Structuring the GLL parsing algorithm for performance

Elizabeth Scott and Adrian Johnstone

Abstract

GLL (Generalised LL) parsing algorithms provide a sequentialisation of recursive-descent style parsing that yields efficient, compiled parsers which admit any context free grammar, including left recursive and non-left-factored rules. The resulting parsers retain the 'recursively decent' property that the structure of the parser closely follows the structure of the grammar; as such it is feasible to debug grammars by tracing the corresponding GLL parser using a conventional code debugger.

In this paper we develop two variants of the GLL algorithm called FGLL and RGLL which respectively support (i) efficient parsing of factorised grammars and (ii) parsing using a reduced set of descriptors. Both techniques yield significant speed up on programming language grammars compared to the base GLL algorithm. We also discuss the ordering of descriptor processing and its effects on performance.

1 Introduction

GLL (Generalised LL) parsing algorithms provide a sequentialisation of recursive-descent style parsing that yields efficient, compiled parsers which admit any context free grammar, including left recursive and non-left-factored rules. The resulting parsers retain what could be called the 'recursively decent' property that the structure of the parser closely follows the structure of the grammar; as such it is feasible to debug grammars by tracing the corresponding GLL parser using a conventional code debugger.

The GLL algorithm in [12] can only handle BNF grammars. It may of course be trivially applied to EBNF specifications by translating EBNF constructs to some equivalent BNF, but in general that would entail the creation of extra nonterminals and associated parse-time stack activity. Directly implemented EBNF constructs can increase efficiency; the extension of GLL to directly support so-called FBNF (BNF with additional syntactic left factorisation capability) is relatively straightforward, and can be done in a way which is essentially invisible to the user of a GLL parser generator. We shall describe FGLL, a process which takes a user defined BNF grammar, carries out automatic syntactic left factorisation to produce an FBNF grammar and then generates a form of GLL parser which is defined directly on the FBNF grammar. The important point here is that, from a user point of view, the resulting parser behaves almost exactly as the corresponding BNF GLL parser, but with potentially significant performance improvements.

Another potential improvement to the efficiency of a GLL parser, without any change from the viewpoint of the user of either the parser generator or the parser itself, is to exploit the fundamental property of context free grammars that any derivation from a nonterminal A is a valid derivation from any instance, $X ::= \alpha A \beta$, of A in any grammar rule. Consider, for example, the grammar

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

and the string $abbaa$. A GLL parser will try one rule, say $S ::= AAb$, matching the first A to a, the second A to bba and then failing on the final a. Then trying the other rule, the parser will repeat the match of A to bba. The repeated match is not actually necessary because the result of the first match will already be recorded in the output derivation structure (essentially facilitating a form of memoisation). We can modify the GLL algorithm so that this match is not repeated by changing the nature of the GLL process descriptors. We refer to the modified approach as reduced descriptor GLL (RGLL)[1]. RGLL requires changes to the GLL support functions which build the underlying graph structured stack, but this is invisible to a user.

The RGLL and FGLL modifications have resonances with the DFA states constructed for LR-parsers [2]. All of the grammar positions represented by the so-called LR items in a given DFA state can be reached from the same input sequence, and all of the positions reachable after 'matching' a nonterminal, X say, are combined under a single X-transition to another DFA state. This grouping together of grammar positions allows deterministic LR behaviour in many cases where a simple recursive descent parser would be non-deterministic. For generalised parsers, both GLR [15] and GLL, nondeterminism does not create a problem, but reducing it makes the parsers more efficient. One of the conclusions of this paper will be that, as for LR parsers, most of the efficiency gain arises from the effective syntactic left factorisation of an input BNF grammar. There is, however, further efficiency gain from applying RGLL particularly for left recursive grammar rules.

In this paper we extend the theoretical treatment in [12] to cover FBNF grammars, describe small extensions to the GLL machinery which are needed to build the output shared packed parse forest derivation representations and we give GLL parser generation templates for FBNF grammars. We describe an automatic transformation which syntactically left factors grammars, and give a mechanism by which GLL parsers can return derivations with respect to the original, unfactored, grammar whilst performing parses using the factored grammars. For RGLL we give the modified parse stack support functions and give a discussion on the impact that this has on parse stack construction. We also discuss the impact of GLL descriptor processing order on the efficiency of the parsers, showing that the processing order can dramatically change the number of contingent actions which must be carried out. If these actions are not implemented carefully they can be much less efficient than their non-contingent alternatives, and then descriptor processing order will impact on GLL performance.

We conclude with some experimental analysis of the advantages and trade-offs of both the FGLL and RGLL approaches for conventional programming language grammars; the motivation is performance: for strings parsed using the ANSI-C grammar the throughput more than doubles.

2 Background - GLL parsing

The core data structures for a GLL parser include: a Graph Structured Stack (GSS) which embeds the 'parse function' call stacks associated with calling instances of nonterminals; a Shared Packed Parse Forest (SPPF) which encodes the (potentially infinite) set of derivations; a set of process descriptors which encode parser configurations and allow the parser to deal efficiently with nondeterminism; a set of pending descriptors which have not yet been processed; and book keeping to track which stack nodes have been processed as part of a 'function call return' (pop). In this section we shall introduce the grammar definitions and notations that we shall use and discuss the GLL approach in a way which allows us to extend it to FGLL and RGLL. An example GLL parser is given in Section 2.4.

A *BNF grammar* consists of a set $\mathbf{T}$ of terminals, a set $\mathbf{N}$ of nonterminals disjoint from $\mathbf{T}$,

[1] A modification of the GLL algorithm which exploits the CFG property by constructing a different form of graph structured stack was originally proposed by Alex ten Brink, see Section 4.

a start symbol $S \in \mathbf{N}$, and a set of grammar rules $X ::= \alpha_1 \mid \ldots \mid \alpha_t$, one for each nonterminal $X \in \mathbf{N}$, where each α_k, $1 \leq k \leq t$, is a string over the alphabet $\mathbf{T} \cup \mathbf{N}$. We refer to the α_k as the *production alternates*, or just *alternates*, of X, and to $X ::= \alpha_k$ as a *production rule*, or just a *production*, and α_k may be the empty string, ϵ.

A *derivation step* is an expansion $\gamma Y \beta \Rightarrow \gamma \alpha \beta$ where $\gamma, \beta \in (\mathbf{T} \cup \mathbf{N})^*$ and α is an alternate of Y. A *derivation* of τ from σ is a sequence $\sigma \Rightarrow \beta_1 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \tau$, also written $\sigma \overset{*}{\Rightarrow} \tau$. A *derivation tree* is a graphical representation of a derivation. The root node is labelled with the start symbol, leaf nodes are labelled with a terminal or ϵ. An interior node is labelled with a nonterminal, X say, and its children are labelled with the symbols of an alternate of X.

As is usual for recursive descent parsers, GLL parsers use one symbol lookahead to reduce the number of potential options at each step of a parse. Thus we need the standard FIRST and FOLLOW sets. If γ is a string in $(\mathbf{N} \cup \mathbf{T})^*$

$$first_T(\gamma) = \{a \in \mathbf{T} \mid \gamma \overset{*}{\Rightarrow} a\beta, \text{ for some } \beta\}$$

$$first(\gamma) = \begin{cases} first_T(\gamma) \cup \{\epsilon\} & \text{if } \gamma \overset{*}{\Rightarrow} \epsilon \\ first_T(\gamma) & \text{otherwise} \end{cases}$$

For $A \in \mathbf{N}$, where S is the grammar start symbol and \$ is the end-of-string symbol:

$$follow_T(A) = \{a \in \mathbf{T} \mid S \overset{*}{\Rightarrow} \alpha A a \beta, \text{ for some } \alpha, \beta\}$$

$$follow(A) = \begin{cases} follow_T(A) \cup \{\ \$\} & \text{if } S \overset{*}{\Rightarrow} \alpha A \\ follow_T(A) & \text{otherwise} \end{cases}$$

We say that A is *left recursive* if $A \overset{*}{\Rightarrow} \gamma \Rightarrow A\delta$ for some γ, δ.

At each step in its execution a GLL parser tests the current input symbol, x, against the current grammar position and terminates the execution thread if there is no derivation from this point that begins with x. Formally, we define a function $testSelect(a, X, \rho)$, where a is a terminal or the end-of-string symbol, X is a nonterminal and $\rho \in (\mathbf{N} \cup \mathbf{T})^*$.

```
$testSelect(a, X, \rho)$ {
  **if** (($a \in$ FIRST($\rho$)) or ($\epsilon \in$ FIRST($\rho$) and $a \in$ FOLLOW($X$)) { return true }
  **else** { return false } }
```

2.1 SPPFs and grammar slots

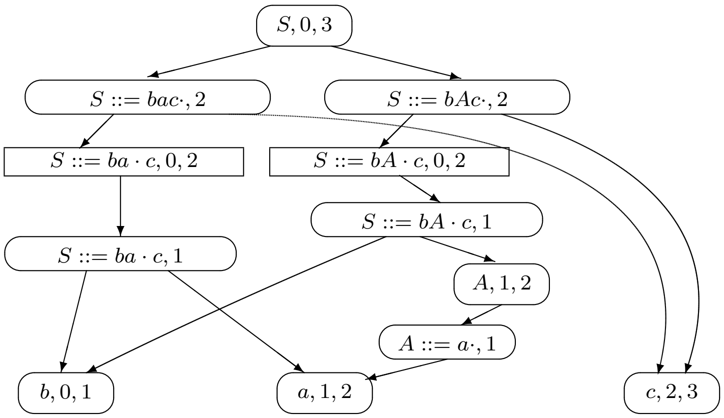
A *shared packed parse forest*, SPPF, is a representation of all of the derivation trees of a given string $a_1 \ldots a_n$. Symbol nodes are labelled with a terminal or nonterminal, x, and the *extent*, (j, i), of the substring generated by that symbol, so $x \overset{*}{\Rightarrow} a_{j+1} \ldots a_i$. Extents are included to ensure that a symbol node is uniquely identified by its label. If the grammar is ambiguous a node can have several families of children, corresponding to different parses of the string. Each family is grouped together under a *packed node*. In order to ensure that the parser has worst-case cubic runtime, the SPPFs are binarised by introducing *intermediate nodes*. (A discussion of the need for SPPF binarisation to achieve cubic general parsing can be found in [13].) For ambiguous grammars intermediate nodes may have several families of children, also grouped together under packed nodes.

A *grammar slot* is a position before or after a terminal or nonterminal on the right hand side of a production rule. We use a 'dot' to indicate a grammar slot. Thus $A ::= \mu \cdot x\nu$ denotes the slot immediately before x. (These slots are called items in LR parsing but we need slots in FBNF grammars as well, and we use the term slot to avoid concerns over the definition of FBNF LR items.) We also need a special slot, L_0, to label the main algorithm loop. We think of this as corresponding to the end of an augmented grammar start rule $S' ::= S\cdot$.

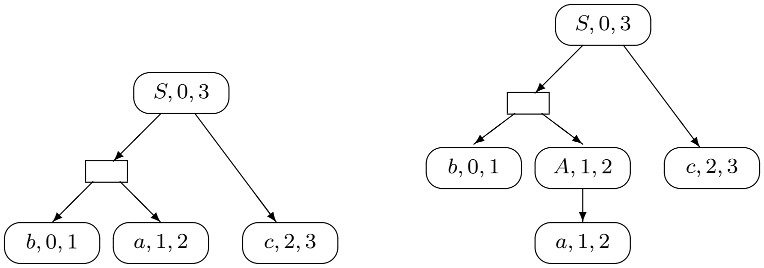
Packed nodes are labelled with a grammar slot and an integer, and intermediate nodes are labelled with a slot and two integer *extents*. For a BNF grammar, the label of a packed node is of the form $(X ::= \alpha x \cdot \beta, k)$, its right child will be a node labelled (x, k, i) and its left child, if it exists, will be a node labelled $(X ::= \alpha \cdot x\beta, j, k)$, or (y, j, k) if $y = \alpha$ has length 1. For example, for the grammar

$$S ::= \; b \; a \; c \mid b \; a \; a \mid b \; A \; c \qquad A ::= a$$

the following is the SPPF for the string *bac*. The rectangular nodes are the intermediate nodes, and the nodes with one integer in their label are the packed nodes.



