A Reference GLL Implementation

Adrian Johnstone
a.johnstone@rhul.ac.uk
Royal Holloway University of London
UK

Abstract

The Generalised-LL (GLL) context-free parsing algorithm was introduced at the 2009 LDTA workshop, and since then a series of variant algorithms and implementations have been described. There is a wide variety of optimisations that may be applied to GLL, some of which were already present in the originally published form.

This paper presents a reference GLL implementation shorn of all optimisations as a common baseline for the real-world comparison of performance across GLL variants. This baseline version has particular value for non-specialists, since its simple form may be straightforwardly encoded in the implementer's preferred programming language.

We also describe our approach to low level memory management of GLL internal data structures. Our evaluation on large inputs shows a factor 3–4 speedup over a naïve implementation using the standard Java APIs and a factor 4–5 reduction in heap requirements. We conclude with notes on some algorithm-level optimisations that may be applied independently of the internal data representation.

CCS Concepts: • **Software and its engineering → Parsers**; • **Theory of computation → Grammars and context-free languages**.

Keywords: Programming language syntax specification, GLL parsers, GLL implementation

ACM Reference Format:
Adrian Johnstone. 2023. A Reference GLL Implementation. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3623476.3623521

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SLE '23, October 23–24, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0396-6/23/10…$15.00
https://doi.org/10.1145/3623476.3623521

1 Introduction

GLL was introduced at the 2009 LDTA workshop [10] in the form of a generator that produces generalisations of simple recursive descent recognisers; this is extended to a full parser in [11]. A variety of extensions to the basic GLL algorithm have been reported, including direct handling of EBNF constructs [13] and multi-parsing with applications to generalised lexing [14, 16]. The control flow within GLL parsers naturally lends itself to combinator-style GLL implementations which are particularly suited to functional programming languages, and several combinator GLL implementations have been reported [4, 17, 21]. The Rascal meta-programming language [7] deploys GLL style parsers.

Performance optimisations of classical GLL include the FGLL variant which improves performance on left-factored grammars, and the RGLL variant which requires fewer independent processing threads [12]. Space and performance optimisations arising from encoding derivation forests using Binary Subtree Representation sets are explored in [15] leading to a discussion of *clustering* and the development of the Clustered Nonterminal Processing (CNP) variant. Machine-level approaches to optimising the data structures required by the GLL algorithm are explored in [5].

This paper provides a new presentation of GLL as a fixed-form 'interpreted' parser which is parametrised by an in-memory representation of the grammar. We have two main objectives (i) to provide an easily accessible 'reference' version of GLL that we hope will facilitate adoption, and (ii) to provide a baseline implementation that allows the throughput and memory consumption of GLL variants to be compared in a principled fashion.

The approach taken here is avowedly procedural in style. The code is written in Java, but is trivially portable to ANSI-C. We have done this so as to provide a reference implementation that minimises dependencies on particular programming styles such as object orientation or functional combinators.

At the datastructure level we provide two implementations: gllBL (baseline) and gllHP (hash pool). The first uses the standard Java API methods to implement sets and lists; this allows for a more readable presentation. The gllHP variant explains how we use low level memory management to enhance performance in our production parsers. We have previously compared and contrasted an idiomatically 'pure' object oriented implementation to an optimised procedural implementation. In that study we found the OO variant to impose a performance overhead [6]. However, there have

been significant advances in Java compiler and Java Virtual Machine performance in the intervening years, and it would be interesting to revisit that work.

By way of introduction, we review the way in which standard recursive descent parsers can be extended to a wider class of grammars by incorporating backtracking. We call our particular approach Ordered Singleton Backtracking Recursive Descent (OSBRD), and note some of its failure modes. We then show how GLL generalises recursive descent by directly handling the parse function call stacks in a combined graph, and by recording parser configurations in process descriptors, allowing all parsing choices to be explored in worst case cubic time and space.

In the rest of this section we summarise the background material needed to describe our implementation. In Section 2 we focus on control flow, moving from a compiled (non-general) OSBRD parser to the state-based interpretive style that we use for GLL. In Section 3 we give a detailed account of gllBL, our baseline GLL algorithm. In Section 4 we present an efficient data representation for the GLL data structures, and in Section 5 we evaluate the performance of these GLL variants on a small number of large and diverse examples. Section 6 contains notes on opportunities to improve the performance of our baseline algorithm, and we conclude in Section 7 with an informal 'practicality' test.

Software artefacts corresponding to this presentation are available in a public repository at https://github.com/AJohnstone2007/referenceImplementation.

Grammar notation A *Context Free Grammar* (CFG) is a 4-tuple $\Gamma = (N, T, S, P)$ denoting respectively, a set of nonterminals, a set of terminals, a start nonterminal and a set of productions with $N \cap T = \emptyset, S \in N, P \in N \times (N \cup T)^*$. An *Extended Context Free Grammar* (ECFG) has productions $P \in N \times \rho$ where ρ is a regular expression over $(N \cup T)^*$.

For small examples, we use the following conventions: ϵ denotes the empty string; a, b, c, x, y, z are elements of T; X, Y, Z are elements of N (along with S); u, v, w are elements of T^*; and $\alpha, \beta, \gamma \in (N \cup T)^*$.

We may specify a grammar simply by enumerating its set of productions in the form $X \rightarrow \alpha$ with the convention that the left-hand side of the first production is the start symbol; where we have multiple alternate productions with the same left hand side $X \rightarrow \alpha \quad X \rightarrow \beta \quad X \rightarrow \ldots$ we may use the shorthand $X \rightarrow \alpha \mid \beta \mid \ldots$

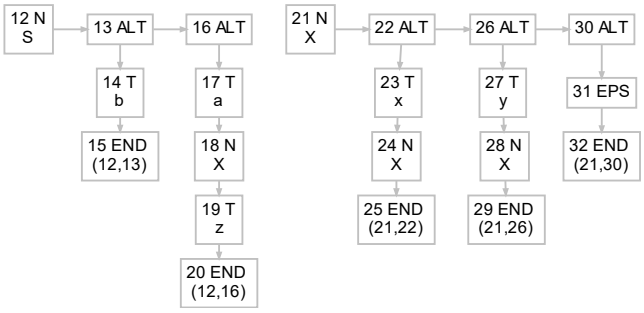
The *derivation step* relation $\Rightarrow$ captures the notion of language generation from a grammar: if we have $\alpha X \gamma$ and $X \rightarrow \beta \in P$, then $\alpha X \gamma \Rightarrow \alpha \beta \gamma$. We write $\overset{*}{\Rightarrow}$ to denote the *derives* relation: for instance $S \overset{*}{\Rightarrow} \alpha$ is the set of *sentential forms* derivable from the start symbol, and $S \overset{*}{\Rightarrow} u$ is the set of *sentences* derivable from the start symbol, that is $L(\Gamma)$ the *language* of Γ.

The special symbol \$ (with $\$ \notin T$) denotes the *end of input string* marker which is appended to putative sentences before input to our parsing and recognition algorithms.

Grammar representation This paper is about implementation, so we must present concrete representations of grammars. We use a set of trees, one for each nonterminal in N. Grammar tree nodes are labelled with a unique integer node index *ni*, a grammar element *el* and two child references named *alt* and *seq*. An appropriate Java declaration is:

```
class GNode{int ni; GElement el; GNode alt, seq;}
```

Using instances of GNode, the grammar $\Gamma_1 = \{S \rightarrow b \quad S \rightarrow aXz \quad X \rightarrow xX \quad X \rightarrow yX \quad X \rightarrow \epsilon\}$ is represented as



In this visualisation, *alt* references are shown as horizontal arrows, and *seq* references as vertical arrows. The instance numbers for grammar nodes are allocated sequentially from a base value which equates to $|T| + |N| + E$ (in this example $5 + 2 + 5 = 12$) where E is the number of element types in the grammar — we shall enlarge on this in Section 4.

