ : $\mathit{code}(A ::= x_1 \cdot x_2 \dots x_f)$
                        $\dots$
                        $\mathit{code}(A ::= x_1 \dots x_{f-2} \cdot x_{f-1}x_f)$
                        $\mathit{code}(A ::= x_1 \dots x_{f-1} \cdot x_f)$
                        $\mathit{pop}(c_U, c_I, c_N);$   $\textbf{goto } L_0$
```

4.3 Dealing with rules

Consider the grammar rule $A ::= \alpha_1 \mid \ldots \mid \alpha_p$. We define *code*$(A)$ as follows.

```
code(A)   =        if(test(I[c_I], A, α_1)) { add(L_{A_1}, c_U, c_I, $) }
                   ...
                   if(test(I[c_I], A, α_p)) { add(L_{A_p}, c_U, c_I, $) }
                   goto L_0
           L_{A_1} : code(A ::= α_1)
                   ...
           L_{A_p} : code(A ::= α_p)
```

4.4 Building a GLL parser for a general CFG

We suppose that the nonterminals of the grammar Γ are $A, \ldots, X$.
m is a constant integer whose value is the length of the input
I is a constant integer array of size $m + 1$
c_I is an integer
GSS is a digraph whose nodes are labelled with elements of the form (L, j) and whose edges are labelled with SPPF nodes
c_U is a GSS node, c_N and c_R are SPPF nodes
$\mathcal{P}$ is a set of GSS node, SPPF node pairs, used by *create*()
$\mathcal{R}$ is a set of parser descriptors
$\mathcal{U}_i$ corresponds to a set of parser descriptors with right extent i.

Then the GLL parsing algorithm for Γ is given by:

```
     read the input into $I$ and set $I[m] := \$$
     create GSS node $u_0 = (L_0, 0)$
     $c_U := u_0$; $c_N := \$$; $c_I := 0$
     for $0 \le j \le m$ { $\mathcal{U}_j := \emptyset$}
     $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$
     goto $L_S$
$L_0$: if $\mathcal{R} \ne \emptyset$ {
          remove a descriptor, $(L, u, i, w)$ say, from $\mathcal{R}$
          $c_U := u$; $c_N := w$; $c_I := i$;  goto $L$ }
     else  if (there exists an SPPF node labelled $(S, 0, m)$) { report success }
           else { report failure }

$L_A$: code$(A)$
    ...
$L_X$: code$(X)$
```

4.5 A GLL parser for Γ_0

```
     read the input into $I$ and set $I[m] := \$$
     create GSS node $u_0 := (L_0, 0)$
     $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$
      $c_U := u_1$; $c_N := \$$; $c_I := 0$
      **for** $0 \leq j \leq m$ { $\mathcal{U}_j := \emptyset$ }
      **goto** $L_S$
$L_0$: **if** $(\mathcal{R} \neq \emptyset)$ { remove $(L, u, i, w)$ from $\mathcal{R}$
                    $c_U := u$; $c_I := i$; $c_N := w$;  **goto** $L$ }
      **else**  **if** (there exists and SPPF node labelled $(S, 0, m)$) report success
           **else** report failure

$L_S$: **if** $(test(I[c_I], S, ASd))$  $add(L_{S_1}, c_U, c_I, \$)$
      **if** $(test(I[c_I], S, aS))$ $add(L_{S_2}, c_U, c_I, \$)$
      **if** $(test(I[c_I], S, \epsilon))$ $add(L_{S_3}, c_U, c_I, \$)$
      **goto** $L_0$
$L_{S_1}$: $c_U := create(R_{A_1}, c_U, c_I, c_N)$;  **goto** $L_A$
$R_{A_1}$: **if** $(test(I[c_I], S, Sd))$ {$c_U := create(R_{S_1}, c_U, c_I, c_N)$;  **goto** $L_S$}
       **else goto** $L_0$
$R_{S_1}$: **if**$(I[c_I] = d)$ $c_R := getNodeT(d, c_I)$  **else goto** $L_0$
       $c_I := c_I + 1$;  $c_N := getNodeP(S ::= ASd\cdot, c_N, c_R)$
       $pop(c_U, c_I, c_N)$;   **goto** $L_0$
$L_{S_2}$: $c_N := getNodeT(a, c_I)$; $c_I := c_I + 1$
      **if** $(test(I[c_I], S, S)$ {$c_U := create(R_{S_2}, c_U, c_I, c_N)$;  **goto** $L_S$}
       **else goto** $L_0$
$R_{S_2}$: $pop(c_U, c_I, c_N)$;  **goto** $L_0$
$L_{S_3}$: $c_R := getNodeT(\epsilon, c_I)$; $c_N := getNodeT(S ::= \cdot, c_N, c_R)$
      $pop(c_U, c_I, c_N)$;  **goto** $L_0$
$L_A$: **if** $(test(I[c_I], A, a))$ $add(L_{A_1}, c_U, c_I, \$)$;  **goto** $L_0$
$L_{A_1}$: $c_R := getNodeT(a, c_I)$; $c_I := c_I + 1$
      $c_N := getNodeP(A ::= a\cdot, c_N, c_R)$
      $pop(c_U, c_I, c_N)$;  **goto** $L_0$
```

4.6 Improvement for LL(1) nonterminals