The two packed node children of the root node $(S, 0, 3)$ correspond to two derivations, one using each of the rules which label the packed nodes. The SPPF embeds the two (binarised) derivation trees below



For a production of length zero or one, an SPPF node will have only one child. Rather than writing special cases of various functions, we use a special 'dummy' node, denoted by Δ, for the missing child. By convention Δ will always be the left child. Dummy nodes will only be explicitly used in the algorithms, for clarity they will be omitted from the displayed graphs.

2.2 Slot notations

GLL parsers are defined via a specification which generates an algorithm from a grammar. The specification takes the form of a set of 'templates' and a parsing algorithm is constructed by substituting actual grammar symbols, alternates and sets of terminals into the templates. In the FGLL parser specification we will use subexpressions of the production alternates and will need to know the position of that subexpression in the production to identify slots. Thus it is convenient to use instance annotations to uniquely define each terminal, nonterminal and ϵ instance in each production rule. These instances essentially correspond to slots. Each symbol instance on the right hand side of a grammar rule is given an unique instance number, which we write as a superscript. We call $x^{(j)}$ a symbol (nonterminal, terminal or ϵ) instance and we say that x is the underlying symbol of the instance $x^{(j)}$. We call an expression, r, whose elements are symbol instances an expression instance, and we write $exp(r)$ for the underlying expression which has the instance subscripts removed.

We let $E_{x^{(j)}}$ denote the slot immediately to the right of a symbol instance $x^{(j)}$, so $E_{x^{(j)}}$ is of the form $Y ::= \alpha x \cdot \delta$. We let $lhs(x^{(j)})$ denote the nonterminal on the left hand side of the production rule in which the instance $x^{(j)}$ occurs.

The functions that build the SPPF take a grammar slot, L, as a parameter. The nodes constructed are labelled with slots related to L; the specific construction depends on the type of slot. The following notation is used for the required properties. We say that L is *eoR*, end-of-rule, if L is the end of a production. We say that L is *fiR*, first-in-rule, if L is $X ::= x.y\tau$ where $x, y \in \mathbf{T} \cup \mathbf{N}$ and if $x \overset{*}{\Rightarrow} \epsilon$ then $x \neq y$ (see the note at the end of Section 2.4 for a discussion of this subtle condition). Finally, lhs_L denotes the nonterminal on the left hand side of L, so $lhs\_E_{x^{(j)}} = lhs(x^{(j)})$.

2.3 GLL parsers

In this section we give an overview of the GLL parser format for BNF grammars. A reader who is not familiar with GLL can find a motivational discussion in [11], an engineering oriented description in [4] and a description of the GLL parser for a BNF grammar in [12]. The specification is slightly different from that given in [12] because we need a different formulation to allow extension to FBNF grammars. However, the actual parsers generated using this specification will be the same as those generated from the specification in [12].

In a GLL parser the parse functions associated with the related recursive descent parser are replaced with algorithm line labels, goto statements and an explicit stack. For each nonterminal A there is a labelled block of code corresponding to each alternate of A and a return label for each position immediately after each nonterminal instance in the context free grammar.

When executing, a GLL parser is essentially traversing the grammar and the input string, and to this end it employs three variables, c_U which holds the current stack top (a GSS node), c_I which holds the current input position and c_N which holds the current SPPF node. When a GLL parser is created for a non-LL(1) grammar, there can be many traversal paths for a given input string. These paths are all explored, and the exploration is managed with the use of *process descriptors*, 4-tuples of the form (L, u, i, w) where L is a grammar slot, u is a GSS node, i is a position in the input string and w is an SPPF node. When a descriptor is created the values of c_U, c_I and c_N are recorded in the descriptor and when a descriptor is processed, these variables are reset using the values in the descriptor. The outer loop of a GLL parser, labelled L_0, selects the next descriptor for processing and processing of a descriptor is terminated with a jump back to L_0.

Each traversal has its own associated stack, and these multiple stacks are combined into a graph structured stack. The stack elements are nodes of the graph and there is a directed edge from node u to node v if u is immediately above v on a stack. The edges of the GSS are labelled with an SPPF node or the dummy node. This edge label will be the left child of the SPPF node constructed when the associated subparse is complete. A node, u, is labelled with a grammar slot and an integer, (L, j), where L is the algorithm line label to be returned to when the stack element is popped, and j is the current input position when (L, j) is created. We call j the *level* of u and write *level_u* to denote this integer.

The function *add*() creates descriptors and ensures that the same descriptor is not created more than once. The function *create*() pushes return labels onto the stack embedded in the GSS and the function *pop*() takes a GSS node $u = (L, j)$ and 'pops' it: for each edge (u, v) in the GSS, i.e. for each stack with u at the top, *pop*() creates a descriptor with code label L and stack node v.

It is possible for new edges to be added to a GSS node u after a *pop*() action has been applied. Thus, when $pop(u, i, z)$ is called, the element (u, z) is stored in a set $\mathcal{P}$. If later a new edge is added to u, by the function *create*(), then the set $\mathcal{P}$ is inspected and any earlier pop actions associated with u will be applied down the new edge by *create*(). We refer to this as contingent pop application and to the pops performed by *create*() as *contingent pop actions.*

The functions $getNodeE(i)$ and $getNodeT(a, i)$ construct (if necessary) and return SPPF nodes labelled (ϵ, i, i) and $(a, i, i + 1)$, respectively. The other symbol and intermediate SPPF nodes are constructed at the same time as their first packed node child by the function $getNode()$.

The functions *add*(), *create*(), *pop*() and $getNode()$ are formally described in Section 2.5, and the formal structure of a GLL parser in defined Section 2.6. But first we give an example GLL parser. In this example, and throughout, we use the following notation:

m	integer: length of the input string
I	integer array of size $m + 1$: input string with $I[m] = \$$
c_I	integer: input pointer
GSS	weighted directed graph: nodes are labelled with elements of the form (L, j) where L is a grammar slot or L_0
c_U	GSS node: current stack top
c_N	SPPF node: current node
c_R	SPPF node: right child
$\mathcal{U}$	current set of descriptors
$\mathcal{R}$	set of descriptors yet to be processed
$\mathcal{P}$	set of GSS node, SPPF pairs which have already been popped
Δ	a dummy node used when there is no left child, omitted from displayed graphs

When the descriptors have all been dealt with, the test for acceptance is made by checking for the existence of the SPPF node labelled with the start symbol and the extent $(0, m)$.

2.4 An example GLL parser

In this section we give an example GLL parser for the grammar

$$S ::= A\ A\ c \qquad A ::= \epsilon$$

followed by a walk-through using a small input string, in order to illustrate the basic GLL mechanisms.

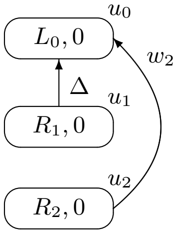
```
    $u_0 := (L_0, 0)$   *(GSS base node)*
    $c_I := 0$   *(current input index)*
    $c_U := u_0$   *(current GSS node)*
    $c_N := \Delta$   *(current SPPF node)*
```

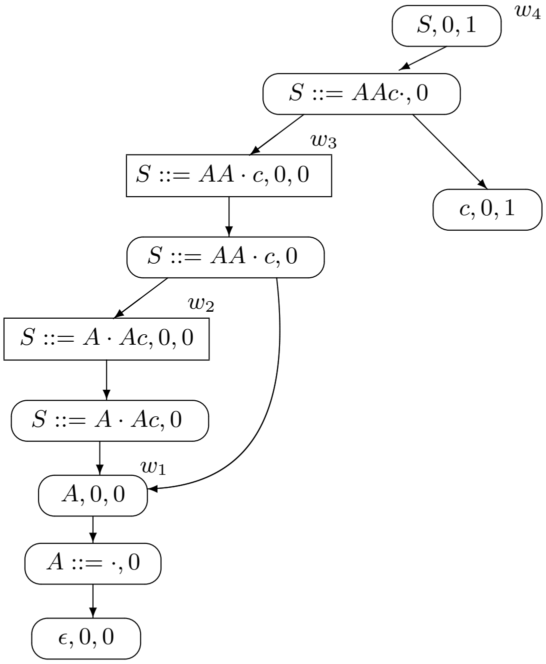
```
        $\mathcal{U} := \emptyset$   *(descriptor set)*
        $\mathcal{R} := \emptyset$   *(descriptors still to be processed)*
        $\mathcal{P} := \emptyset$   *(popped nodes set)*
        **goto** $J_S$

$L_0$:  **if** $(\mathcal{R} \neq \emptyset)$ { remove $(L', k, i, w)$ from $\mathcal{R}$;
                        $c_U := k$;  $c_I := i$; $c_N := w$;  **goto** $L'$ }
         **else** { **if** (there is an SPPF node $(S, 0, m)$) report success
                  **else** report failure }

$J_S$:  **if** $(I[c_I] \in \text{FIRST}(AAc))$  $add(L_{S_1}, c_U, c_I, \Delta)$;   **goto** $L_0$
$L_{S_1}$: $c_U := create(R_1, c_U, c_I, c_N)$;  **goto** $J_A$
 $R_1$: **if** $(I[c_I] \notin \text{FIRST}(Ac))$   **goto** $L_0$
        $c_U := create(R_2, c_U, c_I, c_N)$; **goto** $J_A$
 $R_2$: **if** $(I[c_I] \notin \text{FIRST}(c))$   **goto** $L_0$
         $c_R := getNodeT(c, c_I)$; $c_I := c_I + 1$; $c_N := getNode(S ::= AAc\cdot, c_N, c_R)$
        **if** $(I[c_I] \in \text{FOLLOW}(S))$ $pop(c_U, c_I, c_N)$;   **goto** $L_0$

$J_A$:  **if** $(I[c_I] \in \text{FOLLOW}(A))$  $add(L_{A_1}, c_U, c_I, \Delta)$;   **goto** $L_0$
$L_{A_1}$: $c_R := getNodeE(c_I)$; $c_N := getNode(A ::= \cdot, c_N, c_R)$
        $pop(c_U, c_I, c_N)$;   **goto** $L_0$
```

Example walk-through We illustrate this algorithm using the very simple input $c\$$. Initially, at the line labelled J_S, the test $I[0] = c \in \text{FIRST}(AAc) = \{c\}$ succeeds and so $(L_{S_1}, u_0, 0, \Delta)$ is added to $\mathcal{U}$ and to $\mathcal{R}$, and we have

$$c_U = u_0, \quad c_I = 0, \quad c_N = \Delta, \quad \mathcal{P} = \emptyset, \quad \mathcal{U} = \mathcal{R} = \{(L_{S_1}, u_0, 0, \Delta)\}$$

The algorithm will construct the GSS on the left and the SPPF on the right below.





At line L_0, we remove the element from $\mathcal{R}$, set $L' = L_{S_1}$ and then go to L_{S_1}. Then we create a new GSS node, u_1, labelled $(R_1, 0)$ and an edge labelled $c_N = \Delta$ from u_1 to $c_U = u_0$. Then we set $c_U := u_1$.