Each rule's right-hand side is represented by a sequence of nodes linked by their `seq` reference, terminated with an `END` node and headed by an `ALT` node; all `ALT` nodes for a given nonterminal are linked via their `alt` references and headed by an `LHS` node labelled with the left hand side nonterminal. For `END` nodes, `seq` references the nearest `ALT` ancestor and `alt` references the nearest ancestor `ALT`-header, which for ordinary (non-extended) CFGs will be an `LHS` node. To avoid cluttering the visualisation, `END` references are shown as an ordered pair of reference numbers rather than arrows.

Grammar elements are tuples (*ei, kind, str*) where *ei* is a unique element index, *kind* is one of EOS, T, EPS, N, ALT, END (for end-of-string, terminal, empty string, nonterminal, alternate and end of production) and *str* is a nonterminal or a terminal appropriately. In Java these may be declared as:

```
class GElement {int ei; Kind kind; String str;}
enum Kind {EOS, T, EPS, N, ALT, END}
```

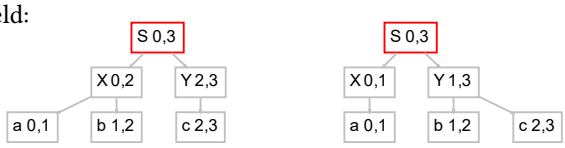
Derivation trees and ambiguity Any production with more than one nonterminal on its right hand side gives rise to multiple derivations in an uninteresting way. For instance the grammar $\Gamma_3 = \{S \rightarrow XY \quad X \rightarrow x \quad Y \rightarrow y\}$ can generate xy in two ways: $S \Rightarrow XY \Rightarrow xY \rightarrow xy$ and $S \Rightarrow XY \Rightarrow Xy \rightarrow xy$. A *leftmost derivation* contains only derivation steps in which the first nonterminal in a rule is expanded, and by convention we shall use only leftmost derivations.

A *derivation tree* is a useful graphical representation of a class of derivations of some string. A derivation tree is an

ordered tree in which the root node is labelled with S, each interior node is labelled with an element of N and each leaf is labelled with some element of T or ϵ. If an interior node X has children labelled $x_1, x_2, \ldots x_n$ then $X \longrightarrow x_1, x_2, \ldots x_n \in P$. The leftmost derivation of $u \in L(\Gamma)$ corresponds to the pre-order traversal of the derivation tree of u.

A CFG Γ is *ambiguous* if there is some $u \in L(\Gamma)$ which has more than one leftmost derivation, and thus more than one derivation tree. For instance the grammar $\Gamma_4 = \{S \longrightarrow XY \quad X \longrightarrow a \mid ab \quad Y \longrightarrow bc \mid c\}$ generates the string abc in two leftmost ways: $S \Rightarrow XY \Rightarrow aY \Rightarrow abc$ and $S \Rightarrow XY \Rightarrow abY \Rightarrow abc$. The corresponding derivation trees are:

The *yield* of a derivation tree node is the substring whose terminal nodes are descendants of that node. In the example above, we see that the terminal nodes and the root node have the same yield in each derivation, but the nodes labelled X and Y have different yields in the two derivations. In any derivation tree, we can find the yield of a node by descending to its leaves, but when dealing with ambiguities it is convenient to annotate each node with its yield. An *annotated derivation tree* is a derivation tree whose node labels have been extended by the left and right indices of the node's yield:



With these annotations we can directly see that the start and terminal nodes of both derivations are 'the same' because their yields are the same in each case, but that the X and Y nodes are different because their corresponding derivation steps generate different substrings of the input.

Recognisers, partial parsers and general parsers A *recogniser* for Γ tests a string u for language containment, i.e. whether $u \in L(\Gamma)$. A *partial parser* tests a string for language containment and returns at least one derivation for at least one string in $L(\Gamma)$. A *general* parser returns all derivations. Most current programming language processors employ partial parsers which (i) admit only a subset of the context free grammars and (ii) return at most one derivation. For many parsing algorithms, (i) is well characterised, but the constraints imposed by (ii) are less well understood in practice for non-trivial cases, which can lead to puzzling outcomes.

As programming languages become more complex, anecdotal evidence shows that implementers increasingly struggle with classical near-deterministic parser generators and resort to hand written front ends. There is a useful discussion at [18] which notes that GCC abandoned Bison based parsers nearly twenty years ago [2, 3]. We hope that the availability of well engineered GLL parsers will allow a return to principled engineering of compiler front ends.

Compiled vs. interpreted parsers We distinguish between *interpreted* parsers which are fixed pieces of code that are parameterised by a data structure encoding the grammar, and *compiled* parsers where the parser code itself encodes the grammar. Classical recursive descent parsers are compiled parsers: they have a parse function for each nonterminal, and the body of a parse function reflects the nonterminal's productions.

Traditionally, shift-reduce parsers are implemented as interpreted parsers operating over a table that represents an automaton derived from the grammar. However, Penello [9] describes *Recursive Ascent* (a compiled LALR parser), and Aho and Ullman give a table-driven predictive LL(k) parser [1, pp338–341].

Parser context Parsing is a *search* problem in which we traverse both a grammar and an input string to locate derivations. The algorithms we shall examine in this paper all work by processing a current parser *context* comprising an index `i` into the input string, a current grammar node `gn`, a current stack, whose top is stack node `sn`, and a current derivation whose most recent step is derivation node `dn`.

In detail (i) compiled parsers do not have an explicit `gn` since the grammar positions correspond to locations in the code; (ii) recognisers do not generate derivations and so do not require a `dn` and (iii) some parsers make use of the host languages call stack, and thus do not need an explicit `sn`.

Lexicalisation Most programming language grammars are defined over terminals which have internal structure rather than simple characters, and the input character string is usually *lexicalised* into a sequence of non-overlapping substrings called *lexemes* each of which belongs to a lexical class which is given a number called the *token*. There are several advantages to this scheme: whitespace may elided from the grammar rules; the alphabet of the grammar is the set of lexeme classes not the set of characters and that improves the resolution of lookahead tests; and for many languages regular recognisers may be used for lexicalisation rather than the full power of a context free parser, and that improves performance. The parser itself then works with strings of tokens, not strings of characters.

The algorithms presented here all assume that inputs have been lexicalised into a string of tokens using longest match.Token number zero is reserved for the end of string symbol \$.

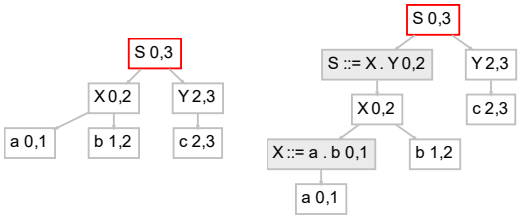
Representing derivation forests A general parser must return all derivations for any string in the language. Simply returning individual derivation trees is impractical as the number of derivations can grow very quickly: consider the grammar $\Gamma_5 = \{Z \longrightarrow S \mid SZ \quad S \longrightarrow XY \quad X \longrightarrow a \mid ab \quad Y \longrightarrow bc \mid c\}$ which is Γ_4 extended with a new start rule, allowing one or more abc substrings to be generated. Each instance

of S will have two ways to generate its substring, hence the total number of derivations will be 2^k where k is the number of substrings.