In the case where a grammar is LL(1) a recursive descent parser will be deterministic and, equivalently, a GLL algorithm will remove a descriptor from $\mathcal{R}$ immediately after it is inserted. Even if the grammar is not LL(1), the GLL algorithm does not need to create descriptors for the LL(1) nonterminals (see Section 3.3 for the definition of an LL(1) nonterminal). The GLL recogniser described in [17] uses a different template for *code*(A) for an LL(1) nonterminal. In this template the calls to $add(L, c_U, c_I, \$)$ are replaced with the code at L. This reduces the number of elements added to $\mathcal{R}$ but does not effect the order of the algorithm. We can make the same modification to the parser version presented in this paper, and we would expect production implementations to do so as this improves the runtime and space requirements for typical grammars and input strings.

However, even without these improvements the parser is worst-case cubic, as we shall show in the next section. We have presented the algorithm in its pure form in this paper as fewer special cases make it easier for the reader to reason about the algorithm's behaviour.

5 The complexity of a GLL parser

Suppose that the input string has length m. We begin by looking at the main algorithm flow as determined by the **goto** statements.

A GLL parser has a main loop which removes elements from $\mathcal{R}$. After an element is removed the parser jumps to a label L. Looking at the code templates we see that the parser execution then continues linearly until either a **goto** L_0 statement is executed, beginning the next iteration of the main loop, or a *create*() or *pop*() function call is made. In the former case, once the function call returns, there is a jump to a label of the form L_A, an m-independent number of calls to the *add*() function, and then a jump to L_0. In the latter case, when the *pop*() call returns there is a jump to L_0.

For a GLL recogniser it is reasonably easy to prove that the algorithm is worst-case $O(m^3)$. The GSS nodes have unique labels of the form (R, i), $0 \leq i \leq m$, thus there are at most $O(m)$ of them, each of which has at most $O(m)$ out-edges. Using an $O(m^2)$ array the GSS can be implemented so that lookup executes in constant time with respect to m. The set $\mathcal{P}$ used by *create*() can also be implemented to allow constant look-up time, thus both *pop*() and *create*() execute in worst-case $O(m)$ time. The recogniser descriptors have the form (L, u, i) where u is a GSS node. Thus there are at most $O(m^2)$ elements added to $\mathcal{R}$ and thus the complexity of the full GLL recogniser is at most $O(m^3)$.

For a GLL *parser* we have to work harder to ensure that the algorithm is worst-case cubic. As we have already discussed, we have to 'binarise' the SPPF. In addition we have to create SPPF nodes at the correct points in the algorithm. It would be natural to store the two current SPPF children in the descriptor and then form their parent node w after the return call which has constructed them. This is because it is the calling slot which determines the label of the parent node. However, this may potentially create $O(m^3)$ descriptors. Thus we have exploited the fact that the calling slot information can be included in the corresponding GSS node to allow the *pop*() function to create the parent node before it creates the descriptor, thus descriptors contain only one SPPF node.

It is also necessary to ensure that $getNodeP()$ and $getNodeT()$ execute in time independent of m. This can be done because, as we shall show, there are at most $O(m^2)$ SPPF nodes with $O(m)$ children, and each of these children has out-degree at most two, thus, as all other SPPF nodes have at most two children, and the SPPF can be represented using an $O(m^3)$ array. In fact there are more efficient SPPF implementations which still ensure that the $getNode$ functions are constant time, but for the complexity proof we require only the existence of some constant look-up time representation of at most cubic size.