At J_A the test $I[0] = c \in \text{FOLLOW}(A) = \{c\}$ succeeds so $(L_{A_1}, u_1, 0, \Delta)$ is added to $\mathcal{U}$ and $\mathcal{R}$, and then removed from $\mathcal{R}$. At L_0:

$$c_U = u_1, \quad c_I = 0, \quad c_N = \Delta, \quad \mathcal{P} = \emptyset, \quad \mathcal{U} = \{(L_{S_1}, u_0, 0, \Delta), (L_{A_1}, u_1, 0, \Delta)\}, \quad \mathcal{R} = \emptyset$$

At L_{A_1}, SPPF nodes labelled $(\epsilon, 0, 0)$, $(A ::= \cdot, 0)$ and $(A, 0, 0)$ are created, and then the node $c_U = u_1$ is popped, with $c_N = w_1$. This results in the creation of an intermediate node, w_2, labelled $(S ::= A \cdot Ac, 0, 0)$ and its associated packed node child, and the descriptor $(R_1, u_0, 0, w_2)$ which is added to $\mathcal{U}$ and $\mathcal{R}$.

$$c_U = u_1, \quad c_I = 0, \quad c_N = w_2, \quad \mathcal{P} = \{(u_1, w_1)\},$$
$$\mathcal{U} = \{(L_{S_1}, u_0, 0, \Delta), (L_{A_1}, u_1, 0, \Delta), (R_1, u_0, 0, w_2)\}, \quad \mathcal{R} = \{(R_1, u_0, 0, w_2)\}$$

Removing $(R_1, u_0, 0, w_2)$ from $\mathcal{R}$, at R_1 we create the GSS node, u_2, labelled $(R_2, 0)$ with an edge to $c_U = u_0$ labelled $c_N = w_2$. At J_A the descriptor $(L_{A_1}, u_2, 0, \Delta)$ is added to $\mathcal{U}$ and $\mathcal{R}$. When this descriptor is processed, at L_0, we get

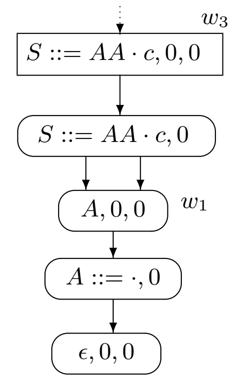
$$c_U = u_2, \quad c_I = 0, \quad c_N = \Delta, \mathcal{P} = \{(u_1, w_1)\},$$
$$\mathcal{U} = \{(L_{S_1}, u_0, 0, \Delta), (L_{A_1}, u_1, 0, \Delta), (R_1, u_0, 0, w_2), (L_{A_1}, u_2, 0, \Delta)\}, \quad \mathcal{R} = \emptyset$$

At L_{A_1}, SPPF nodes labelled $(\epsilon, 0, 0)$, and $(A, 0, 0) = w_1$ are found to already exist, so $c_N := w_1$ and no initial SPPF activity occurs. The node $c_U = u_2$ is then popped, resulting in the creation of an intermediate node, w_3, labelled $(S ::= AA \cdot c, 0, 0)$ with grandchildren w_2 and w_1, and the descriptor $(R_2, u_0, 0, w_3)$.

$$\mathcal{P} = \{(u_1, w_1), (u_2, w_1)\}, \quad \mathcal{R} = \{(R_2, u_0, 0, w_3)\}$$
$$\mathcal{U} = \{(L_{S_1}, u_0, 0, \Delta), (L_{A_1}, u_1, 0, \Delta), (R_1, u_0, 0, w_2), (L_{A_1}, u_2, 0, \Delta), (R_2, u_0, 0, w_3)\}$$

Removing the descriptor from $\mathcal{R}$ we have $c_N = u_0$ and $c_N = w_3$, and at R_2 we create the SPPF nodes $(c, 0, 1)$, $(S ::= AAc\cdot, 0)$ and $(S, 0, 1) = w_4$. When u_0 is popped no action is taken as it has no out edges, so at L_0, $\mathcal{R}$ is empty. The algorithm reports success as a result of the SPPF node w_4.

Note, this simple example does not trigger the application of the elements of the contingent pop set $\mathcal{P}$, but it does illustrate the subtlety of the SPPF intermediate node creation. The property *fiR* stops the creation of a redundant intermediate node above a leftmost child. However, SPPFs have the property that symbol nodes are uniquely defined by their labels, and this ultimately underpins the proof that an SPPF has at most cubic size. If we allowed the *fiR* property to hold for the slot $S ::= A \cdot Ac$ then the corresponding intermediate node would be supressed, resulting in a structure which is a multigraph and not a graph.



2.5 GSS and SPPF constructing functions

For FBNF parsers, the functions *add*(), *pop*() and *create*() are the same as those for the BNF parser [12]. But *getNode*() needs to be modified for FGLL, and all the functions need to be modified for RGLL, so we give the BNF versions here for later reference.

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

$pop(u, i, z)$ {
      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 $getNode(L, w, z)$
              $add(L, v, i, y)$ } }

$create(L, u, i, w)$ {
   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 $getNode(L, w, z)$
               $add(L, u, h, y)$) where $z$ has right extent $h$ } }
   return $v$ }

$getNode(L, w, z)$ {          // Symbol or intermediate node
    if ($L$ is fiR) { return $z$ }
    else {
       suppose that $w$ has label ($q, k, i$)
       if $w = \Delta$ let $j := k$  else suppose that $w$ has label ($s, j, k$)
       if ($L$ is eoR)  set  $t := lhs\_L$   else  set $t := L$
       if there does not exist an SPPF node $y$ labelled ($t, j, i$) create one
       if $y$ does not have a child labelled ($L, k$) {
                   create one with right child $z$ and, if $w \neq \Delta$, left child $w$ }
       return $y$ } }
```

2.6 GLL BNF parser specification

For each nonterminal X in the grammar there is a section, *code*(X), of the algorithm. This section is labelled J_X and is built using functions *code*(r), where r is a string of nonterminal, terminal and ϵ instances. In the following specification the notation we use for the variables is the same as that used in Section 2.3.

Outer level structure of a GLL parser

We suppose that the nonterminals of a grammar Γ are $A, \ldots, X$, and that S is the start nonterminal. Then the GLL algorithm for Γ is specified by:

```
     set $m$ to be the length of the input string
     read the input into an array $I$ and set $I[m] := \$$
     create GSS node $u_0 = (L_0, 0)$
     $c_U := u_0$; $c_N := \Delta$; $c_I := 0$; $\mathcal{U} := \emptyset$; $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$
     goto $J_S$
$L_0$: if $\mathcal{R} \neq \emptyset$ { 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 }

$J_A$: code$(A)$
     ...
$J_X$: code$(X)$
```

The *code*() templates for a BNF grammar

We write templates *codeNT*$(x^{(j)})$ for instances $x^{(j)}$ of each symbol $x \in \mathbf{T} \cup \mathbf{N} \cup \{\epsilon\}$ and templates *code*(r) for strings, r, of such symbols. Then we give templates for each grammar rule.

For a terminal instance $a^{(j)}$ with underlying terminal a

$$\begin{array}{rcl} codeTN(a^{(j)}) & = & c_R := getNodeT(a, c_I); \quad c_I := c_I + 1 \\ & & c_N := getNode(E_{a^{(j)}}, c_N, c_R) \end{array}$$

For a nonterminal instance $Y^{(j)}$ with underlying nonterminal Y

$$\begin{array}{rcl} codeTN(Y^{(j)}) & = & \quad c_U := create(E_{Y^{(j)}}, c_U, c_I, c_N); \ \mathbf{goto}\ J_Y \\ & & E_{Y^{(j)}} : \end{array}$$

For instances $\epsilon^{(j)}$ we need both symbol and string versions

$$codeTN(\epsilon^{(j)}) \quad = \quad c_R := getNodeE(c_I); \quad c_N := getNode(E_{\epsilon^{(j)}}, c_N, c_R)$$

$$\begin{array}{rcl} code(\epsilon^{(j)}) & = & codeTN(\epsilon^{(j)}) \\ & & pop(c_U, c_I, c_N); \quad \mathbf{goto}\ L_0 \end{array}$$

For a nonempty string of terminal and nonterminal instances, $g_1 \dots g_d$, where X is $lhs(g_1)$

$$\begin{array}{rcl}
code(g_1 \dots g_d) & = & codeTN(g_1) \\
 & & \mathbf{if}(testSelect(I[c_I], X, exp(g_2 \dots g_d))\ is\ false)\ \ \mathbf{goto}\ L_0 \\
 & & codeTN(g_2) \\
 & & \dots \\
 & & \mathbf{if}(testSelect(I[c_I], X, exp(g_d))\ is\ false)\ \mathbf{goto}\ L_0 \\
 & & codeTN(g_d) \\
 & & \mathbf{if}(I[c_I] \in follow(X))\ pop(c_U, c_I, c_N);\ \ \mathbf{goto}\ L_0
\end{array}$$

For a grammar rule of the form $X ::= \alpha_1 \mid \dots \mid \alpha_p$, let r_i denote the instance version of α_i, $1 \leq i \leq p$. We define $code(X)$ as follows.

$$\begin{array}{rcl}
code(X) & = & \mathbf{if}(testSelect(I[c_I], X, \alpha_1))\ \{\ add(L_{r_1}, c_U, c_I, \Delta)\ \} \\
 & & \dots \\
 & & \mathbf{if}(testSelect(I[c_I], X, \alpha_p))\ \{\ add(L_{r_p}, c_U, c_I, \Delta)\ \} \\
 & & \mathbf{goto}\ L_0 \\
 & L_{r_1}: & code(r_1) \\
 & & \dots \\
 & L_{r_p}: & code(r_p)
\end{array}$$

3 FGLL parsing

In this section we define syntactically left factorised BNF (FBNF) grammars and corresponding GLL-style parsers. An FGLL parser is generated for a BNF grammar Γ by syntactically left factorising Γ to obtain Γ', and then generating a GLL-style parser from Γ' using the specification defined below. The syntactic left factorisation procedure is defined in Section 3.5, the issue of derivations with respect to the original grammar is addressed in Section 3.6 and in Section 5 the performance of the FGLL parser is compared to corresponding BNF GLL and reduced descriptor GLL parsers.

3.1 FBNF grammars

For an alphabet $\mathcal{A}$ a syntactically left factorised slf-expression over $\mathcal{A}$ is defined as follows

- Any string in $\mathcal{A}^*$ is an slf-expression.
- If γ is a nonempty string in $\mathcal{A}^*$ and $\rho_1, \dots, \rho_k$ are slf-expressions, then $\gamma(\rho_1 | \dots | \rho_k)$ is an slf-expression.

Slf-expressions are regular expressions and hence denote sets of strings in the usual manner. We write $\delta \in \rho$ to indicate that $\delta \in \mathcal{A}^*$ is a string in the pattern of the regular expression ρ.

An *FBNF grammar* is a modification of a BNF grammar in which the production alternates, α_i, are slf-expressions over $\mathbf{T} \cup \mathbf{N}$. An FBNF grammar *derivation step* is of the form $\gamma Y \beta \Rightarrow \gamma \delta \beta$, where α is a production alternate of Y and $\delta \in \alpha$.

We extend the definition of FIRST sets to slf-expressions as follows. For a string γ

$$first(\gamma(\rho_1 | \dots | \rho_k)) = \begin{cases} first_T(\gamma) \cup first(\rho_1) \cup \dots \cup first(\rho_k) & \text{if } \gamma \overset{*}{\Rightarrow} \epsilon \\ first(\gamma) & \text{otherwise} \end{cases}$$

We need a slot type for labelling lines of the parsing algorithm associated with alternates inside slf-expressions. As for BNF grammars, we give each terminal, nonterminal and ϵ instance on the right hand sides of productions of an FBNF grammar a unique instance number. Slf-expressions, r, whose elements are symbol instances will be referred to as slf-expression instances, and $exp(r)$ will denote the underlying slf-expression obtained by removing the instance superscripts. For each slf-expression instance r, which begins immediately after an opening parenthesis, (, or alternation symbol, |, we write L_r for the slot immediately to the left of the first symbol of $exp(r)$. So these slots are of the form $Y ::= \alpha(\ \cdot\mu_1 \mid \ldots \mid \mu_d\ )$ or $Y ::= \alpha(\ \mu_1 \mid \ldots \mid \gamma(\ \nu_1 \mid \ldots \mid \cdot\nu_f)) \mid \ldots \mid \mu_d\ )\ )$ etc.

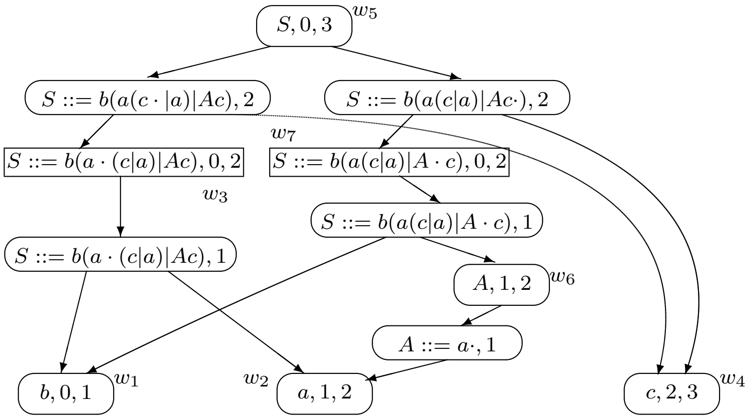
We also need to identify slots at the end of alternates for SPPF construction. We say that L is *eoA*, end-of-alternate, if L is a position immediately before an alternation symbol | or a closing parenthesis).

3.2 SPPFs for FBNF grammars

For FBNF, to ensure the uniqueness of packed node labels for a given parent, for productions $X ::= \rho$ where ρ contains bracketed subexpressions, the packed nodes under nodes for X will be labelled with slots at the end of alternates rather than the end of the production. For example, for the grammar

$$S ::= b\ (\ a\ (\ c \mid a\ ) \mid A\ c\ ) \qquad A ::= a$$

and input `bac` there will be two packed nodes $(S ::= b(a(c|a)|Ac\cdot), 2)$ and $(S ::= b(a(c\cdot|a)|Ac), 2)$, under the node $(S, 0, 3)$.


$S, 0, 3$
w_5
$S ::= b(a(c\cdot|a)|Ac), 2$
$S ::= b(a(c|a)|Ac\cdot), 2$
w_7
$S ::= b(a\cdot(c|a)|Ac), 0, 2$
$S ::= b(a(c|a)|A\cdot c), 0, 2$
w_3
$S ::= b(a(c|a)|A\cdot c), 1$
$S ::= b(a\cdot(c|a)|Ac), 1$
$A, 1, 2$
w_6
$A ::= a\cdot, 1$
$b, 0, 1$
w_1
w_2
$a, 1, 2$
$c, 2, 3$
w_4

Note that the structure of the derivation trees will not reflect parenthesisation in the grammar, although in some cases the FBNF SPPF will have more node sharing than the SPPF for the corresponding BNF grammar.

For an FGLL parser the GSS and SPPF construction functions described in Section 2.5 are unchanged except that the condition **if**$(L\ is\ eoR)$ in $getNode()$ is modified to **if**$(L\ is\ eoR\ or\ eoA)$.

3.3 GLL parsers for FBNF grammars

Apart from the minor change to $getNode()$ mentioned in the previous section, the GSS and SPPF construction functions for GLL FBNF parsers are the same as those for BNF parsers.

The specification for the parsers themselves is an extension of the BNF specification. We just add one further template for slf-expression instances which contain parentheses.

For a slf-expression instance $g_1 \dots g_d($ r_1 $|$ $\dots$ $|$ r_k $)$, where $d, k \geq 1$, $g_1, \dots, g_d$ are terminal and nonterminal instances, and X is $lhs(g_1)$