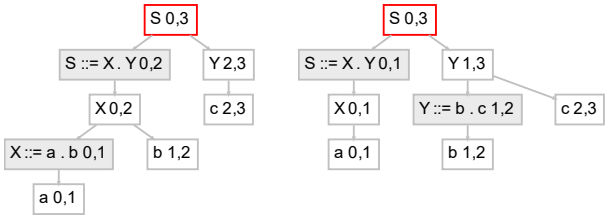
We noted above that some of the elements of the two derivations in Γ_4 are the same in that they have matching labels, and thus generate the same substring of the element. Tomita [20] observed that derivation trees may be combined by *sharing* and *packing* into a directed acyclic graph called a Shared Packed Parse Forest (SPPF). We say that an SPPF *embeds* derivation steps and derivations. If $u \in L(\Gamma)$ then the SPPF for u in Γ will contain a node labelled $S, 0, n$ where S is the start symbol and n is the length of u, in other words a node labelled with the start symbol whose yield is the entire string.

In Tomita's original formulation, SPPFs embed standard derivation trees in which internal nodes are labelled with a nonterminal and have out-degree k_p where k_p is the length of some production p. Whilst building derivations, we need to know both the production we are working on and the position within it. If we were to use this form of SPPF, then in general the dn element of our parser context would need to be a pair $p, j (0 < j < k)$ where j specifies a position within the production. Instead, we binarise our derivation trees by adding additional intermediate nodes.

This binarisation of derivations allows us to encode a complete grammar position into a single SPPF node, so that the dn field in our parser context can comprise a single node. For the grammar $\Gamma_6 = \{S \rightarrow abc\}$ the single derivation in 'flat' and binarised forms are:

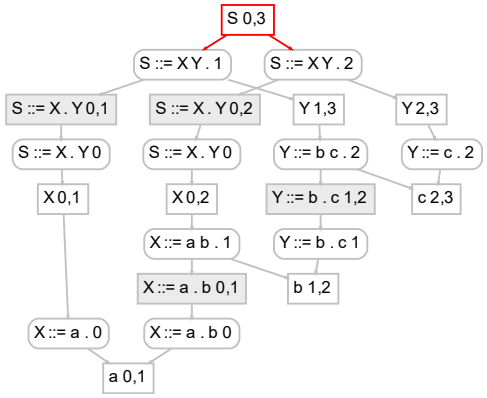


The binarised annotated derivation trees for Γ_4 are



The binarisation or 'intermediate' nodes are labelled with a position *within* a production. We use Knuth's 'item' notation and mark the position with a dot. In visualisations, we shade the binarisation nodes.

The corresponding SPPF constructed from the two annotated derivation trees for Γ_4 is



2 Backtracking Recursive Descent Parsers

To motivate the GLL approach, we begin by examining the simplest useful CFG parser we know of: a compiled non-general backtracking parser with only a single thread of control. We then present the same algorithm implemented in two interpreted styles: the first a 'folding' of the compiled parser and the second a state-machine implementation closely related to the interpreted GLL parser to be discussed in the next section.

Compiled style - osbrdG Here is one style of backtracking parser for

$\Gamma_1 = \{S \rightarrow b \mid aXz \quad X \rightarrow xX \mid yX \mid \epsilon\}$

```
1  boolean p_S() { // Attempt to match nonterminal S
2    int el = i; DNode eDN = dn; //store global variables at entry
3
4    if (input[i]==2/*b*/) {i++; du(13); return true;}
5
6    i = el; dn = eDN; //recall global variables for next production
7    if (input[i]==1/*a*/) {i++; // bump input pointer on success
8      if (p_X()) { // call parse function for nonterminal Z
9        if (input[i]==5/*z*/) {i++; // bump input pointer on success
10         du(16); return true;}}} // end; update derivation
11   return false;
12 }
13
14 boolean p_X() { // Attenmpt to match nonterminal S
15   int el = i; DNode eDN = dn; //store global variables at entry
16
17   if (input[i]==3/*x*/) {i++;
18   if (p_X()) { du(22); return true;}}
19
20   i = el; dn = eDN; //recall global variables for next production
21   if (input[i]==4/*y*/) {i++;
22   if (p_X()) { du(26); return true;}}
23
24   i = el; dn = eDN; //recall global variables for next production
25   /* epsilon */ du(30); return true; // epsilon always matches
26 }
```

A parse begins by loading the input with the lexicalised sequence of tokens, setting `i` to zero, `dN` to `null` and then calling `p_S()`. The string is accepted if on return, `input[i]` is the end of string symbol.

We call this algorithm *Ordered Singleton Backtrack Recursive Descent* (OSBRD) to emphasise that it (i) treats the productions of a nonterminal in an ordered fashion (which may cause some productions to be ignored) and (ii) returns at most one derivation.

This particular implementation is called `osbrdG` because the code has been *generated* from an input grammar. If we change the grammar, then we must regenerate a new parser. The generated code is effectively a bidirectional 'pretty print' of the grammar, in the sense that one may read the grammar in a single pass and output the parser code, and one may read the code in a single pass and output the grammar. Nigel Horspool calls this the *Recursively Decent* property.

Each nonterminal Z has a corresponding boolean parse function `p_Z()`. On entry, parse functions record the input index at entry `eI` and the entry derivation node `eDN`. The productions $Z \rightarrow \alpha$ are then examined in the order they were written. The sequence of elements in α is tested using a nest of `if` statements: nonterminals are tested by calling the corresponding parse function, and terminals by testing the current input character against that terminal's token number (with the current index being incremented on success). If all of a production's tests return true, then the derivation is extended by one step using function `du(n)` and the parse function returns true, otherwise the current index and derivation node are reset to their entry values and the next production is tested. If no production matches, the parse function returns false.

The leftmost derivation is encoded as a linked list of production numbers: function `du(int n)` updates the current derivation by head-inserting a node: in detail the production number is the number of the corresponding `ALT` node in our representation. A suitable Java declaration for derivation nodes is **class** DNode {**int** altn; DNode next;}

There are three major deficiencies in the OSBRD algorithm: (i) on some inputs it does not terminate; (ii) on some inputs it will require exponential time to terminate; and (iii) on some inputs it will terminate but incorrectly reject strings that are in the language.Versions of this algorithm have been reported many times. The most comprehensive treatment is in Aho and Ullman's 1972 monograph [1, pp.56–469] where the technique is called *TDPL*: they note that *It can be quite difficult to determine what language is defined by a TDPL program* which should be read as a warning. We do not recommend the approach for serious work; we use it here merely as a stepping stone to understanding GLL parsing.

Interpreting via a function - osbrdF There is a closely related interpreted implementation that 'folds' all of the parse functions into a single function `osbrdF` which takes a grammar node representing the nonterminal to be tested – in our implementation, we use the corresponding `LHS` node from our representation.

Instead of laying out the productions as nests of `if` statements, we have an outer loop over the `alt` references and an inner loop over the `seq` references enclosing a switch statement which performs the appropriate action for each kind of grammar node. Java (and many other languages) use the dot operator to specify fields from composite data structures hence, for instance, `gn.s` refers to the string field of the current grammar node. We make use of Java's named-continue feature to allow a failed match to immediately proceed to the next `alt` iteration. An implementation in C or C++ might use a `goto` to achieve the same effect.

```
1  boolean osbrdF(GNode lhs) {
2    int ei = i; DNode edn = dn;
3    altLoop: for (GNode alt = lhs.alt; alt != null; alt = alt.alt) {
4      i = ei; dn = edn;
5      GNode gn = alt.seq;
6      while (true) {
7        switch (gn.el.kind) {
8        case T: if (mt(gn)) {i++; ; gn = gn.seq; break;}
9                else continue altLoop; // failure; next alternate
10       case N: if (osbrdF(lhs(gn))) {gn = gn.seq; break;}
11               else continue altLoop; // failure; next alternate
12       case EPS: gn = gn.seq; break; // epsilon always matches
13       case END: du(alt.ni); return true; // end; update derivation
14       }}}
15   return false;
16 }
```

Interpreting with explicit stack management - osbrdE A general parser must handle non-determinism: during processing of some parser context we may identify more than one successor context that must be explored. OSBRD handles some (but not all) nondeterminisms through limited backtracking. Both the `osbrdG` and `osbrdF` implementations rely on the host language's function call mechanism to implictly manage a single stack of nonterminal instances. This is a fundamental weakness, because the sequence of parser contexts that may be examined is limited to those that conform to a last-in, first-out discipline. In a conventional procedural language such as Java without continuations, there is no way of loading the function call stack with a particular state.