The algorithm has been designed so that at any point if c_U, the current GSS stack-top node, has index j then the current SPPF node, c_N, is either \$ or has extent (j, c_I) or will not be used further before being reinitialised at L_0. In particular, for any descriptor (L, u, i, w) we have that either $w = \$$ or w has extent (j, i) where u has index j. This is certainly true at the start of the algorithm. We consider the points in the algorithm where the values of c_I, c_N and c_U are changed. Looking at the code templates we see that c_I and c_N are always assigned together in a manner that ensures that c_N has extent (j, c_I). When c_N is assigned a new value the value of its left extent, j, does not change thus the relationship with c_U is preserved. The value of c_U is only changed by a call to the *create* function. This function finds or creates a GSS node, v, indexed with c_I and labels an edge from v to c_U with c_N. This is the only way in which parts of the GSS are constructed so we have that all edges in the GSS are of the form (v, w, u) where if v has index i and u has index j then $w = \$$ or w has extent (j, i). After *create* returns, c_U is indexed by c_I, the algorithm performs an action of the form `goto` $\mathtt{L_X}$, then constructs descriptors $(L_{X_j}, c_U, c_I, \$)$ and performs `goto` $\mathtt{L_0}$. Thus c_N is not used after c_U is changed by *create*. Descriptors constructed at labels of the form L_X have SPPF node \$. Descriptors constructed by the *pop* function have the form (L, u, i, x) where there is a GSS edge (v, w, u). Looking at the *pop* function, at the time of creation we have $v = c_U$, $i = c_I$ and x has grandchildren w and c_N. Thus, if v has index k and u has index j then w has extent (j, k) and c_N has extent (k, i) so x has extent (j, i). If $(v, z) \in \mathcal{P}$ then looking again at the *pop* function which creates these pairs, we have that w has left extent the index, g say, of c_U so x has extent (g, h). Thus, in both cases the relationship above holds for c_I, c_U and c_N when these values are reinitialised at L_0. We summarise this discussion in the following lemma.

Lemma 1 *(a) For any descriptor (L, u, i, w) we have that either $w = \$$ or w has extent (j, i) where u has index j.*
(b) For any element $(u, z) \in \mathcal{P}$ we have that z has left extent j where u has

index j.
(c) Given a GSS (v, w, u) edge from v to u labelled w, suppose that v has label $(R, A ::= \alpha X \cdot \beta, i)$ and u has label (R', t, j). If $\alpha = \epsilon$ then $w = \$$, if $\alpha = x$, where x is a terminal, nonterminal or ϵ, then w has label (x, j, i), and if $|\alpha| \geq 2$ then w is the intermediate node with label $(A ::= \alpha \cdot X\beta, j, i)$.

Theorem 2 *For a GSS generated by a GLL parser for a CFG Γ on an input string of length m, each node is the source node for at most $O(m)$ edges, and hence the GSS has at most $O(m)$ nodes and $O(m^2)$ edges.*

Proof The node u_0 has no out-edges. All the other GSS nodes have unique labels of the form $(R, A ::= \alpha X \cdot \beta, i)$ where R is the label associated with the grammar slot $A ::= \alpha X \cdot \beta$ and $0 \leq i \leq m$. Thus the GSS has at most $O(m)$ nodes.

For a GSS edge (v, w, u) whose source node is v, target node is u and whose label is the SPPF node w, the extent of w is (j, i) where j, i is the index of u, v respectively. Thus there are at most $O(m)$ edges whose source node is v and hence the GSS has at most $O(m^2)$ edges.

Theorem 3 *The SPPF generated by a GLL parser on an input string of length m has at most $O(m^3)$ nodes and edges.*

Proof The SPPF terminal symbol nodes have labels of the form $(a, i, i+1)$ and there are m of these. The SPPF ϵ nodes have labels (ϵ, i, i) and hence there are at most m of these. The SPPF nonterminal symbol nodes have labels of the form (A, j, i) and there are at most $O(m^2)$ of these.

The intermediate nodes have labels of the form (t, j, i), where there is a grammar slot $A ::= \alpha \cdot \beta$, so there are at most $O(m^2)$ of these.

The packed nodes are children of a symbol or intermediate node and have labels (t, k) where t is a grammar slot $A ::= \alpha \cdot \beta$. If the parent node is a nonterminal it has label (A, j, i) and if it is an intermediate node it has label (t, j, i), for some $0 \leq j \leq i \leq m$. So there are at most $O(m)$ packed node children of each parent and hence the SPPF has at most $O(m^3)$ packed nodes, and at most $O(m^3)$ nodes in total.