```
        code(g_1 ... g_d(  r_1 | ... | r_k )) =

                      $codeTN(g_1)$
                      $\textbf{if}(testSelect(I[c_I], X, exp(g_2 \dots g_d))\ is\ false)$  $\textbf{goto}\ L_0$
                      $codeTN(g_2)$
                      $\dots$
                      $\textbf{if}(testSelect(I[c_I], X, exp(g_d))\ is\ false)$ $\textbf{goto}\ L_0$
                      $codeTN(g_d)$
                      $\textbf{if}(testSelect(I[c_I], X, exp(r_1)))\ \{\ add(L_{r_1}, c_U, c_I, c_N)\ \}$
                      $\dots$
                      $\textbf{if}(testSelect(I[c_I], X, exp(r_k)))\ \{\ add(L_{r_d}, c_U, c_I, c_N)\ \}$
                      $\textbf{goto}\ L_0$
                 $L_{r_1}$ : $code(r_1)$
                      $\dots$
                 $L_{r_k}$ : $code(r_k)$
```

3.4 An example FGLL parser

Consider again the grammar:

$$S ::= b\ (\ a\ (\ c\ |\ a\ )\ |\ A\ c\ ) \qquad A ::= a$$

and instance the symbols as follows

$$S ::= b^{(1)}\ (\ a^{(1)}\ (\ c^{(1)}\ |\ a^{(2)}\ )\ |\ A^{(1)}\ c^{(2)}\ ) \qquad A ::= a^{(3)}$$

We let L_{S_1}, L_{S_2}, L_{S_3}, L_{S_4}, L_{S_5}, and L_{A_1} denote the slots immediately before $b^{(1)}$, $a^{(1)}$, $c^{(1)}$, $a^{(2)}$, $A^{(1)}$, and $a^{(3)}$, respectively.

To make the parser easier to read we shall instantiate the actual slots for the general $E_{x^{(j)}}$ denoted slots of the specification templates, and we shall let R_1 denote the slot $E_{A^{(1)}}$, the slot immediately after $A^{(1)}$.

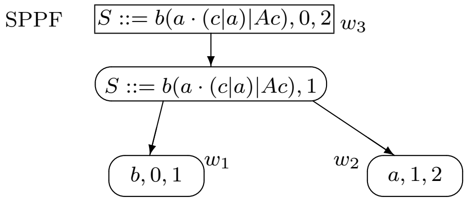
The GLL parser for this FBNF grammar is then as follows.

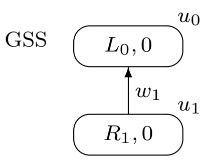
```
      $u_0 := (L_0, 0)$; $c_U := u_0$; $c_N := \Delta$; $c_I := 0$; $\mathcal{U} := \emptyset$; $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$
      goto $J_S$
$L_0$:  if ($\mathcal{R} \neq \emptyset$) { remove $(L, u, i, w)$ from $\mathcal{R}$
                    $c_U := u$; $c_I := i$; $c_N := w$;  goto $L$ }
      else  if (there exists and SPPF node labelled $(S, 0, m)$) report success
            else report failure

$J_S$:  if ($testSelect(I[c_I], S, b(a(c|a)|Ac))$  $add(L_{S_1}, c_U, c_I, \Delta)$
      goto $L_0$
$L_{S_1}$:  $c_R := getNodeT(b, c_I)$; $c_I := c_I + 1$
       $c_N := getNode(S ::= b \cdot (a(c|a)|Ac), c_N, c_R)$
       if ($testSelect(I[c_I], S, a(c|a)))$  $add(L_{S_2}, c_U, c_I, c_N)$
       if ($testSelect(I[c_I], S, Ac))$  $add(L_{S_5}, c_U, c_I, c_N)$
      goto $L_0$
$L_{S_2}$: $c_R := getNodeT(a, c_I)$; $c_I := c_I + 1$
      $c_N := getNode(S ::= b(a \cdot (c|a)|Ac), c_N, c_R)$
      if $(testSelect(I[c_I], S, c))$  $add(L_{S_3}, c_U, c_I, c_N)$
      if $(testSelect(I[c_I], S, a))$  $add(L_{S_4}, c_U, c_I, c_N)$
      goto $L_0$
$L_{S_3}$: $c_R := getNodeT(c, c_I)$; $c_I := c_I + 1$
     $c_N := getNode(S ::= b(a(c \cdot |a)|Ac), c_N, c_R)$
     if$(I[c_I] \in follow(S))$ $pop(c_U, c_I, c_N)$;  goto $L_0$
$L_{S_4}$: $c_R := getNodeT(a, c_I)$; $c_I := c_I + 1$
     $c_N := getNode(S ::= b(a(c|a\cdot)|Ac), c_N, c_R)$
     if$(I[c_I] \in follow(S))$ $pop(c_U, c_I, c_N)$;  goto $L_0$
$L_{S_5}$: $c_U := create(R_1, c_U, c_I, c_N)$;  goto $J_A$
$R_1$: if $(testSelect(I[c_I], S, c)$ is false)   goto $L_0$
     $c_R := getNodeT(c, c_I)$; $c_I := c_I + 1$
     $c_N := getNode(S ::= b(a(c|a)|Ac\cdot), c_N, c_R)$
     if$(I[c_I] \in follow(S))$ $pop(c_U, c_I, c_N)$;  goto $L_0$

$J_A$: if $(testSelect(I[c_I], A, a))$ $add(L_{A_1}, c_U, c_I, \Delta)$;  goto $L_0$
$L_{A_1}$: $c_R := getNodeT(a, c_I)$; $c_I := c_I + 1$
     $c_N := getNode(A ::= a\cdot, c_N, c_R)$
     if$(I[c_I] \in follow(A))$ $pop(c_U, c_I, c_N)$;  goto $L_0$
```

We illustrate the algorithm using the string *bac*\$ and constructing the SPPF shown at the end of Section 3.2. The final descriptor set will be

$$\begin{aligned}\mathcal{U} = \{ \quad & (L_{S_1}, u_0, 0, \Delta), (L_{S_2}, u_0, 1, w_1), (L_{S_5}, u_0, 1, w_1), \\ & (L_{S_3}, u_0, 2, w_3), (L_{A_1}, u_1, 1, \Delta), (R_1, u_0, 2, w_7) \ \}\end{aligned}$$

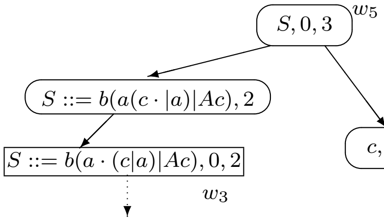
The descriptors will be constructed in the order that they are listed above, and will be removed from $\mathcal{R}$ in insertion (first in, first out) order. The SPPF names $w_1, \ldots, w_7$ refer to the labels on the SPPF in Section 3.2. We initialise the variables and then go to line J_S.
J_S: since $I[0] = b$, $testSelect(I[c_I], S, b(a(c|a)|Ac))$ succeeds so $(L_{S_1}, u_0, 0, \Delta)$ is added to $\mathcal{U}$ and $\mathcal{R}$, and then removed from $\mathcal{R}$ at L_0.
L_{S_1}: the SPPF node w_1 is created and the assignments $c_R := w_1$, $c_I := 1$ are made. Because of the fiR property, $getNodeT()$ returns w_1, so $c_N := w_1$. Since $c_I = 1$ and $I[1] = a$, both $testSelect()$ calls return true, and so $(L_{S_2}, u_0, 1, w_1)$ and $(L_{S_5}, u_0, 1, w_1)$ are added to $\mathcal{U}$ and $\mathcal{R}$.
L_{S_2}: the SPPF node w_2 is created, then assignments $c_R := w_2$, $c_I := 2$, and then $getNodeT()$ returns w_3, so $c_N := w_3$. Since $I[c_I] = I[2] = c$, only the first call to $testSelect()$ returns true, and $(L_{S_3}, u_0, 2, w_3)$ is added to $\mathcal{U}$ and $\mathcal{R}$.


SPPF
$S ::= b(a \cdot (c|a)|Ac), 0, 2$
w_3
$S ::= b(a \cdot (c|a)|Ac), 1$
$b, 0, 1$
w_1
w_2
$a, 1, 2$



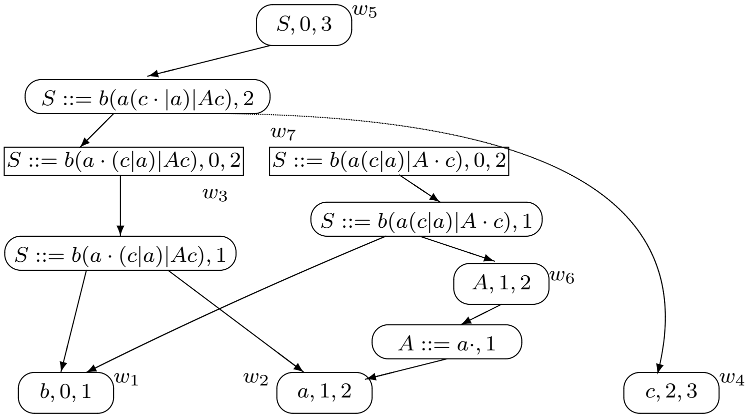
L_{S_5}: $c_I = 1$, $c_U = u_0$, $c_N = w_1$. A GSS node, u_1, labelled $(R_1, 0)$ is created along with an edge labelled w_1 to u_0, then $c_U := u_1$ and we go to J_A.

J_A: $I[1] = a$, so $(L_{A_1}, u_1, 1, \Delta)$ is added to $\mathcal{U}$ and $\mathcal{R}$, and we go to L_0.
L_{S_3}: $c_I = 2$, $c_U = u_0$, $c_N = w_3$. The SPPF node w_4 is created, $c_R := w_4$, $c_I := 3$, and then the SPPF node w_5 is created, and $c_N := w_5$. Since $I[3] = \$$, $pop(u_0, 3, w_5)$ is called, but has no action other than to add (u_0, w_5) to $\mathcal{P}$. So we return to L_0.



$S, 0, 3$
w_5
$S ::= b(a(c \cdot |a)|Ac), 2$
$S ::= b(a \cdot (c|a)|Ac), 0, 2$
w_3
$c, 2, 3$
w_4

L_{A_1}: $c_I = 1$, $c_U = u_1$, $c_N = \Delta$. An SPPF node labelled $(a, 1, 2)$ already exists, so $c_R := w_2$ and $c_I := 2$, and then w_6 is created by $getNode()$, $c_N := w_6$. Then $pop(u_1, 2, w_6)$ creates the SPPF node w_7 and the descriptor $(R_1, u_0, 2, w_7)$, and adds (u_1, w_6) to $\mathcal{P}$.



$S, 0, 3$
w_5
$S ::= b(a(c \cdot |a)|Ac), 2$
w_7
$S ::= b(a \cdot (c|a)|Ac), 0, 2$
$S ::= b(a(c|a)|A \cdot c), 0, 2$
w_3
$S ::= b(a(c|a)|A \cdot c), 1$
$S ::= b(a \cdot (c|a)|Ac), 1$
$A, 1, 2$
w_6
$A ::= a \cdot, 1$
$b, 0, 1$
w_1
w_2
$a, 1, 2$
$c, 2, 3$
w_4

R_1: $c_I = 2$, $c_U = u_0$, $c_N = w_7$. We get $c_R := w_5$ and then $getNode()$ adds a new packed node, with children w_7 and w_4, as a child of the existing node w_5. As w_5 already exists, no new descriptors are created and, at L_0, $\mathcal{R} = \emptyset$ and the algorithm terminates in success.

3.5 Automatic factorisation

Our application focus in this paper is on improving the efficiency of the GLL algorithm on BNF grammars. So, for the FGLL algorithm, the first step is a factorisation from BNF into FBNF. In this section we describe the automatic factorisation process in our ART GLL parser generator [5]. The general principle is to load each nonterminal's productions into a trie data structure,[2] and then to write out the contents of the trie as an FBNF rule.

A *trie* or *prefix tree* stores a set of strings in such a way that common prefixes are shared. The root and leaf nodes of a trie are special *scaffolding* nodes denoted by \$. Each interior node is labelled with an element of a string.

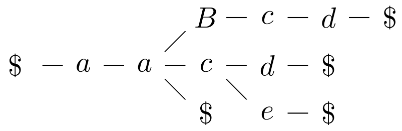
An empty trie comprises a single scaffolding node. We load a string s of length n into the trie by maintaining an index i into the string and a pointer t into the tree. Initially i is 1 and t points to the root of the trie.