A GLL parser works by saving parser contexts for later processing in a way that allows any context to be processed independently of the others, and not surprisingly it uses an explicit stack data structure to allow switching between contexts. Our next step (`osbrdE`) is to implement OSBRD using explicit stack management in a style that matches our GLL baseline implementation. At this stage we still only have a single stack for all contexts: this is only an OSBRD implementation.

As before, derivations are also developed within a linked list of `DNodes`, with the most recent derivation step held in variable `dn`.

Stack entries must contain all of the information in the stack frame for function `osbrdF()`: that is a return position in the grammar and the local variables `eI` and `eDN`. A suitable

Java declaration is
class SNode {GNode rN; **int** el; SNode next; DNode eDN;}

We model the stack with a linked list of SNodes and hold the stack top in variable sn. A perhaps unexpected side-effect of removing reliance on the runtime function call stack is that we may no longer use the unwinding of recursive calls to handle some aspects of backtracking. Instead, we explictly traverse the grammar representation, executing stack pops as we go, and this adds complexity to the match fail code at lines 7–13 in osbrdE.

Termination of a parse is triggered by popping the root element. If this occurs during back tracking (line 10) then the parse has failed and osbrdE() returns false. If the root element is popped whilst processing an END node (line 18) then we have reached the end of a production in S, and osbrdE() returns true: the caller must then check to see whether the entire string has been consumed.

```
1  boolean osbrdE() {
2    initialise();
3    while (true)
4    switch (gn.el.kind) {
5    case T:
6      if (mt(gn)) {i++; gn = gn.seq; break;}
7      else { while (true) { // On failure, backtrack
8        while (gn.el.kind != Kind.END) gn = gn.seq;
9        if (gn.alt.alt == null) { // No more productions; return
10         gn = ret();
11         if (sn == null) return false;} // No more stack frames; fail
12       else { // restore context
13         i = ((StackNode) sn).ei; dn =((StackNode) sn).edn;
14         gn = gn.alt.alt.seq; break;}}}
15     break;
16   case N: call(gn); break;
17   case EPS: gn = gn.seq; break;
18   case END:
19     du(gn.alt.ni); gn = ret(); if (sn == null) return true; break;
20 }}
```

3 A Baseline Interpreted GLL Parser - gllBL

The GLL algorithm takes the basic control flow patterns of our recursive parse functions and recasts them as a collection of separate threads that may be independently executed. The only data items required by a thread are the unchanging input and the four context elements identified earlier: a grammar node gn, an input index i, a top-of-stack node sn and a most-recent derivation step node dn.

Each parse thread may thus be uniquely characterised by a 4-tuple *descriptor*, declared in the style we have been using as: **class** Descriptor {GNode gn; **int** i; SNode sn; DNode dn;} A descriptor captures the starting context for a thread which is loaded into global variables gn, i, sn and dn. These values then evolve during execution of the thread, and often potential new starting contexts are identified for later processing.

At the outermost level, GLL is a worklist algorithm that selects descriptors from a collection of awaiting descriptors. During execution of a thread, new descriptors may be created: for instance when an instance of a nonterminal is encountered, the algorithm will create descriptors for each of that nonterminal's productions.

A GLL parse begins with a single awaiting descriptor (lhs(S), 0, gssRoot, null), that is the LHS grammar node for the start nonterminal, input index zero, a reference to the GSS base node and an empty derivation step. The parse terminates when the descriptor collection is empty.

Good performance of the algorithm relies on efficient implementation of the collection of descriptors, the collection of stacks and the collection of derivations. We delay consideration of those mechanisms until after we have examined the control flow aspects of the algorithm.

```
1  void gllBL() {
2    initialise();
3    nextDescriptor: while (dequeueDesc())
4    while (true) {
5      switch (gn.el.kind) {
6        case T: if (input[i] == gn.el.ei)
7                  {du(1); i++; gn = gn.seq; break;}
8                  else // abort thread on mismatch
9                    continue nextDescriptor;
10       case N: call(gn); continue nextDescriptor;
11       case EPS: du(0); gn = gn.seq; break;
12       case END: ret(); continue nextDescriptor;
13 }}}
```

This top level control flow is pleasingly simple. It has the same structure as for osbrdE, but is relieved of the complex backtracking code. This is because descriptors are created for each production by the call() function. When a terminal mismatch occurs, we can simply abandon the current thread by executing **continue** nextDescriptor. We do not need to backtrack to find the next viable alternate because it will already either have been processed, or be in the queue for processing. The call(), ret() and du() functions update the SPPF and GSS: we describe them below.

Thread management In a context free parser, a context once processed need never be processed again (although see notes on *contingent pops* below). The key to achieving a cubic upper bound on performance is to maintain a set of previously encountered descriptors descS in addition to a set of descriptors descQ awaiting processing. The call() function (line 9 above with definition below) then uses function queueDesc() to only load a descriptor to descQ if it has never been seen before. In this implementation we use the Java double-ended queue Deque for descQ. Since additions to descQ are guarded by checks on descS, descriptors can never appear more than once within descQ.

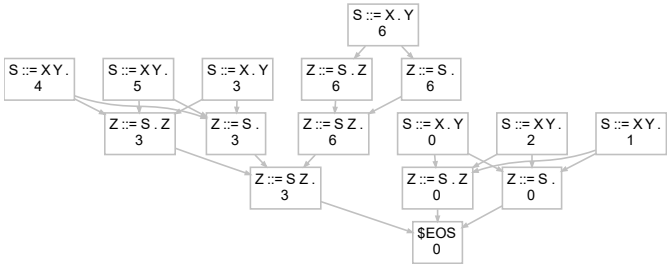
```
1  Set<Desc> descS; Deque<Desc> descQ;
2  GNode gn;
3  GSSN sn;
```

```
4  SPPFN dn;
5
6  void queueDesc(GNode gn, int i, GSSN gssN, SPPFN sppfN) {
7    Desc tmp = new Desc(gn, i, gssN, sppfN);
8    if (descS.add(tmp)) descQ.addFirst(tmp);
9   }
10
11 boolean dequeueDesc() {
12  Desc tmp = descQ.poll();
13  if (tmp == null) return false;
14  gn = tmp.gn; i = tmp.i; sn = tmp.sn; dn = tmp.dn;
15  return true;
16 }
```

The `dequeueDesc()` function polls `descQ` to remove the head element and unloads its fields into the global context variables, or returns `false` if no descriptors are available.

Stack management In the parsers discussed here, the purpose of the stack is to store a return point from the call to a nonterminal. A general parser starts with a single stack but new stacks may diverge from it. The shared bases of these stacks naturally need only be represented once, but Tomita [20] observed that when two stacks for a context free parser show the same stack top, their future behaviour will be indistinguishable. As a result, stacks may be *merged* together when they have the same top state, and the whole set of stacks may be represented by a directed acyclic graph called a Graph Structured Stack (GSS). (As an aside, the structure is perhaps more reasonably called a stack structured graph, but the term GSS is in common use.)