The packed nodes have at most two children so there are at most $O(m^3)$ edges whose source node is a packed node. The symbol and intermediate nodes have at most $O(m)$ children, all of which are packed nodes. Thus there are at most $O(m^3)$ edges whose source node is symbol or intermediate node. This gives a total of at most $O(m^3)$ edges.

Theorem 4 *The runtime and space complexities for a GLL parser for a CFG Γ are at most $O(m^3)$.*

Proof Theorems 3 and 2 show that the space required for the SPPF and GSS is at most $O(m^3)$.

We have seen, Lemma 1(a), that there are at most $O(m^2)$ descriptors and the function *add* ensures that each one is added to $\mathcal{R}$ at most once. Thus the outer loop of the algorithm, at L_0, executes at most $O(m^2)$ times. We now consider the execution of the algorithm for one descriptor (L, u, i, w).

If L is of the form L_A then the algorithm calls *add* an m-independent number of times, and then jumps back to L_0. The $\mathcal{U}_i$ are equivalent to disjoint subsets of $\mathcal{R}$ and are each of size $O(m)$. Thus *add*() can be implemented to execute the search in time independent of m.

If L is of the form L_{A_l} then the algorithm may make several calls to the $getNode$ functions, but the number of these calls is bounded by the length of the grammar rule. The SPPF can be implemented to permit constant search time so these calls do not add to the complexity of the algorithm. Eventually the algorithm will get to either a jump to L_0 or a call to *pop* or *create*. Since a GSS node has at most $O(m)$ out edges and $\mathcal{P}$ has at most $O(m)$ elements corresponding to a given node v, both of these functions have complexity at most $O(m)$. When a *pop* function returns the algorithm immediately jumps to L_0. When *create* returns the algorithm jumps to a label of the form L_X and then jumps to L_0 after an m-independent number of call to *add*.

If L is of the form L_{R_k} the algorithm behaves in the same way as when L is of the form L_{A_l}. Thus we have that the complexity of the algorithm between jumps to L_0 is worst-case $O(m)$ and hence the overall complexity of a GLL algorithm on worst-case grammars is $O(m^3)$.

6 Implementation and experimental results

In this section we look at some details of our implementations and give performance statistics for both highly ambiguous 'pathological' grammars and for the ANSI-C standard grammar. We shall show that parsing of long strings in worst-case grammars is practical on current computer systems, and the indications are that parsing of conventional near-deterministic grammars can be made competitive with traditional techniques.

6.1 Implementing dynamic **goto**

The parser and recogniser algorithms presented here and in [17] use a 'dynamic' `goto` statement in the sense that labels are assigned to variables, and the construction `goto` L is used where L is variable containing a label. This is done to allow descriptors to contain control flow contexts.

In assembly language, direct implementation is easy since the address of a piece of code is a first class object, but only a few high level programming languages provide variables that can hold labels — for instance early FORTRAN compilers supported the assigned GO TO statement (although it was deleted from FORTRAN-95). More interestingly, the GNU C99 and C++ compilers provide the `&&`*label* operator which returns a `void*` pointer representing the address of the labelled code, and allows constructions such as `goto *ptr` so that control can be transferred to the value of a void pointer acting as label variable. However, use of these extension features is rightly deprecated by those interested in safety and security of code. Of course, some languages such as Java do not provide support for even static `goto` statements.

A more structured alternative may be to encode the parser fragments as separate functions and use function pointers. However, function call and return represents a potentially significant overhead at runtime given that many parser fragments are extremely short. In Java implementations, encoding the fragments as member functions and using virtual dispatch may be faster than using state-based dispatch. We shall explore these issues in a future engineering-oriented paper.

As we mentioned above, our preferred technique is to establish an enumeration corresponding to the labels within the parser, and effectively use a state-machine based approach to dispatch the fragments. In effect, we associate a unique integer with each label and use that integer in the descriptors.