```
$loadTrie(s)$ {
  $i := 1$
```

[2] We are grateful to a referee for pointing out that tries are also used for left factoring in [8]

```
  $t$ is set to point to the root of the trie
  while ($i \leq n$) {
     if there exists a child $c$ of $t$ labelled $s[i]$ { advance $t$ to $c$ }
     else { create a child $c$ of $t$ labelled $s[i]$
             advance $t$ to $c$ }
     $i := i + 1$ }
   add a scaffolding node as a child of $t$
}
```

Consider the rule $S ::= a\ a\ B\ c\ d \mid a\ a\ c\ d \mid a\ a\ c\ e \mid a\ a$ which factorises to $S ::= a\ a\ (\ B\ c\ d \mid c\ (\ d \mid e\ ) \mid \epsilon\ )$. If we load the unfactorised productions into a trie, we have:



The scaffolding nodes are represented here as $ nodes.

The trie directly represents the bracketing structure of the left-factored rule, and it is easy to write out the factored rule by traversing the trie. Note that an empty branch occurs whenever a scaffolding node has siblings: empty branches are rendered as epsilon sub-rules.

As noted by a referee, the above algorithm works even for grammars with repeated rules. Repeated rules will result in the same linear subgraph up to the penultimate node, and then will end a separate scaffolding node for each repetition.

3.6 Extracting derivations in the original grammar

An uncomfortable side-effect of any factorisation process is that derivations from the associated parser would normally be in the factorised grammar, not the original grammar. This obscures understanding, and requires some care in the implementation of semantic actions. We expect the internal behaviour of an FGLL parser to be invisible to a user who supplies a BNF grammar to the parser generator, and producing derivations in a modified grammar would certainly not be acceptable.

We should first note that in a generalised parser, semantic actions cannot usually be executed at parse time as a side-effect of the parsing, even for unambiguous grammars; fully general parsers may well explore many avenues which are not part of the eventual derivation. So semantic evaluation will in general be delayed until after a complete SPPF has been constructed. At that point a single derivation can be extracted, according to some disambiguation strategy if necessary, and returned for downstream use. It is this derivation which should be with respect to the original grammar.

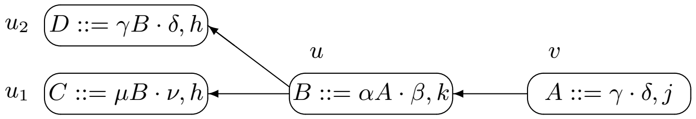
It is a feature of an FGLL built SPPF that it is easy to extract a derivation in the original, unmodified grammar. There is a one-to-one relationship between slots which are *eoA* or *eoR*, as defined in Section 3.1, in the factored grammar, the slots which are *eoR* in the original grammar, and the scaffolding leaf nodes in the trie. For example, $S ::= aa(Bcd|c(d \cdot |e)|\epsilon)$ corresponds to $S ::= aacd\cdot$. This provides a map between the derivation steps in the as-parsed grammar and the equivalent derivation steps in the original grammar.

To facilitate derivation extraction, as each production $X ::= \alpha$ in the original grammar is loaded into the trie we can create an attribute in the final scaffolding node that contains the slot $(X ::= \alpha.)$. We then write into the FGLL parser the rules of the original grammar and a map from slots in the factorised grammar to slots in the original grammar. After parsing is complete, the SPPF can be traversed with any chosen derivation selection specification. As each set of siblings is completed, the map can be checked to see if the rightmost slot corresponds to a factorised production, and if it does the derivation tree construction will use the corresponding sequence of elements from the original production.

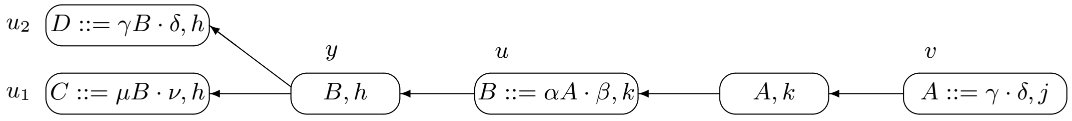
4 Reduced descriptor GLL parsers

The descriptors defined in Section 2.3 contain slightly more information than is actually needed to record a parser configuration. A GSS node $u = (X ::= \alpha A \cdot \beta, k)$ is created at the point of what would be a call to the parse function for A in a recursive descent parser. The current GSS node is included in a descriptor as part of the parser configuration. In fact, any action which is applied to u will eventually also be applied to any GSS node with a label of the form $(Y ::= \gamma A \cdot \delta, k)$. This is because, as described in the introduction, any valid derivation from A at the point $X ::= \alpha A \cdot \beta$ is also a valid derivation at the point $Y ::= \gamma A \cdot \delta$. By modifying the GSS construction functions we can change the descriptors to record just the level k rather than the GSS node (the nonterminal A is identifiable from the label in the descriptor). This reduces the number of descriptors created and processed and also removes some repeated activity associated with A.

Whilst writing his MSc thesis [14], ten Brink made this observation that the runtime efficiency of some GLL recognisers can be improved by removing repeated activity associated with multiple instances of nonterminals. Ten Brink's proposal involves changing the nature of the GSS, producing a different graph whose edges are labelled with nodes of the original GSS and whose nodes are new nodes associated with grammar nonterminals. Although not expressed in this form by ten Brink, it is perhaps easiest to understand the modification as an extension of an original GSS


u_2
$D ::= \gamma B \cdot \delta, h$
u
v
u_1
$C ::= \mu B \cdot \nu, h$
$B ::= \alpha A \cdot \beta, k$
$A ::= \gamma \cdot \delta, j$

to a bipartite graph in which nodes u and v which were adjacent in the original GSS are separated by a new nonterminal node. GSS pop actions are then applied down paths of length 2 between nonterminal nodes.


u_2
$D ::= \gamma B \cdot \delta, h$
y
u
v
u_1
$C ::= \mu B \cdot \nu, h$
B, h
$B ::= \alpha A \cdot \beta, k$
A, k
$A ::= \gamma \cdot \delta, j$

The advantage is that when a GSS node in the original format has two or more edges to nodes at the same level, a pop action down these edges results in a single descriptor $(B ::= \alpha A \cdot \beta, y, i)$ rather than two or more descriptors of the form $(B ::= \alpha A \cdot \beta, u_1, i)$, $(B ::= \alpha A \cdot \beta, u_2, i)$ etc. (Ten Brink focussed only on recognisers so neither the descriptors nor the GSS include SPPF nodes.)

One of the major attractions of the GLL approach is that the algorithm parse traces correspond directly to the set of paths through a nondeterministic recursive descent parser, and the GSS is the combination of the corresponding function call stacks. Hence we do not want to change the nature of the GSS in the manner suggested by ten Brink. However, it is possible to modify the GSS construction functions to achieve a similar efficiency gain without changing the structure of the GSS. The modified construction functions are more complicated than the original GLL ones, and care is needed to ensure that the data structures support constant look-up time, but the implementation of these functions does not need to be visible to someone building or using a GLL parser, whereas the GSS is visible to them. Thus we gain efficiency without disrupting the user view of the algorithm. The visible change is that descriptors contain GSS levels rather than GSS nodes. For this reason we call this *reduced descriptor GLL*, RGLL.

We give the specification of an RGLL parser and, in Section 6, we make comparisons with the corresponding FGLL parsers. Note, the reduced descriptor approach may reduce the number of descriptors and thus some computational overhead, but it does not save any SPPF or GSS construction.

4.1 The RGLL GSS construction functions

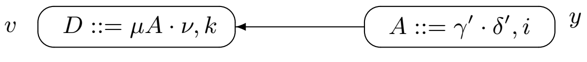
Recall that a GSS node u has a label of the form $(B ::= \alpha A \cdot \beta, k)$ and we treat L_0 as being of the form $S' ::= S \cdot$, so u_0 is labelled $(S' ::= S \cdot, 0)$. We call A the nonterminal of u and B the left hand side of u, and we write *level_u*, *nt_u* and *lhs_u* to denote k, A and B respectively. Motivated by the discussion above, the idea is conceptually to group together GSS nodes which have the same level and the same nonterminal, so u and v are in the same group if *level_u* = *level_v* and *nt_u* = *nt_v*. If *level_u* = *level_v* and *nt_u* = *nt_v* we say that u and v are *pop equivalent*; we can apply a given pop action to all GSS nodes with the same nonterminal and level at the same time.

The focus of our original presentation [12] was to describe GLL as an extension of recursive descent, so that it is easy to understand the basic approach. Recording the current code position, input position and stack top in the descriptors is thus natural. However, as discussed above, we can write instead *reduced descriptors* of the form $(B ::= \alpha A \cdot \beta, k, i, w)$, where k is a GSS level. This reduces the number of descriptors created and processed, and rather than having the global variable c_U hold the current GSS node, it simply holds the current GSS level. (The nonterminal is passed directly into the pop action at the point where it is called.) We also modify the *create*() function so that it doesn't return anything.

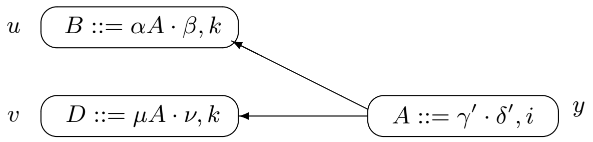
A cost is that it is necessary to represent the GSS in a way which allows the set of all nodes of the form $(B ::= \alpha A \cdot \beta, k)$ for a specified A and k to be found in time proportional to the number of such nodes. The implementation we use in Section 6 has this property.

The set $\mathcal{P}$ is used by *create*() to perform contingent pop actions down new edges. In the reduced descriptor formulation a pop action may have been applied at a node which is pop equivalent to the source node of a new edge. Thus we record pop actions which have been applied to nodes with a given nonterminal and level. The level is the left extent of the SPPF node, z, so $\mathcal{P}$ is a set of elements of the form (A, z).

The other change when using reduced descriptors is that some in-edges are added to GSS nodes on creation. For any GSS node y, if *lhs_y* is A, say, then for each child, v, of y we have *nt_v* = A.

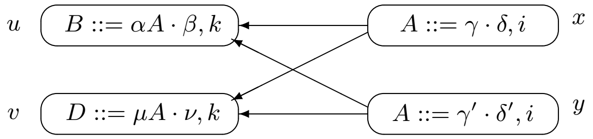


Furthermore, because of the fundamental property of context free grammars discussed above, two pop-equivalent GSS nodes will have the same set of parent nodes once the GSS is complete. Thus if, during the non-reduced descriptor GLL parse, a GSS node u is created with *nt_u* = A and *level_u* = *level_v* then at some point an edge will also be added from y to u. So in an RGLL parse, when a node u is created, if there is a pop-equivalent node v then for all edges into v a corresponding edge into u is immediately created.



u
$B ::= \alpha A \cdot \beta, k$
v
$D ::= \mu A \cdot \nu, k$
$A ::= \gamma' \cdot \delta', i$
y

Also, if a new node x is created as a parent of v (or of u) then it is also immediately made a parent of the pop-equivalent node u (or v).



u
$B ::= \alpha A \cdot \beta, k$
$A ::= \gamma \cdot \delta, i$
x
v
$D ::= \mu A \cdot \nu, k$
$A ::= \gamma' \cdot \delta', i$
y

The GSS construction functions for an RGLL parser are thus defined as follows.