This visualisation shows the GSS that results from a GLL parse of Γ_5 and string *abcabc*. Each node has a grammar position (shown as a production with a dot after the nonterminal that was called) and a left index which is value of the input index at the point of call. The GSS is initially loaded with a *stack root* node labelled (EOS, 0) which ensures that all stacks are tied to a common base node.



Paths between the top and bottom node represent the result of successful matches to the whole string. The other paths result from parses which ultimately failed to match the entire string.

GSS nodes contain a grammar node which acts as the return point; an input index `i` which will be the leftmost index for the substring matched by this call; a set of out edges which references the stacks converging on this node; and a set of *contingent pops* which will be discussed below.

A suitable Java declaration is:

```
class GSSN {GNode gn; int i;
            Set<GSSE> edges; Set<SPPFN> pops;}
```

In a parser GSS edges are also labelled with the derivation node that was current at the time of their creation. A GSS edge also requires a reference to the destination of the edge. An appropriate Java declaration is:

```
class GSSE {GSSN dst; SPPFN sppfnode; }
```

The `call()` and `ret()` functions manage the evaluation of threads involving nonterminal instances. The basic requirement is to extend a stack on call, and to retrieve the return point in the grammar on return. In addition, a `call()` must queue descriptors for the start of each production in the called nonterminal (lines 18 and 19 below), and a `ret()` must check to see if we have reached the bottom of the stack, in which case we check for acceptance (lines 23–25).

```
1  Map<GSSN, GSSN> gss;
2  GSSN gssRoot;
3
4  GSSN gssFind(GNode gn, int i) {
5   GSSN gssN = new GSSN(gn, i);
6   if (gss.get(gssN) == null) gss.put(gssN, gssN);
7   return gss.get(gssN);
8  }
9
10 void call(GNode gn) {
11  GSSN gssN = gssFind(gn.seq, i);
12  GSSE gssE = new GSSE(sn, dn);
13  if (!gssN.edges.contains(gssE)) {
14   gssN.edges.add(gssE);
15   for (SPPFN rc : gssN.pops)
16    queueDesc(gn.seq, rc.ri, sn, sppfUpdate(gn.seq, dn, rc));
17  }
18  for (GNode p = rules(gn).alt; p != null; p = p.alt)
19   queueDesc(p.seq, i, gssN, null);
20 }
21
22 void ret() {
23  if (sn.equals(gssRoot)) {
24   if (accepting(gn)) accepted |= (i == input.length – 1);
25   return;
26  }
27  sn.pops.add(dn);
28  for (GSSE e : sn.edges)
29   queueDesc(sn.gn, i, e.dst, sppfUpdate(sn.gn, e.sppfnode, dn));
30 }
```

Perhaps the most subtle part of the GLL algorithm is the handling of *contingent pops*. When we perform a return at a GSS node `sn` we want the parse to continue at all of the return points encoded by the out-edges of `sn`. Edges can be added to a GSS node *after* a pop has occurred, and in such a case the earlier pop (or pops) must be applied to that new edge. Thus each GSS node contains a *pop set* which records pop actions (line 27) so that they may be applied to any new edges that subsequently may be added (lines 15–16 above).

Derivation updates Derivations, as discussed above, are represented using binarised SPPFs.

```
class SPPFN {GNode gn; int li; int ri;
             Set<SPPFPN> packNS;}
class SPPFPN {GNode gn; int pivot; SPPFN lC; SPPFN rC;}
```

A derivation update may be triggered in two ways: (i) the main gllBL algorithm adds elements via function `du()` when a terminal or an ϵ-rule is matched, and (ii) the `call()` and `ret()` functions add elements associated with nonterminals.

The SPPF is modelled as a set of SPPF nodes that carry a set of pack node children which in general will be updated during a parse. A Java quirk is that the standard `Set` API does not easily support updates to set elements. Now, Java sets are implemented as maps from elements to themselves, and so without loss of performance we can explicitly use such a map to allow retrieval of elements.

```
1  Map<SPPFN, SPPFN> sppf;
2
3  SPPFN sppfFind(GNode dn, int li, int ri) {
4    SPPFN tmp = new SPPFN(dn, li, ri);
5    if (!sppf.containsKey(tmp)) sppf.put(tmp, tmp);
6    return sppf.get(tmp);
7  }
8
9  SPPFN sppfUpdate(GNode gn, SPPFN ln, SPPFN rn) {
10   SPPFN ret = sppfFind(gn.el.kind == Kind.END ? gn.seq : gn,
11                        ln == null ? rn.li : ln.li,
12                        rn.ri);
13   ret.packNS.add(
14     new SPPFPN(gn, ln == null ? rn.li : ln.ri, ln, rn));
15   return ret;
16 }
17
18 void du(int width) {
19    dn = sppfUpdate(gn.seq, dn, sppfFind(gn, i, i + width));
20 }
```

Function `sppfFind()` returns an existing SPPF node, or adds a new SPPF node with empty pack node set.

Function `sppfUpdate()` takes a grammar node and two existing SPPF nodes and either identifies or creates the corresponding subtree.

Initialisation The core data structures are re-initialised for each parse: lines 2-3 create new, empty, data structures, lines 4-5 creates the base node for the GSS, line 6 initialises the global context variables and lines 7-8 load desriptors for each production of the start symbol.

```
1  void initialise() {
2    descS = new HashSet<>(); descQ = new LinkedList<>();
3    sppf = new HashMap<>(); gss = new HashMap<>();
4    gssRoot = new GSSN(grammar.endOfStringNode, 0);
5    gss.put(gssRoot, gssRoot);
6    i = 0; sn = gssRoot; dn = null;
7    for (GNode p = grammar.startNode.alt; p != null; p = p.alt)
8      queueDesc(p.seq, i, sn, dn);
9  }
```

4 Data Structure Optimisation

In *The Design and Evolution of C++* [19, p.211] Stroustroup wrote:

Many programs create and delete a large number of small objects of a few important classes … The allocation and deallocation of such objects with a general-purpose allocator can easily dominate the run time and sometimes the storage requirements of the programs.

The GLL algorithm is a perfect exemplar of this situation since it performs very little actual calculation: the algorithm is dominated by the conditional creation of small elements under control of tests based on simple integer comparisons. When built with the native Java APIs in an object oriented fashion as we have done in gllBL, runtime is dominated by allocation of small objects.

Stroustroup advocated the use of custom heap management for these kinds of programs, and claimed speedups of as much as an order of magnitude for applications in which the heap became heavily fragmented. GLL does not fragment the heap in that our core data structures (the GSS, the SPPF and the descriptor set) only ever grow: essentially the only deallocations that occur are the worklist elements that reference descriptors (which themselves are not deallocated). As a result we might not expect Stroustroup's fragmentation-driven factor ten speedup, but as we shall show in section 5, the scheme discussed in this section does yield speedup factors of 3–4 over the matching Java API implementation, whilst reducing memory demands by a factor of between 4 and 5.

The general approach is to (i) map all grammar objects including grammar positions onto the integers in a single sequence and (ii) to allocate elements sequentially from a pool of memory blocks. Essentially we eschew the use of Java objects, and make each grammar 'object' a small sequence of integers with contiguous memory addresses.

Enumeration of grammar elements The primitive objects manipulated by the GLL algorithm are: the end of string symbol, the terminals, the epsilon symbol, the nonterminals the elements not otherwise accounted for (such as `ALT` and `END`) and finally the grammar nodes. We arrange all of these in a sequence, and number them from zero. So, for Γ_1 the sequence is 0:EOS 1:a 2:b 3:x 4:y 5:z 6:EPS 7:S 8:X 9:ALT 10:END

The sequence then continues with the grammar nodes, the first of which is always the node which labels the GSS root (11 in this case), followed by the nodes representing the grammar rules as shown on page 1. Hence the node numbers there start at 12.