In C, then, the following set of fragments with assumed start state L_0:

L_0: *action* **goto** L_1;
L_1: *action* **goto** L_2;
L_2: *action* **goto** L_0;

is written as

```
enum states {L0, L1, L2};
enum states state=L0;
while(1) switch(state) {
   case L0: /*action*/ state = L1; break;
   case L1: /*action*/ state = L2; break;
   case L2: /*action*/ state = L0; break;  }
```

There is an overhead here compared to the assembler level implementation: at the end of each fragment we load the state variable with the next fragment to be executed, and then break out of the switch, loop back and re-evaluate the state variable. The implementer may wish to use static `goto` statements if they are available for the static jumps.

6.2 Implementing $\mathcal{R}$

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 interrupted 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, w). It is easy to see that if $\mathcal{R}$ doesn't contain any elements of the form (L, u, k, w) with $k \leq j$ then no further elements of this form will be created. Thus, if the elements of $\mathcal{R}_j$ are processed before any of those in R_{j+1}, $0 \leq j < m$, then the sets $\mathcal{U}_j$ will be constructed in the corresponding order allowing $\mathcal{U}_j$ to be deleted once $\mathcal{R}_k = \emptyset$, $k \leq j$.

6.3 Implementing queries

Function *create*(L, u, i, w) contains the conditional:

if there is not already a GSS node labelled (L, i) ...

Similarly, function $getNodeT(x, i)$ contains the clause

if there is no SPPF node labelled (x, i, h) ...

and function $getNodeP(X ::= \alpha \cdot \beta, z, w)$

if there does not exist an SPPF node y labelled (t, j, i) ...

We could implement these queries by linearly searching the SPPF or GSS structures, but of course they are worst-case cubic and quadratic, respectively, in the length of the input, so a linear search could require quadratic or cubic time. To preserve the worst-case cubic execution time of the GLL algorithm, we need to be able to answer queries in constant time.

A simple solution is to maintain arrays *SPPFindex* and *GSSindex* of pointers to nodes, with each possible SPPF or GSS node having its own unique cell in the corresponding index array. If the node has not yet been created, then the cell is NULL. If it has, then the cell points to the node.

In the case of the SPPF, a label comprises a grammar slot and three indices ranging over $0 \dots m$ so we need a four dimensional array of such pointers. This ensures the cubic *time* complexity of the algorithm but, as *SPPFindex* is always of cubic size, it implies cubic *space* for all cases, independent of the size of the SPPF itself. This is impractical for strings of lengths in the tens of thousands. Even the GSS size is always quadratic in the length of the input. For real languages the level of ambiguity is low and the size of the SPPF and GSS will be much closer to linear in the size of the input.

A straightforward engineering solution is to not allocate full arrays but instead to pre-allocate one dimension and then allocate subsidiary vectors on demand. For a two-dimensional array of, say, integers we initially allocate only an array P of pointers to one dimensional arrays of integers. When accessing element $[i, j]$ we first check to see whether $P[i]$ is NULL. If it is, then we allocate an array corresponding to the missing row, before accessing element $P[i][j]$. In worst-case this does not save space, but when we have SPPF's which are much less than cubic sized, significant savings can be made whilst maintaining the unit-time lookup property. For a full discussion of this approach see [10]. The results reported in the next section use this technique to reduce the size of `SPPFindex`.

6.4 Performance whilst parsing a highly ambiguous language

The following grammar, Γ_2,

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

is highly ambiguous; standard GLR parsers for Γ_2 are $O(m^4)$. In [19] we reported that the GLR version of Bison could not parse b^{20} and even our efficient RNGLR algorithm requires 884.141 CPU seconds to parse b^{100}, whilst the binary-forest BRNGLR algorithm requires only 1.232 CPU seconds. The measurements in that paper were made on a 1.6GHz Pentium-M (in full power mode) with 256 MByte of memory running Windows-XP. The code was compiled using the Intel compiler bundled with the Borland 5.01 C++ development suite, optimised for speed.