```
$add(L, k, i, w)$ { **if** (($L, k, i, w) \notin \mathcal{U}$ { add $(L, k, i, w)$ to $\mathcal{U}$ and to $\mathcal{R}$ } }

$pop(A, k, i, z)$ {
    **if** $(A, z) \notin \mathcal{P}$ {
        add $(A, z)$ to $\mathcal{P}$
        **for** each GSS node $u = (Y ::= \alpha A \cdot \beta, k)$ {
            **for** each edge $(u, w, v)$ {
                let $y$ be the node returned by $getNode(Y ::= \alpha A \cdot \beta, w, z)$
                $add(Y ::= \alpha A \cdot \beta, level\_v, i, y))$ } } } }

$create(L, k, i, w)$ {
    let $L$ be of the form $B ::= \tau A \cdot \mu$
    **if** there is not already a GSS node labelled $(L, i)$ {
        create a GSS node $v$ labelled $(L, i)$
        **if** there exists a GSS node $v' \neq v$ which is pop equivalent to $v$ {
        let $v' \neq v$ be some node which is pop equivalent to $v$
        **for** all edges $(x, f, v')$ { create an edge $(x, f, v)$ } } }
    let $v$ be the GSS node labelled $(L, i)$
     **for** each GSS node $u$ with label of the form $(Y ::= \alpha B \cdot \beta, k)$ {
         create an edge from $v$ to $u$ labelled $w$ }
     **for** all $(A, z) \in \mathcal{P}$ where the left extent of $z$ is $i$ {
         let $y$ be the node returned by $getNode(L, w, z)$
         $add(L, k, h, y)$ where $z$ has right extent $h$ } } }
```

The SPPF construction functions for an RGLL parser are unchanged, they are the same as those given in Section 2.5.

4.2 RGLL parser specification

By changing the type of c_U from GSS node to integer we can essentially use the BNF GLL specification for RGLL. The outer structure of an RGLL parser is the same as that of the

BNF parser specified in Section 2.6 except that c_U is initialised to 0 rather than to u_0. The templates $code(\epsilon^{(j)})$ and $code(g_1 \dots g_d)$ have $pop(c_u, c_i, c_N)$ replaced with $pop(X, c_u, c_i, c_N)$ where X is $lhs(\epsilon^{(j)})$ or $lhs(g_1)$ respectively. The template $codeTN(Y^{(j)})$ has a change to the use of $create()$ as follows.

$$\begin{array}{rcl} codeTN(Y^{(j)}) & = & create(E_{Y^{(j)}}, c_U, c_I, c_N);\ \textbf{goto}\ J_Y \\ & E_{Y^{(j)}}: & \end{array}$$

The template for $code(X)$ has a corresponding change to the calls to $add()$: $add(L_r, c_U, c_I, \Delta)$ is replaced with $add(L_r, c_I, c_I, \Delta)$. The double use of c_I here is an artefact of the algorithm; descriptors created for the start of alternates have a GSS node whose level is the same as the current input index. In the *create* and *pop* functions, $add()$ can be called with integers which are not the same as each other.

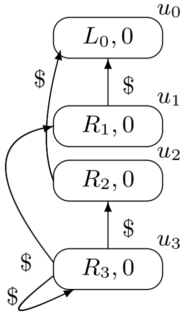
4.3 Example

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

```
        $u_0 := (L_0, 0)$;  $c_I := 0$;  $c_U := 0$;  $c_N := \Delta$; $\mathcal{U} := \emptyset$;  $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$
        goto $J_S$

$L_0$:   if ($\mathcal{R} \neq \emptyset$) {  remove $(L, k, i, w)$ from $\mathcal{R}$;
                          $c_U := k$;  $c_I := i$; $c_N := w$;  goto $L$ }
         else { if (there is an SPPF node $(S, 0, m)$) report success
                   else report failure }

$J_S$:   if ($I[c_I] \in \{b\}$)  $add(L_{S_1}, c_I, c_I, \Delta)$
        if ($I[c_I] \in \{b\}$)  $add(L_{S_2}, c_I, c_I, \Delta)$;   goto $L_0$

$L_{S_1}$:  $create(R_1, c_U, c_I, c_N)$;  goto $J_A$
 $R_1$:  if ($I[c_I] \not\in \{b\}$)   goto $L_0$
         $c_R := getNodeT(b, c_I)$; $c_I := c_I + 1$; $c_N := getNode(S ::= Ab\cdot, c_N, c_R)$
         if ($I[c_I] \in \{\Delta\}$) $pop(S, c_U, c_I, c_N)$;   goto $L_0$

$L_{S_2}$:  $create(R_2, c_U, c_I, c_N)$;  goto $J_A$
 $R_2$:  if ($I[c_I] \not\in \{b\}$)  goto $L_0$
         $c_R := getNodeT(b, c_I)$; $c_I := c_I + 1$; $c_N := getNode(S ::= Ab \cdot c, c_N, c_R)$
         if ($I[c_I] \not\in \{c\}$)  goto $L_0$
         $c_R := getNodeT(c, c_I)$; $c_I := c_I + 1$; $c_N := getNode(S ::= Abc\cdot, c_N, c_R)$
         if ($I[c_I] \in \{\Delta\}$) $pop(S, c_U, c_I, c_N)$;   goto $L_0$

$J_A$:   if ($I[c_I] \in \{b\}$)  $add(L_{A_1}, c_I, c_I, \Delta)$
         if ($I[c_I] \in \{b\}$)  $add(L_{A_2}, c_I, c_I, \Delta)$;   goto $L_0$