The purpose of this enumeration is to avoid the need to carry type information around: we can tell, for instance, if an element of the sequence is a terminal simply by checking that it is greater than zero and less than the sequence value for ϵ. Terminals appear first in the sequence because algorithms employing lookahead test input tokens against sets

of terminals, and a bit vector representation is more space efficient if the elements in the set have small values.

Pool based memory management gllBL maintains six sets: GSS nodes, GSS edges, pop elements, SPPF nodes, SPPF packed nodes and descriptors. Our main goal is to reduce the volume of calls to the system's heap allocation routines for these sets. We do this by explictly coding C-like structures for each set element and allocating them sequentially into a pool of memory blocks. The elements require 5, 5, 5, 7 and 6 integers each respectively, and our memory blocks are 2^{26} integers long (that is 256MByte each for 32-bit integers), so the number of calls to the system allocator is negligible.

Our memory references are 32-bit integers which can manage up to 4G integer locations, or 16Gbyte of memory. Since our blocks sizes are always a power of 2, we can use shift and mask operations to extract block number and address within block:

```
int poolGet(int index) {
  return
  pool[index >> poolAddressOffset][index & poolAddressMask];
}

void poolSet(int index, int value) {
  pool[index >> poolAddressOffset][index & poolAddressMask] =
  value;
}
```

The sets themselves are implemented as hash tables using separate chaining. The chain link is at offset zero in each of the structure elements. As always with hash tables, the effectiveness of the hash function and the table's load factor will dictate performance. We will illustrate the load factor impact in Section 5.

The GLL Hash Pool implementation – gllHP Our final implementation is gllBL modified to use this *Hash Pool* memory management scheme. The full code is available in the repository. This is the top level control flow: it is essentially identical to gllBL except that all context elements are integers which are used as references into the pool data; the lookup table `kindOf` is used to encode the grammar element's type; and advancing gn to the next sequence element simply requires gn to be incremented, since grammar nodes are numbered sequentially.

```
void gllHP() {
  initialise();
  nextDescriptor: while (dequeueDescriptor())
  while (true) {
    switch (kindOf[gni]) {
    case T: if (input[i] == elementOf[gni])
                {d(1); i++; gni++; break;}
                else continue nextDescriptor;
      case N: call(gni); continue nextDescriptor;
      case EPS: d(0); gni++; break;
      case END: ret(); continue nextDescriptor;
}}}
```

5 Performance Evaluation

Our intention is that gllBL and gllHP will provide reference performance data for future studies. To that end, the repository includes large corpora of Java 18 and Standard ML code with associated grammars, and we expect to expand those holdings in future. In this paper we restrict ourselves to six grammars and only five strings, but these strings are large — in the range 84–130 kBytes. We take this approach because we want to concisely present throughput and memory consumption figures for large real-world examples.

As well as gllBL and gllHP, we present some data for gllOpt which is an older implementation of GLL that is compiled, uses lookahead on both descriptor creation and pops and has the same hash pool data structure mechanisms as gllHP. gllOpt is not as tightly engineered for performance: it has support for trace messages and statistics gathering code, as well as support for some of the other GLL variants mentioned above. Results in all categories improve as we move from gllBL to gllHP to gllOpt; we would expect to see further improvements in gllOpt when a performance engineered version is available.

We use a range of programming language grammars: the ANSI-C grammar from the Kernighan and Ritchie textbook; the ANSI C++ grammar from the 1997 Public Review Document which underpinned C++98; the C# version 1.2 grammar and the Java Language Specification version 1 and 2 grammars. Larger studies using Java 18 and Standard ML may be found in the repository. The JLS2 grammar uses extended CFG constructs. We produced two variants by expanding closure using left or right recursion: we would expect the left recursive version to be more demanding of a GLL parser.

Test strings include the full source code for the parser generators RDP (C), GTB (C) and ART (C++) along with a Twitter client (C#) and multiple concatenations of an implementation of Conway's Game of Life (Java).

Data structure cardinalities We begin by looking at the cardinalities of the various GLL data structures (Table 2). Of course, gllBL and gllHP generate exactly the same cardinalities so we only compare gllBL to gllOpt which uses lookahead. As expected, the lookahead significantly reduces the number of descriptors and indeed the cardinalities of the other sets too. It is clear from this table that the amount of ambiguity encountered drives the size of these sets. That is as we should expect: were the grammars LL(1) then there would be no nondeterminism and we might hope to approach a linear cost. GLL performance is worse case cubic in the length of the input, and as the level of nondeterminism goes up we would expect to move towards that cubic bound. Thus the rightmost column is important. We can see that ANSI C++ is much more ambiguous than ANSI C when run on gtbSrc and rdpSrc.

Throughput Speed measurements in Table 3 were made using a Dell XPS 15 9510 laptop with 16GByte of installed

String	Heap BL	Heap HP	Factor	Pool
artSrc	3,165,999,856	545,260,192	5.81	562,135,472
gtbSrc	3,637,287,192	817,889,952	4.45	666,862,111
rdpSrc	2,814,125,624	545,260,192	5.16	506,684,413

Table 1. Heap utilisation for ANSI C++

memory and an Intel Core i7-11800H eight-core processor running at 2.3GHz. The experiments were run from the command line under Microsoft Windows 10 Enterprise version 10.0.19042 using Oracle's Java HotSpot(TM) 64-Bit Server VM (build 14.0.2+12-46, mixed mode, sharing).

The nanosecond timing routines in the Java System API do not accurately reflect computational load in multicore systems and can even return negative values. As a result we used the System.currentTimeMillis() to measure runtimes even though its resolution is only around 0.03 seconds; for each experiment we made 10 runs and report here the mean run time in milliseconds.

We deliberately disabled the resizing of hash tables in gllHP and gllOpt, and set the size of the tables to be twice that required to handle the gtbSrc input when run with the ANSI-C grammar (the highlighted first line). The throughput in tokens per second ranges from 113,458 for JLS2 right down to 6,701 for gtbSrc and ANSI C++. Looking at the cardinailities table, it is clear that much of this variation arises from the hash table load factor.

Hash table load factor To further investigate this effect, Table 4 shows partial histograms of bucket occupancy in the hash tables. The ANSI C++ examples show significant hash table congestion, with in one case more than 10% of the bucket lists having four or more elements.

Heap utilisation We wanted to get a measure of memory consumption, comparing gllBL to gllHP. It is quite difficult to measure heap utilisation in Java but we can approximate it by asking for the free heap size before and after a run. Table 1 shows the results for the three pieces of C/C++ source code running with the ANSI C++ grammar. As a consistency check we have also included (in column Pool) the *computed* size of all of the data structure elements which we can derive from the known data structure cardinalities and the size of their elements. These figures should clearly be treated with great caution, but they do indicate that a factor four reduction in memory footprint is achieved with the HashPool implementation. This is unsurprising.

6 Potential Future Work on Optimisation

GLL was introduced at the 2009 LDTA workshop [10]. A variety of improvements to the basic GLL algorithm have been reported since then which we summarise here along with some ideas that have not (as far as we know) appeared yet in the literature. In future work, we shall present implementations of some of these ideas in the same style as our baseline (gllBL) and hashpool (gllHP) implementations so as to produce a consistent evaluation of their strengths and weaknesses. In what follows, each paragraph is a separate optimisation opportunity.