We have constructed a GLL recogniser and a GLL parser for Γ_2 by hand. On the same system used for experiments in [19] with input string b^{100}, the recogniser completed in 1.182 CPU seconds, and the parser in 3.034 CPU seconds. These times are commensurate with BRNGLR, although a little slower. However, there are subtle effects that we have not yet fully investigated. For

Table 1
GLL data structure statistics for Γ_2

m	GSS nodes	GSS edges	$\|U\|$	$max(\|U_i\|)$	SPPF nonpacked nodes	SPPF packed nodes	SPPF edges
50	247	18,189	31,372	1,699	2,550	61,300	183,850
100	497	73,864	125,247	3,449	10,100	495,100	1,485,200
150	747	167,039	281,622	5,199	22,650	1,676,400	5,029,050
200	997	297,714	500,497	6,949	40,200	3,980,200	11,940,400
250	1,247	465,889	781,872	8,699	62,750	7,781,500	23,344,250
300	1,497	671,564	1,125,747	10,449	90,300	13,455,300	40,365,600
350	1,747	914,739	1,532,122	12,199	122,850	21,376,600	64,129,450
400	1,997	1,195,414	2,000,997	13,949	160,400	31,920,400	95,760,800
450	2,247	1,513,589	2,532,372	15,699	202,950	45,461,700	136,384,650
500	2,497	1,869,264	3,126,247	17,449	250,500	62,375,500	187,126,000

instance, using an experimental EBNF GLL template, a GLL parser for the EBNF grammar Γ_2'

$$S \ ::= \ b \mid S \ S \ [\ S \ ]$$

(which is, perhaps, the 'natural' EBNF version of Γ_2) parses b^{100} in only 0.941 CPU seconds.

To demonstrate the practicality of GLL on even such highly ambiguous grammars, we ran the parser and recogniser for Γ_2 on input strings b^{50}, b^{100}, $\ldots b^{500}$. For convenience we used a faster machine with an Intel Xeon E5450 processor running at 3GHz under openSuse 11.1 (64-bit mode) with 24GByte of physical memory. On this machine, b^{100} is parsed in 0.746 CPU seconds for the fully optimised version — a factor four speed up over the Centrino based system used in the previous experiments.

Figure 6.4 shows the results for standard compilation with `g++` and fully optimised for speed compilation using the `-O3` option. Each data point represents the mean of five separate runs. From top to bottom (slowest to fastest) the curves represent the mean run times for the unoptimised parser, the optimised parser, the unoptimised recogniser and the optimised recogniser. All four curves are cubic to a high level of confidence with R^2 values differing from one only in the fourth decimal place.

The sizes of the GSS and SPPF structures for the GLL parser for Γ_2 are shown in Table 1.

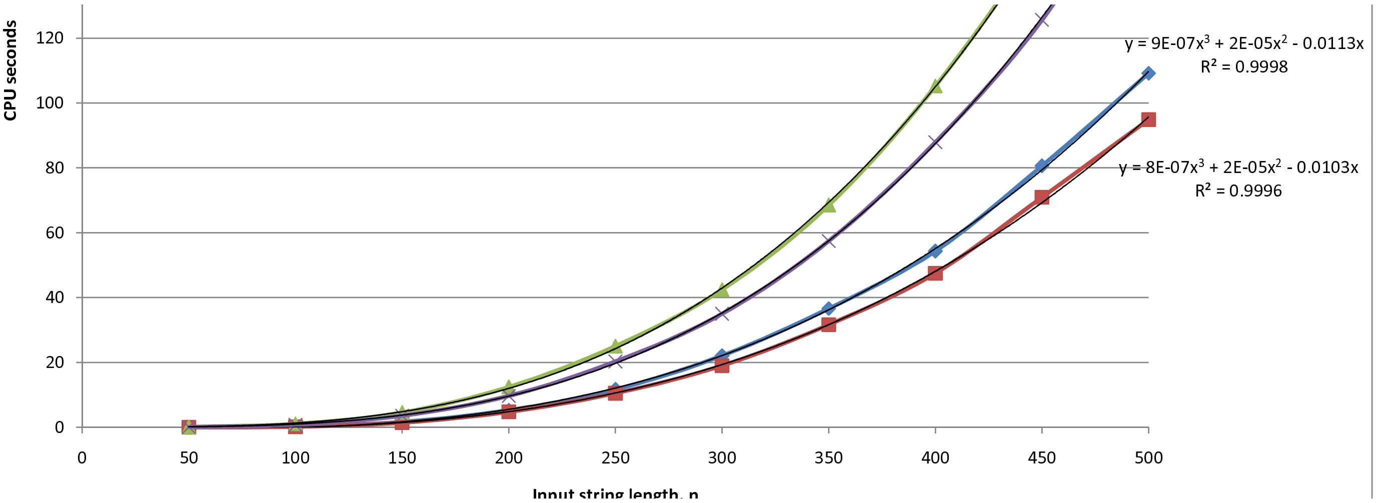


Fig. 1. Cubic performance on Γ_2

6.5 Performance on the standard ANSI-C grammar

For simplicity, the hand written recognisers and parsers used in the previous section do not make full use of potential space saving techniques and so are rather demanding of virtual memory.