$L_{A_1}$:  $create(R_3, c_U, c_I, c_N)$;  goto $J_A$
 $R_3$:  if ($I[c_I] \not\in \{b\}$)  goto $L_0$
         $c_R := getNodeT(b, c_I)$; $c_I := c_I + 1$; $c_N := getNode(A ::= Ab\cdot, c_N, c_R)$
         if ($I[c_I] \in \{b\}$) $pop(A, c_U, c_I, c_N)$;   goto $L_0$
```

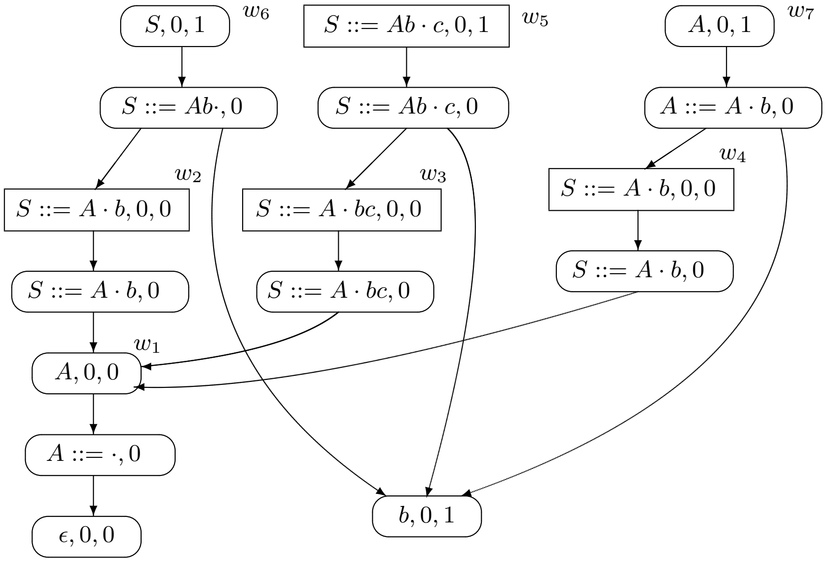
```
$L_{A_2}$: $c_R := getNodeE(c_I)$;  $c_N := getNode(A ::= \cdot, c_N, c_R)$
    $pop(A, c_U, c_I, c_N)$;   **goto** $L_0$
```

We set $I = [b, \$]$.



$$\mathcal{P} = \{\ (A, w_1),\quad (S, w_6)\ \}$$

$$\begin{aligned} \mathcal{U} \ = \ & \{\ (L_{S_1}, 0, 0, \Delta), (L_{S_2}, 0, 0, \Delta), (L_{A_1}, 0, 0, \Delta), (L_{A_2}, 0, 0, \Delta), \\ & (R_1, 0, 0, w_2), (R_2, 0, 0, w_3), (R_3, 0, 0, w_4)\ \} \end{aligned}$$



For the non-reduced parser we have:

$$\begin{aligned} \mathcal{U} \ = \ & \{(L_{S_1}, u_0, 0, \Delta), (L_{S_2}, u_0, 0, \Delta), (L_{A_1}, u_1, 0, \Delta), (L_{A_2}, u_1, 0, \Delta), \\ & (L_{A_1}, u_2, 0, \Delta), (L_{A_2}, u_2, 0, \Delta), (L_{A_1}, u_3, 0, \Delta), (L_{A_2}, u_3, 0, \Delta) \\ & (R_3, u_3, 0, w_4), (R_3, u_2, 0, w_4), (R_3, u_1, 0, w_4), (R_2, u_0, 0, w_3), (R_1, u_0, 0, w_2)\} \end{aligned}$$

$$\mathcal{P} = \{\ (u_1, w_1),\quad (u_2, w_1),\quad (u_3, w_1),\quad (u_0, w_6)\ \}$$

5 The impact of FGLL and RGLL

Factorised grammars in general generate smaller GSS and SPPF structures than their natural BNF equivalents. Significantly, we also see a reduction in the number of calls to the support functions. Recall that the purpose of the SPPF is to share sub-parts of derivations. The effect of this is that in a BNF grammar there may be several attempts to construct a shared piece of the SPPF for the common prefixes of some productions, but when factorised we only need a single instance of the prefix's derivation in the SPPF. This can result in a reduction in both the amount of processing and the size of the SPPF. Sometimes factorisation can add additional ϵ alternates and thus additional SPPF leaf nodes, as is the case in the second example below. However this is more than offset by the reductions as a result of increased node sharing.

Consider this triplet of language-equivalent grammars.

BNF1 $S ::= A\,A\,a\,a \mid A\,A\,A\,c \mid A\,A\,a\,b \qquad A ::= a\,a$

XNT1 $S ::= A\,A\,X \qquad A ::= a\,a \qquad X ::= a\,Y \mid A\,c \qquad Y ::= a \mid b$

FBNF1 $S ::= A\,A\;(\;a\;(\;a \mid b\;) \mid A\,c\;) \qquad A := a\,a$

Grammar XNT1 is a factorisation of BNF1 which uses an extra nonterminal; grammar FBNF1 uses parentheses to achieve the same effect.

When matching the string `aaaaaa` we see the following behaviour: the first two lines of results are for GLL on the two grammars, the third and fourth lines are for FGLL and RGLL, respectively, on BNF1.

	Nodes	Edges	Descriptors	NonP nodes	Pack nodes	pop set	add()	pop()	create()	getNodeE()	getNodeT()	getNode()
	GSS			SPPF			Support calls					
BNF1	8	7	16	16	9	7	16	7	8	0	17	15
XNT1	6	5	11	14	7	5	11	5	6	0	8	9
FGLL	4	3	9	13	6	3	9	3	4	0	8	8
RGLL	8	7	12	16	9	3	16	3	8	0	9	11

Both XNT1 and FGLL show a significant reduction in the size of the data structures and the number of calls made. The FGLL/BNF1 configuration is more efficient than GLL/XNT1 in respect of both data structure size and overall number of calls. (FBNF1 is the internal grammar constructed in the FGLL parser. We can construct an FGLL parser for FBNF1 directly, but this will give the same data structure statistics as the FGLL/BNF1 version.) For such a small example, the improvements seem small. However, they scale with string length.

Next we consider another triplet of grammars which generate the language of strings containing an even number of `b`'s.

BNF2 $S ::= b\,b \mid b\,b\,S$

XNT2 $S ::= b\,b\,X \qquad X ::= \epsilon \mid S$

FBNF2 $S ::= b\,b\;(\;\epsilon \mid S\;)$

When run on the string $\mathtt{b}^{20}$ we see the same general SPPF improvement for FGLL as with the previous grammars, but for XNT2 the triggering of instances of the extra nonterminals bloats the GSS, and generates an increase in the number of calls to the GSS support functions.

	Nodes	Edges	Descriptors	NonP nodes	Pack nodes	pop set	add()	pop()	create()	getNodeE()	getNodeT()	getNode()
	GSS			SPPF			Support calls					
BNF2	10	9	29	50	29	10	29	10	10	0	40	31
XNT2	20	19	39	52	30	20	39	20	20	1	20	32
FGLL	10	9	29	42	20	10	29	10	10	1	20	22
RGLL	10	9	29	50	29	10	29	10	10	0	40	31

However, this example shows that even left factoring using extra nonterminals can significantly reduce the size of the SPPF and the number of calls to the SPPF support functions, although not by as much as using FGLL. Also, the advantage is offset by an increase in GSS activity. Thus, depending on the relative efficiency of a parser's GSS and SPPF implementations, there may or may not be an overall performance advantage. Using FGLL gives the efficiency gains without the losses.

The BNF1 example shows that both FGLL and RGLL generate efficiencies for grammars where there are nonterminals whose alternates have the same nonterminals at the start. The BNF2 example shows that FGLL delivers these efficiencies in the case of common terminals rather than nonterminals.

There are also cases in which RGLL can deliver efficiencies when FGLL does not. As we have already said in Section 4, the efficiency gain for an RGLL parser arises because any valid derivation from A at a point $X ::= \alpha A \cdot \beta$ is also a valid derivation at a point $Y ::= \gamma A \cdot \delta$, if both points are reached at the same input position. Thus the parse actions associated with the second instance of A need not be repeated. In BNF1 such points occur in alternates with common left hand ends. Grammars with left recursion, whether direct or indirect, also naturally have grammar points of this form because, of course, for a left recursive nonterminal there is an initial calling point and then a subsequent recursive call which is encountered before any further input is read. This situation does not involve common left hand ends of alternates and thus FGLL does not give any efficiency improvement. For the following grammar, BNF3, both the syntactically left factored and nonterminal factored grammars are the same as BNF3.

BNF3 $\quad S ::= A\, a \qquad A ::= B\, b \qquad B ::= A\, d \mid d$

However, the RGLL parser exhibits the following efficiencies over the basic GLL parser when parsing the string $(db)^4 a$

	Nodes	Edges	Descriptors	NonP nodes	Pack nodes	pop set	add()	pop()	create()	getNodeE()	getNodeT()	getNode()
	GSS			SPPF			Support calls					
BNF3	4	4	21	19	9	13	23	13	5	0	13	15
RGLL	4	4	16	19	9	9	20	9	4	0	9	11

6 Performance on programming language grammars

In this section we give experimental results for real programs in ANSI-C, ANSI-C++, C#, and Java using grammars from the relevant programming language standards documents which have been processed using our automatic transformation, implemented as an optional step in our ART parser generator.

We have removed the lexical parts of each grammar by introducing the following rules whose left hand sides are tokens.