Thread management gllBL and gllHP do not use lookahead. Wherever there is a break in control flow, we can reduce the number of descriptors being created by using lookahead to suppress descriptors for threads that will immediately terminate: the lookahead effectively allows us to pre-compute whether the first match operation in a thread will fail. In fact the effect can be quite large, since where we have rules of the form $X \rightarrow \alpha Y \beta \quad Y \rightarrow Z\gamma \quad Z \rightarrow \delta$ and `gn` corresponds to $X \rightarrow \alpha \cdot Y\beta$, a failing lookahead test will suppress descriptor creation for both Y and Z. We can also use lookahead in the `ret()` function to suppress descriptor creation when the current input symbol is not in the FOLLOW set of the left hand side of the current rule. We give some initial results from lookahead implementations in the next section.

Scott McPeak reported on *Elkhound* [8] which is a table-driven generalised LR parser which runs deterministically on those parts of the table which are context free. In GLL, descriptors also only need to be created when true nondeterminism is detected. The conditions under which descriptor creation may be suppressed are yet to be fully studied, but we note that a combination of lookahead and FIFO-style short circuiting effectively ensure deterministic execution for calls to rules which are LL(1), since the alternate rules will have disjoint FIRST sets so at most one new thread can exist.

The presentation here uses the language of threads to describe GLL control flow, and it is natural to wonder whether a truly multi-threaded implementation running on a modern multi-core processor would demonstrate significant speedups. Initial experiments using Java threads have not been encouraging, which is perhaps to be expected since as we have already noted, GLL performs very little actual computation: nearly all actions are conditional data structure updates, and those data structures are global to all threads. Hence the ratio of inter-thread communication to in-thread computation is high. However, our experience with gllHP has shown that the general purpose Java libraries cannot compete with a tuned implementation, and so we might imagine that there are highly tuned approaches to distributing the GLL algorithm over multiple processors that might be worthwhile. The current ubiquity of multi-core hardware makes this an attractive goal for further research.

In principle, the number of contingent pop actions may be reduced by choosing a suitable execution order for the descriptors. In practice it is not clear whether there are significant gains to be had since the work associated with the pop has to be performed under any ordering, and the overhead of scanning the pop list for each GSS node is not great. There may be cache effects that can be exploited if we achieve

Grammar	String	Tokens	Mode	Descriptors	GSS Node	GSS Edge	Pops	Symbol	Packed	Ambig
ANSI C	gtbSrc	36,828	Opt	4,178,345	564,437	2,042,843	559,859	297,677	261,401	515
			gllBL	6,578,603	946,975	2,989,166	776,934	881,128	829,463	526
ANSI C	rdpSrc	26,552	Opt	3,122,638	417,204	1,510,486	425,730	222,206	195,799	138
			gllBL	4,803,532	699,089	2,219,720	567,128	637,043	602,230	139
ANSI C++	artSrc	36,445	Opt	9,493,519	1,036,075	4,755,333	874,868	473,257	475,542	27,362
			gllBL	20,250,528	2,496,038	8,069,723	1,227,578	1,310,876	1,306,193	49,682
ANSI C++	gtbSrc	36,828	Opt	13,061,222	1,270,903	6,392,785	1,110,400	561,139	562,843	26,081
			gllBL	24,091,341	2,647,731	9,650,268	1,430,990	1,531,742	1,513,670	50,039
ANSI C++	rdpSrc	26,552	Opt	9,687,071	942,742	4,709,390	841,963	425,385	426,291	18,925
			gllBL	18,294,156	2,056,206	7,427,350	1,061,367	1,142,989	1,125,019	35,806
C# 1.2	twitter	33,841	Opt	2,024,014	443,304	1,056,916	390,990	255,343	225,052	2,670
			gllBL	4,659,342	1,140,430	2,170,525	555,474	639,254	604,999	11,460
JLS1	life	36,976	Opt	2,302,532	505,179	1,249,154	402,501	260,377	223,401	0
			gllBL	4,175,967	883,950	1,948,492	599,751	710,803	655,277	0
JLS2 left	life	36,976	Opt	858,335	262,104	395,704	316,328	343,729	313,678	24,725
			gllBL	3,475,876	699,565	1,022,346	462,108	745,514	620,736	30,650
JLS2 right	life	36,976	Opt	783,455	266,303	375,252	296,777	336,505	305,478	23,500
			gllBL	3,196,289	642,062	822,362	413,005	719,486	582,283	30,650

Table 2. Effect of lookahead on data structure cardinalities

Grammar	String	Characters	Tokens	CPU seconds			Speedup		Throughput tokens s^{-1}		
				BL	HP	Opt	HP	Opt	BL	HP	Opt
ANSI C	**gtbsrc**	**117,557**	**36,828**	**4.14**	**1.32**	**0.85**	**3.13**	**4.90**	**8,888**	**27,801**	**43,537**
ANSI C	rdpsrc	84,778	26,552	2.76	0.92	0.63	3.00	4.39	9,605	28,833	42,126
ANSI C++	artsrc	118,922	36,445	14.67	5.30	3.87	2.76	3.79	2,485	6,870	9,425
ANSI C++	gtbsrc	117,557	36,828	18.82	6.92	5.50	2.72	3.42	1,957	5,321	6,701
ANSI C++	rdpsrc	84,778	26,552	13.95	4.64	3.84	3.01	3.64	1,903	5,728	6,923
C# 1.2	twitter	131,323	33,841	3.38	0.84	0.67	4.02	5.06	9,998	40,239	50,554
JLS 1	life	125,594	36,976	2.67	0.81	0.62	3.31	4.30	13,871	45,899	59,639
JLS2 left	life	125,594	36,976	1.89	0.56	0.35	3.40	5.46	19,543	66,539	106,651
JLS2 right	life	125,594	36,976	1.75	0.49	0.33	3.58	5.36	21,182	75,802	113,458

Table 3. Throughput using hash table tuned for load factor 2 on ANSI C parsing gtbsrc

spatial locality of actions, and that suggests that processing descriptors in a first-in, first-out manner might be advantageous. In the gllBL implementation we have used a double ended queue so as to explore such effects. gllHP uses a stack to hold the descriptors, and thus is FIFO.

If we are using FIFO descriptor scheduling then we can short circuit the enqueue/dequeue operations for the final descriptor in a call action, since once loaded it will be immediately unloaded, so we might as well directly load the context variables.

Derivation representation The binarised SPPF as described here has obvious redundancies. In the binarisation scheme above, the last element of each production has an intermediate node parent with only one child, and this can be suppressed with the element directly attached as the left child of the preceding intermediate node, so in general we would only need $k-2$ binarisation nodes for sequences of length $k(k>2)$, and no binarisation node for a sequence of length 1 or 2.

Terminal nodes themselves can be omitted since the parent pack node and its parent symbol node contain the indices into the string for that terminal.

Pack nodes are only required where there is ambiguity, and as we shall see in Table 2 below, for current programming language grammars the proportion of symbol and intermediate nodes that are ambiguous is small, thus large potential savings are possible.