As we have discussed in Section 3.1, extending recognition algorithms to parsers is a non-trivial process. For strings consisting of several thousand tokens and grammars such as ANSI-C where the number of grammar slots (which forms the constant of proportionality) is in the hundreds, pre-allocation of cubic space for the SPPF is impractical. We can deal with this, as above, by allocating one or two dimensions of the array on demand. Alternatively, our production GLL parser generator, ART, uses a more flexible approach in which the full array space is divided into a large number, K, of sub-spaces. Enough memory is allocated to hold just one of these sub-spaces, and the element at location v in the full array is then mapped to location (v modulo K). Elements are chained together into lists, just as they would be in a hash table using open addressing; in fact this approach is equivalent to using a hash table except that the hash function guarantees that there is a fixed upper

Table 2
Parsing ANSI-C

Input length	GSS nodes	GSS edges	$\|U\|$	CPU secs
4,291	60,638	219,263	447,962	0.248
26,551	417,204	1,510,486	3,122,639	1.802
36,827	564,437	2,042,843	4,178,346	2.492

bound to the length of each list, and so the worst-case cubic performance is not compromised. For applications in which good average case (as opposed to best possible worst-case) performance is desired, a more conventional hashing function may be used to ensure best possible utilisation of the sub-array lists.

Table 2 shows the results of running a GLL parser for the ANSI-C 89 grammar implemented using this approach. The input strings are the source code for `bool`, a Quine-McCluskey Boolean minimiser (4,291 tokens), the source code for RDP, a recursive descent parser generator (26,551 tokens), and the source code for GTB, our Grammar Tool Box (36,827 tokens). The input has already been tokenised so no lexical analysis needed to be performed. For long strings, we achieve a parse rates of around 14,500 tokens per second. Standard `g++` compiles optimised C++ on the same system at around 8,500 tokens per second. Let us assume for the sake of argument that the runtime of the LALR parser used within `g++` is negligible, then replacing it with our GLL based parser would result in compilation performance of around 5,400 tokens per second. This shows that using generalised parsers based on the GLL algorithm is practical, even for standard programming languages whose syntax is designed for near-deterministic parsing.

7 Conclusions and final remarks

In [17] we showed that GLL recognisers are straightforward to construct and reported on a hand constructed recogniser for the standard ANSI-C grammar: the experimental data reported shows that the GLL algorithm is practical. In this paper we have shown how to construct, in a similarly easy manner, a worst-case cubic GLL parser for any context free grammar. GLL parsers have the desirable property of recursive descent parsers that the parser structure matches the grammar structure.

The GLL algorithm can be modified in a straightforward way so that it does not require parallel process construction in the parts of the grammar that are LL(1). In addition, there are opportunities to implement the data structures in ways that lower the algorithm ‘constants of proportionality’, giving better *average* space and runtime performance particularly in the typical case of a grammar with little or no ambiguity. These implementation issues will be discussed in a future paper.

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] Bryan Ford. Packrat parsing:: simple, powerful, lazy, linear time, functional pearl. *SIGPLAN Not.*, 37(9):36–47, 2002.

[8] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122, 2004.

[9] 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.

[10] Adrian Johnstone and Elizabeth Scott. Modelling GLL parser implementations. In M.van den Brand B.Malloy, S.Staab, editor, *SLE 2010*, volume 6563 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, 2011.

[11] Donald E Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

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

[13] Terence Parr. *ANTLR home page.* `http://www.antlr.org`.

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

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

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

[17] Elizabeth Scott and Adrian Johnstone. GLL parsing. In Jurgen Vinju and Torbjorn Ekman, editors, *LDTA'09 9th Workshop on Language Descriptions, Tools and Applications*, also in Electronic Notes in Theoretical Computer Science, 2009.

[18] Elizabeth Scott and Adrian Johnstone. Recognition is not parsing – SPPF-style parsing from cubic recognisers. *Science of Computer Programming*, 75:55–70, 2010.

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

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

[21] E Ukkonen. Upper bounds on the size of LR(k) parsers. *Inform, Proc. Letters*, 20:99–103, 1985.

[22] 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.

[23] 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.

[24] 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.

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