```
identifier ::= 'ID';
stringLiteral ::= 'STRING';
charLiteral ::= 'CHAR';
integerLiteral ::= 'INTEGER';
realLiteral ::= 'REAL';
booleanLiteral ::= 'true' | 'false';
```

We then used a separate lexical analyser to read programs written in the target language and pretty-print them with instances of identifiers and literals replaced with the corresponding tokens; thus our results are for token level grammars. Lexical structure of languages varies in complexity, and lexical processing can require significant computation; we use these 'tokenised' strings to normalise the performance of the phrase-level parser.

Our example source code strings listed below come from a variety of projects, mostly in-house, and so cannot claim to be fully representative. Nevertheless, these examples give good indications as to performance in general, and we have used them as standard benchmarks over several years.

Our grammars are drawn from programming language standards documents. We used the 1989 ANSI C standard, and a late-1997 draft of the ANSI C++ standard. Since C is (almost) a subset of C++, we can use our C examples to test both parsers; there is also one C++-specific string. The C++ grammar is considerably more baroque than the C grammar, so we would expect parsing to be more expensive. Our C# grammar is from the 2002 ECMA standard for C# version 1.2, and for Java, we have used the BNF grammar from the first Java Language Standard. We note that these grammars are far from LL-deterministic. Our tools report numbers of LL(1) violations ranging from 335 for the C# grammar to 162 for the Java grammar; the C# grammar has 56 left recursive nonterminals and the Java grammar has 36; and all the grammars apart from the one for ANSI-C have nonterminals X such that $X \overset{*}{\Rightarrow} \epsilon$ but $\textsc{first}(X)$ and $\textsc{follow}(X)$ are not disjoint. The test strings are as follows:

Name	Language	Tokens	Function
gtb_src.tok	C	36,827	Full source code for the grammar tool box
rdp_full.tok	C	26,551	Full source code for the rdp RD parser generator
artsupport.tok	C++	36,444	C++ support for ART parsers (two concatenations)
twitter.tok	C#	33,840	Fragment from a Twitter application (ten concatenations)
life.tok	Java	36,975	Conway's Game of Life with graphics (ten concatenations)

The efficiency gains associated with the RGLL, FGLL and combined RFGLL variants are shown in Table 1 in comparison to the performance of the base algorithm described in [12]. All four parsing methods have been implemented in the same programming language, compiled with identical levels of optimization, and are programmed with similar data structures and

	CPU s	tokens/s	Descriptors	Symbol Nodes	Packed nodes	Nodes	Edges	$\|P\|$
				SPPF		GSS		
ANSI C++ on artsupport.tok								
base	2.00	18,249	9,493,519	473,257	475,542	1,036,075	4,755,333	874,868
RGLL	0.84	43,231	1,894,009	473,257	475,542	1,036,075	4,755,333	327,317
FGLL	0.47	77,872	1,988,699	455,001	442,696	699,979	1,041,455	550,054
RFGLL	0.44	83,396	1,285,245	455,001	442,696	699,979	1,041,455	327,317
ANSI C++ on gtb_src.tok								
base	2.89	12,761	13,061,222	561,139	562,843	1,270,903	6,392,785	1,110,400
RGLL	1.09	33,755	2,366,346	561,139	562,843	1,270,903	6,392,785	400,296
FGLL	0.59	62,103	2,584,505	546,631	534,871	851,117	1,344,152	696,118
RFGLL	0.55	67,449	1,581,871	546,631	534,871	851,117	1,344,152	400,296
ANSI C++ on rdp_full.tok								
base	2.01	13,196	9,687,071	425,385	426,291	942,742	4,709,390	841,963
RGLL	0.78	34,040	1,771,700	425,385	426,291	942,742	4,709,390	305,134
FGLL	0.45	58,611	1,946,233	413,356	404,307	628,871	999,579	529,526
RFGLL	0.39	68,079	1,185,093	413,356	404,307	628,871	999,579	305,134
ANSI C on gtb_src.tok								
base	0.76	48,203	4,178,345	297,677	261,401	564,437	2,042,843	559,859
RGLL	0.41	90,707	1,127,572	297,677	261,401	564,437	2,042,843	220,185
FGLL	0.30	124,416	1,331,757	301,884	252,578	377,255	665,042	362,288
RFGLL	0.25	147,308	766,618	301,884	252,578	377,255	665,042	220,185
ANSI C on rdp_full.tok								
base	0.53	50,002	3,122,638	222,206	195,799	417,204	1,510,486	425,730
RGLL	0.30	89,397	844,344	222,206	195,799	417,204	1,510,486	167,593
FGLL	0.22	121,237	1,009,146	226,338	190,570	279,895	496,272	279,154
RFGLL	0.19	141,984	576,271	226,338	190,570	279,895	496,272	167,593
C# 1.2 on twitter.tok								
base	0.39	86,769	2,024,014	255,343	225,052	443,304	1,056,916	390,990
RGLL	0.28	120,427	837,254	255,343	225,052	443,304	1,056,916	170,453
FGLL	0.25	135,360	1,157,630	256,449	220,738	365,471	606,718	317,958
RFGLL	0.22	154,521	695,780	256,449	220,738	365,471	606,718	170,453
Java JLS1 on life.tok								
base	0.42	87,827	2,302,532	260,377	223,401	505,179	1,249,154	402,501
RGLL	0.31	118,510	911,430	260,377	223,401	505,179	1,249,154	171,002
FGLL	0.28	131,584	1,321,207	255,502	212,601	414,304	755,554	311,701
RFGLL	0.23	158,013	740,655	255,502	212,601	414,304	755,554	171,002

Table 1: Performance on programming language grammars

algorithmic approaches. The pattern for all of our combinations of grammars and strings is that FGLL achieves greater speedup than RGLL, but that using both techniques together offers the best performance.

Although FGLL is the most effective mechanism for nearly all grammar idioms, as illustrated by the grammar BNF3 above, FGLL cannot improve the performance of left recursive rules, since there is no factorisation for such rules. The RGLL technique, though, can reduce the amount of work required when parsing left recursive rules. On the other hand, RGLL alone does not exploit all of the speedup available to FGLL.

The main driver for performance is the number of independent computations being performed, which corresponds to the number of descriptors created. The detailed timing effects are complex since individual descriptors might process only an epsilon production, a production with a single nonterminal instance or a production with a long sequence of terminals. Variations in the size of the GSS and SPPF are also significant, since initial node creation is significantly more costly than simply finding a pre-existing node.

Table 1 also describes the size of the main GLL data structures, which gives an indication of memory use. The RGLL variant constructs the same SPPF and GSS as the base algorithm, but in each case very substantially reduces the number of descriptors. FGLL provides substantial reductions in the size of the GSS, and usually also in the size of the SPPF. Epsilon rules can generate additional nodes in the FGLL SPPF, and in the ANSI-C and C# examples this effect is sufficient to cause a small increase in the overall size of the FGLL SPPF. The FGLL variant also generates larger pop element sets (recorded in the column labelled $|\mathcal{P}|$) than the RGLL version, but actually performs fewer applications of the pop elements. The combined RFGLL algorithm displays the lower number of pop elements.

Apart from this reduction in space, the reduction in activity leads to significantly greater throughput, with speedup factors between about 2 and 5 for these grammars. In this paper we have limited our discussion to algorithm-level variants of the original GLL parsing algorithm. A detailed consideration of the effects driving these reductions in processing time and space requires analysis at the level of a particular implementation (see for instance [4]); we shall examine the space of data structure and control flow implementations for GLL in a future engineering-focused paper.

7 Ordering descriptor processing

We complete our algorithmic level GLL performance considerations by looking at the possible impact of the order in which descriptors are processed and at some other *ad hoc* issues.

As we have already mentioned in Section 2.5, it is possible for new edges to be added to a GSS node after a pop action has been applied to that node. For this reason the set, $\mathcal{P}$, of pop actions is maintained and the *create*() function applies these pops, referred to as contingent pops, when it adds a new edge to an existing node. We can ask whether it is possible to process the descriptors in some order that ensures that all the edges are added to a GSS node u before any pop action is carried out on u. In fact this is not the case. In some cases a new edge is added from u as a result of a pop action being applied to u. The grammar in Section 2.4 has this property for RGLL, and for the closely related grammar

$$S ::= \ A \ A \ c \qquad A ::= \ B \qquad B ::= \ \epsilon$$

the base GLL algorithm requires contingent pops, regardless of the descriptor processing order.

The need for contingent pops is paralleled in other general parsing algorithms. In Earley's algorithm [3], when the COMPLETER adds a new item, $(X ::= \alpha \cdot D\beta, k)$ say, it checks to see if the REDUCER has already applied a reduction of the form $D ::= \gamma \cdot$ at the current level k. If so then the reduction is immediately applied to the new item. For grammars with right recursion, in Tomita's GLR algorithm [15] the REDUCER can add new edges to existing GSS nodes. Tomita addressed this by adding duplicate nodes, but this causes nontermination for grammars with hidden left recursion. Farshi [9] 'solved' the problem by doing a brute force search rather than using duplicate nodes, but this was not very efficient. The RNGLR algorithm [10] solved the problem by extending the definition of a reduction, effectively defining 'contingent reductions'.

Since it is not possible to order the GLL descriptors so that pops are only applied once all the edges of a node have been constructed, we cannot avoid the overhead of maintaining the set $\mathcal{P}$ and checking for contingent pops in the *create*() function. However, depending on the implementation, it may improve the efficiency of a GLL algorithm if elements are added to the set $\mathcal{P}$ as late as possible, thus keeping the number of contingent pops as low as possible. (Note, the total number of pops always remains the same, decreasing the number of contingent pops increases the number of non-contingent pops.) In particular, a naïve implementation might perform a linear search of $\mathcal{P}$ when a contingent pop is detected, and that could become very costly.

Because a descriptor which pops a GSS node will always be created after a descriptor which creates the node has been processed, processing descriptors in creation order is likely to reduce the number of contingent pops. This has the advantage of not requiring any additional 'machinery'. However, ordering the processing by descriptor type can reduce the number of contingent pops further.

The processing of a given descriptor $(X ::= \alpha \cdot \beta, u, i, w)$ begins when it is removed from $\mathcal{R}$ and concludes with a `goto` $\mathtt{L}_0$ action. The `goto` action either follows the return from a call to *pop*(), or follows a `goto` $\mathtt{J}_A$ action which follows a return from a call to *create*(). The latter case occurs when β is of the form $vA\delta$, and the former case occurs when β does not contain any nonterminals. If β has no nonterminals we refer to $(X ::= \alpha \cdot \beta, u, i, w)$ as a *popping descriptor*. The number of contingent pops can be reduced to close to minimum by selecting a popping descriptor for processing only if there are no non-popping descriptors in $\mathcal{R}$. Of course, depending on the implementation this may introduce an overhead associated with establishing the descriptor type.

The table below shows the worst and best variants for these optimisations for our six grammar and input string examples. In each case, the data labelled 'stack plain' shows the effect of grouping all descriptors into a single set, and accessing it in Last-In, First-out (stack) order. Since there is only one set of descriptors, there are no delayed descriptors to be processed. The 'queue delayed' variant splits the descriptors into non-popping (Immediate) and popping (Delayed) subsets. Descriptors are only fetched from the Delayed set if the Immediate set is empty, and both sets are accessed in creation order.

The impact on contingent pops is dramatic: in the case of C# and Java the 'queue delayed' model yields zero contingent pops on these input strings, and for ANSI C and C++ the number of contingent pops is reduced by four orders of magnitude.

The work done by contingent and primary pops is equivalent; only the point within the algorithm at which they are executed varies, with the caveat that we must be able to locate contingent pops efficiently. An implementation which naïvely searches the whole pop set for applicable contingent pops might display very significant speedup for 'queue delayed' over 'stack plain'. In our implementation, in addition to the global pop set we maintain local lists of pops for each GSS node or GSS cluster, and that allows us to efficiently iterate over the contingent pops without searching the whole pop set. The data in this table shows that this optimisation is likely to be crucial for any efficient GLL implementation; however once efficient iteration over pops is available, the extra performance gains associated with descriptor ordering are very small since the reduced iteration is counterbalanced by the extra computation of conditionals that is required for every descriptor set access.

	Descriptors		Pops		CPU s
	Delayed	Immediate	Primary	Contingent	
ANSI C++ on artsupport.tok					
stack plain		9,493,519	2,768,466	2,108,950	2.00
queue delayed	1,737,840	7,755,679	4,944,148	6,046	2.00
ANSI C++ on gtb_src.tok					
stack plain		13,061,222	3,995,505	3,144,256	2.89
queue delayed	2,394,127	10,667,095	7,396,673	7,935	2.85
ANSI C++ on rdp_full.tok					
stack plain		9,687,071	2,978,431	2,368,538	2.01
queue delayed	1,800,176	7,886,895	5,554,538	6,894	2.00
ANSI C on gtb_src.tok					
stack plain		4,178,345	1,334,537	829,473	0.76
queue delayed	995,059	3,183,286	2,164,241	95	0.73
ANSI C on rdp_full.tok					
stack plain		3,122,638	1,007,294	623,496	0.53
queue delayed	741,485	2,381,153	1,630,858	6	0.52
C# 1.2 on twitter.tok					
stack plain		2,024,014	684,842	289,786	0.39
queue delayed	476,648	1,547,366	976,898	0	0.38
Java JLS1 on life.tok					
stack plain		2,302,532	676,201	378,900	0.42
queue delayed	497,826	1,804,706	1,055,101	0	0.42

Finally we note two other potential efficiency savings. It is possible to make the storage space required for the GSS smaller. The SPPF node labels on the GSS edges are not strictly necessary. The label of the SPPF node can be computed from the label of the source GSS node and from the level of the target GSS node. Of course, this label does have to be computed and then the SPPF node must be found. Whether it is more efficient to record the nodes in the GSS or to compute the labels will depend on the implementation. Furthermore, since the size of the GSS is worst case quadratic while the size of $\mathcal{U}$ is worst case cubic, the size of the GSS is unlikely to be an issue in practice.

The fact that the GSS edge label can be computed implies that all the edges between two given nodes will have the same label. Although we have not constructed a proof, we believe that the nature of the GLL algorithm ensures that there is never an attempt to insert the same GSS edge twice. An informal argument supporting this has been given by Afroozeh [1]. We have also used ART to count the number of attempts to insert a GSS edge for all of the grammars and strings in our extensive repository of test examples, and in all cases this number is equal to the number of GSS edges, adding experimental support for the hypothesis. Thus the test on the third line of the *create*() function in Section 2.5 may probably be suppressed without increasing the work done by the parser. The test is not, in any case, required for algorithm correctness it is just there to avoid repeated computation.

8 Conclusions and related work

We have extended our GLL algorithm to FGLL and RGLL, and shown that throughput improvements of up to a factor five are available on real programming language grammars. These speed improvements are accompanied by very significant reductions in the cardinalities of the data structures constructed by the GLL algorithm, with concomitant space reductions.

We have demonstrated that syntactic left factorisation of grammars delivers significant gains, and this technique will be applicable in many parsing technologies. In FGLL the factorisation is essentially invisible to the user as an input BNF grammar is automatically transformed and the BNF derivations are reconstructed after parsing is complete. Of course, FGLL performs identically to GLL where the grammar offers no opportunities for factorisation.

The RGLL technique is specific to GLL parsers. However, the underlying context free property on which it is based could yield analogous improvements to other styles of top down parser. The changes to base GLL required for the FGLL and RGLL variants are orthogonal to each other, and can be applied together. Our experimental results demonstrate the resulting FRGLL parsing algorithm is preferable in all cases to the base GLL algorithm.

We have also discussed two mechanisms for ordering descriptor evaluation: the extraction of descriptors in LIFO or FIFO order; and the partitioning of the descriptor set into popping and nonpopping elements. We have shown that for naïve implementations which require search, the reduction in contingent pops arising from the partitioning approach is extremely significant, but that for implementations which maintain separate lists of pops in addition to the main set, the improvements are slight.

In an earlier paper on generalised LR parsing [10] we gave a review of the landscape of nondeterministic grammar parsing approaches. As far as we are aware, little has been published on the usefulness of syntactically left factoring grammars to improve parser performance. This is not particularly surprising as the concepts of syntactic left factorisation and combined nonterminal instance processing are not natural topics for study for LR-style parsers; the LR tables already incorporate these efficiencies. So the literature on GLR parsing does not address the topics covered in this paper. For example, the GLR-based ASF+SDF compiler generator [16] accepts EBNF constructs but converts them to BNF for LR table construction.

Top-down techniques [7] lend themselves to efficient direct implementation of EBNF grammars. However, much of the focus of generalised top-down parser generators has been on dealing efficiently with issues such as backtracking and left recursive grammars. The tendency to use on-the-fly semantic evaluation may also have discouraged interest in automatic left factoring.

There is little work directly related to RGLL, however the Rascal [6] developers have implemented a preliminary version of RGLL and informally report efficiency improvements. But the work is still under development and there is no published statistical data.

The work presented in this paper sits at the nexus of LR-style factoring and direct EBNF support, offering efficient top-down EBNF-style parsing whilst retaining the structures and behaviour inherent in the underlying user-supplied BNF grammar.

References

[1] A. Afroozeh. *Private Communication.* CWI Amsterdam, 2014.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, 1986.

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

[4] A. Johnstone and E. Scott. Modelling GLL parser implementations. In M. d. B. B.Malloy, S.Staab, editor, *SLE 2010*, volume 6563 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, 2011.

[5] A. Johnstone and E. Scott. Translator generation using ART. In M. d. B. B.Malloy, S.Staab, editor, *SLE 2010*, volume 6563 of *Lecture Notes in Computer Science*, pages 306–315. Springer-Verlag, 2011.

[6] P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation*, pages 108–177. IEEE, 2009.

[7] D. E. Knuth. Top-down syntax analysis. *Acta Informatica*, 1:79–110, 1971.

[8] P. Ljunglöf. *Pure Functional Programming, an advanced tutorial.* Göteburg University, Sweden, 2002.

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

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

[11] E. Scott and A. Johnstone. GLL parsing. *Electronic Notes in Theoretical Computer Science*, 253:177–189, 2010.

[12] E. Scott and A. Johnstone. GLL parse-tree generation. *Science of Computer Programming*, 78:1828–1844, 2013.

[13] E. Scott, A. Johnstone, and G. Economopoulos. A cubic Tomita style GLR parsing algorithm. *Acta Informatica*, 44(6):427–461, 2007.

[14] A. P. ten Brink. *Disambiguation mechanisms and disambiguation strategies, Masters Thesis.* Eindhoven University of Technology, 2013.

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

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