If pack nodes are labelled with the left and right indices from their parent symbol, then they contain all of the information required to encode the derivation forest and the

Grammar	String	Mode	Pool bytes	Pool/tok	1	2	3	≥ 4	% ≥ 4
ANSI C	gtbSrc	Opt	119,434,167	3,243	5,249,823	1,065,880	139,023	25,498	0.39
		gllHP	191,663,728	5,204	7,937,594	1,960,621	320,110	44,516	0.43
ANSI C	rdpSrc	Opt	89,192,035	3,359	4,235,001	701,905	69,906	11,063	0.22
		gllHP	140,115,278	5,277	6,652,636	1,193,682	143,439	14,323	0.18
ANSI C++	artSrc	Opt	266,870,915	7,323	7,738,658	2,994,531	757,995	257,778	2.19
		gllHP	560,677,098	15,384	8,812,860	5,823,056	2,723,507	1,381,572	7.37
ANSI C++	gtbSrc	Opt	364,085,364	9,886	8,237,055	4,063,565	1,352,693	581,128	4.08
		gllHP	665,091,529	18,059	8,216,549	6,298,134	3,450,289	2,182,890	10.83
ANSI C++	rdpSrc	Opt	270,071,141	10,171	7,638,987	2,995,175	762,042	260,622	2.24
		gllHP	505,374,437	19,033	9,034,932	5,393,298	2,293,843	1,019,367	5.75
C#	twitter	Opt	60,884,465	1,799	3,495,829	397,677	30,893	2,880	0.07
		gllHP	138,587,616	4,095	6,672,188	1,269,676	163,048	17,001	0.21
JLS1	life	Opt	68,910,630	1,864	3,820,275	485,504	43,484	5,234	0.12
		gllHP	125,525,127	3,395	6,319,822	1,108,786	128,263	12,761	0.17
JLS2 left	life	Opt	29,385,111	795	2,204,270	132,437	6,585	242	0.01
		gllHP	102,413,841	2,770	5,227,440	761,699	82,727	6,671	0.11
JLS2 right	life	Opt	27,315,451	739	2,102,990	121,011	5,915	251	0.01
		gllHP	93,771,492	2,536	4,848,491	651,856	65,324	5,844	0.10

Table 4. Effect of suboptimal hash table load factor

parent symbol and intermediate nodes are redundant. This is the basis if the *Binary Subtree Representation* (BSR) described in [15]. As well as reducing memory requirements, this approach reduces derivation updates to set-addition of BSR elements which simplifies and speeds up the operation.

The use of the Kleene and Positive closures can act as hints to the parser to use iteration rather than recursion which may yield performance improvements. Extended context free grammars offer opportunities to both reduce stack activity and compress derivations. Extending a GLL *recogniser* to handle extended CFG constructs is straightforward, but correctly embedding all derivations in the SPPF for a GLL *parser* requires care: a complete scheme is given in [13] in which extended constructs are referred to as 'bracketed' constructs.

7 Concluding remarks

We have discussed two reference implementations of GLL: gllBL which uses standard Java API objects to implement the core data structures and gllHP which uses explicit memory management.

The key question is whether GLL is a plausible engineering option compared to classical approaches. There is no question that the approach is *very* expensive compared to the near-deterministic techniques developed in the 1970s: in some cases gllHP needs as much as 8kbytes of memory per input character. However even for very long inputs of over 100kByte characters parsed using the ANSI C++ grammar with its many ambiguities, gllHP needs no more than 0.8Gbyte of memory which is only 5% of the memory on a typical modern 16GByte laptop computer, thus these gargantuan memory demands are not a practical problem.

Throughput is also much less than for a classical parser, but similarly manageable on modern hardware. We propose the informal metric *Good Enough for Gnu* (GEG) as a threshold test for utility. We imagine that the current GNU C compiler is re-engineered with a GLL parser. Assuming that the existing classical parser takes negligible resources, a parser is GEG if it slows GNU C down by no more than 10%.

The underlying source code for the gtbSrc string comprises 996,776 characters which when compiled with GNU C++ in its default mode requires 10.5s, and when compiled with -Ofast, 18.8s. Hence we would like our general parsers to process this string in at most 1–2 seconds.

gtbHP processes gtbSrc in 1.35s using the ANSI C grammar, and 7.02s for the much more challenging ANSI C++ grammar. However gtbHP is a baseline implementation (albeit with efficient memory management), is interpreted and is written in Java. Informal experiments indicate that adding in lookahead and using a compiled parser will improve throughput by a factor of around two; that converting the code to ANSI C will produce another factor two improvement; and that setting the hash table load factors appropriately for ANSI C++ (rather than ANSI-C as here) may give a further factor 1.5 improvement.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., USA.

[2] GNU. 2023. New C parser. https://gcc.gnu.org/wiki/New_C_Parser. Accessed: 2023-07-06.

[3] GNU. 2023. New C parser [patch]. https://gcc.gnu.org/legacy-ml/gcc-patches/2004-10/msg01969.html. Accessed: 2023-07-06.

[4] Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. 2016. Practical, General Parser Combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (St. Petersburg, FL, USA) *(PEPM '16)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2847538.2847539

[5] Adrian Johnstone and Elizabeth Scott. 2011. Modelling GLL Parser Implementations. In *Software Language Engineering*, Brian Malloy, Steffen Staab, and Mark van den Brand (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 42–61.

[6] Adrian Johnstone and Elizabeth Scott. 2015. Principled software microengineering. *Science of Computer Programming* 97 (2015), 64–68. https://doi.org/10.1016/j.scico.2013.11.018 Special Issue on New Ideas and Emerging Results in Understanding Software.

[7] Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. 168–177. https://doi.org/10.1109/SCAM.2009.28

[8] Scott McPeak and George C. Necula. 2004. Elkhound: A Fast, Practical GLR Parser Generator. In *Compiler Construction*, Evelyn Duesterwald (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–88. https://doi.org/10.1007/978-3-540-24723-4_6

[9] Thomas J. Pennello. 1986. Very Fast LR Parsing. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction* (Palo Alto, California, USA) *(SIGPLAN '86)*. Association for Computing Machinery, New York, NY, USA, 145–151. https://doi.org/10.1145/12276.13326

[10] Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010), 177–189. https://doi.org/10.1016/j.entcs.2010.08.041 Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[11] Elizabeth Scott and Adrian Johnstone. 2013. GLL parse-tree generation. *Science of Computer Programming* 78, 10 (2013), 1828–1844. https://doi.org/10.1016/j.scico.2012.03.005 Special section on Language Descriptions Tools and Applications (LDTA'08 & '09).

[12] Elizabeth Scott and Adrian Johnstone. 2016. Structuring the GLL parsing algorithm for performance. *Science of Computer Programming* 125 (2016), 1–22. https://doi.org/10.1016/j.scico.2016.04.003

[13] Elizabeth Scott and Adrian Johnstone. 2018. GLL syntax analysers for EBNF grammars. *Science of Computer Programming* 166 (2018), 120–145. https://doi.org/10.1016/j.scico.2018.06.001

[14] Elizabeth Scott and Adrian Johnstone. 2019. Multiple Lexicalisation (a Java Based Study). In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering* (Athens, Greece) *(SLE 2019)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/3357766.3359532

[15] Elizabeth Scott, Adrian Johnstone, and L. Thomas van Binsbergen. 2019. Derivation representation using binary subtree sets. *Science of Computer Programming* 175 (2019), 63–84. https://doi.org/10.1016/j.scico.2019.01.008

[16] Elizabeth Scott, Adrian Johnstone, and Robert Walsh. 2023. Multiple Input Parsing and Lexical Analysis. *ACM Trans. Program. Lang. Syst.* 45, 3, Article 14 (jul 2023), 44 pages. https://doi.org/10.1145/3594734

[17] Daniel Spiewak. 2023. gll-combinators. https://index.scala-lang.org/djspiewak/gll-combinators. Accessed: 2023-09-05.

[18] StackOverflow. 2023. Are GCC and Clang parsers really handwritten? https://stackoverflow.com/questions/6319086/are-gcc-and-clang-parsers-really-handwritten. Accessed: 2023-07-06.

[19] Bjarne Stroustrup. 1995. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., USA.

[20] Masaru Tomita. 1985. An Efficient Context-Free Parsing Algorithm for Natural Languages. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2* (Los Angeles, California) *(IJCAI'85)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 756–764.

[21] L. Thomas van Binsbergen, Elizabeth Scott, and Adrian Johnstone. 2018. GLL Parsing with Flexible Combinators. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering* (Boston, MA, USA) *(SLE 2018)*. Association for Computing Machinery, New York, NY, USA, 16–28. https://doi.org/10.1145/3276604.3276618

Received 2023-07-07; accepted 2023-09-01