Modelling GLL Parser Implementations

Adrian Johnstone and Elizabeth Scott

Royal Holloway, University of London, Egham, Surrey, TW20 0EX, UK
{a.johnstone,e.scott}@rhul.ac.uk

Abstract. We describe the development of space-efficient implementations of GLL parsers, and the process by which we refine a set-theoretic model of the algorithm into a practical parser generator that creates practical parsers. GLL parsers are recursive descent-like, in that the structure of the parser's code closely mirrors the grammar rules, and so grammars (and their parsers) may be debugged by tracing the running parser in a debugger. While GLL *recognisers* are straightforward to describe, full GLL parsers present technical traps and challenges for the unwary. In particular, naïve implementations based closely on the theoretical description of GLL can result in data structures that are not practical for grammars for real programming language grammars such as ANSI-C. We develop an equivalent formulation of the algorithm as a high-level set-theoretic model supported by table-based indices, in order to then explore a set of alternative implementations which trade space for time in ways which preserve the cubic bound.

Keywords: GLL parsing, general context-free parsing, implementation spaces, time-space tradeoffs.

1 The Interaction between Theory and Engineering

In Computer Science, the theoretician is mostly concerned with the correctness and asymptotic performance of algorithms whereas the software engineer demands 'adequate' time complexity on typical data coupled to memory requirements that do not cause excessive swapping. The theoretician's concerns are independent of implementation but the engineer's concerns are dominated by it and so the two communities do not always communicate well. As a result, our discipline has not yet achieved the comfortable symbiosis displayed by, for example, theoretical and experimental physicists.

The dominant characteristic of theoretically-oriented presentations of algorithms is *under specification*. It is fundamental practice for a theoretician to specify only as much as is required to prove the correctness of the results because this gives those results the widest possible generality, and thus applicability.

For the software engineer, under specification can be daunting: they must choose data structures that preserve the asymptotic performance demonstrated by the theoretical algorithm, and sometimes the performance expectations are only implicit in the theoretician's presentation. For instance, theoretical algorithms will often use sets as a fundamental data type. To achieve the lowest

B. Malloy, S. Staab, and M. van den Brand (Eds.): SLE 2010, LNCS 6563, pp. 42–61, 2011.

asymptotic bounds on performance the algorithm may need sets that have constant lookup time (which suggests an array based implementation) or sets whose contents may be iterated over in time proportional to their cardinality (which suggests a linked list style of organisation). The engineer may in fact be less concerned by the asymptotic performance than the average case performance on typical cases, and so a hash-table based approach might be appropriate. These implementation issues may critically determine the take up of a new technique because in reworking the algorithm to accommodate different data representations, the implementer may introduce effects that make the algorithm incorrect, slow or impractically memory intensive in subtle cases.

This paper is motivated by our direct experience of the difficulties encountered when migrating a theoretically attractive algorithm to a practical implementation even within our own research group, and then the further difficulties of communicating the results and rationale for that engineering process to collaborators and users.

The main focus of this paper is a modelling case study of our GLL generalised parsing algorithm [6] which yields cubic time parsers for all context free grammars. Elsewhere we have presented proofs of correctness and asymptotic bounds on performance along with preliminary results that show excellent average case performance on programming language grammars. It is clear, however, that the theoretical presentations have proved somewhat difficult for software engineers, who may find the notation opaque or some of the notions alien, and who may miss some of the critical assumptions concerning data structures which are required to have constant lookup time. Direct implementation of these structures consumes cubic memory and thus more subtle alternatives are required. In this paper we shall explicitly address the motivation for our choice of high level data structures, and explain how we migrate a naïve version to a production version by successive refinement. Our goal is to describe at the meta-level the process by which we refine algorithm implementations.

We view this as a modelling process. Much of the model-driven engineering literature is concerned with programming in the large, that is the composition of complete systems from specifications at a level of abstraction well away from the implementation platform, potentially allowing significant interworking and reuse of disparate programming resources. This paper is focused on programming in the small. We use a specification language that avoids implementation details of the main data structures, and then use application specific measures to refine the specification into an implementation with optimal space-time tradeoff. We do this in a way that lends itself to automation, holding out the prospect of (semi-)automatic exploration of the implementation space. It is worth investing this effort because we are optimising a *meta*-program: our GLL parser generator generates GLL parsers, and every parser user will benefit from optimisations that we identify.

Our models are written in LC, a small object-oriented language with an extremely simple type system based on enumerations and tables. Our goal is to develop a notation that is comfortable for theoretical work from which

implementations may be directly generated, and which also allows tight specification of memory idioms so that the generated implementations can be tuned using the same techniques that we presently implement manually, but with much reduced clerical overhead.

We begin with a discursive overview of GLL and then describe some aspects of the LC language. We give an example GLL parser written in LC and explain its operation. We then show how memory consumption may be significantly reduced without incurring heavy performance penalties. We finish by discussing the potential to semi-automatically explore the space of refined implementations in search of a good time-space trade off for particular kinds of inputs.

2 General Context Free Parsing and the GLL Algorithm

Parsing is possibly one of the most well studied problems in computer science because of the very broad applicability of parsers, and because formal language theory offers deep insights onto the nature of the computational process. Translators such as compilers take a program written in a source (high level) language and output a program with the same meaning written in a target (machine) language. The syntax of the source language is typically specified using a context free grammar and the meaning, or semantic specification, of a language is typically built on the syntax. Thus the first stage of compilation is, given an input program, to establish its syntactic structure. This process is well understood and there exist linear algorithms for large classes of grammars and cubic algorithms that are applicable to all context free grammars.

Formally, a *context free grammar* (CFG) consists of a set $\mathbf{N}$ of non-terminal symbols, a set $\mathbf{T}$ of terminal symbols, an element $S \in \mathbf{N}$ called the start symbol, and a set of grammar rules of the form $A ::= \alpha$ where $A \in \mathbf{N}$ and α is a string in $(\mathbf{T} \cup \mathbf{N})^*$. The symbol ϵ denotes the empty string, and in our implementations we will use # to denote ϵ. We often compose rules with the same left hand sides into a single rule using the alternation symbol, $A ::= \alpha_1 \mid \ldots \mid \alpha_t$. We refer to the strings α_j as the *alternates of A*.

We use a grammar Γ to define a language which is a set of strings of terminals. We do this by starting with the start symbol and repeatedly replacing a nonterminal with one of its alternates until a string containing no nonterminals is obtained. A *derivation step* is an expansion $\gamma A\beta \Rightarrow \gamma\alpha\beta$ where $\gamma, \beta \in (\mathbf{T} \cup \mathbf{N})^*$ and $A ::= \alpha$ is a grammar rule. A *derivation* of τ from σ is a sequence $\sigma \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \tau$, also written $\sigma \overset{*}{\Rightarrow} \tau$ or, if $n > 0$, $\sigma \overset{+}{\Rightarrow} \tau$. The language defined by a CFG Γ is the set of all strings u of terminals which can be derived from the start symbol, S, of Γ. Parsing is the process of determining, given a string u, some or all of the derivations $S \overset{*}{\Rightarrow} u$.

Of the linear parsing techniques perhaps the most widely used is the LR-table driven stack based parser [3]. For the class of grammars which admit LR-parsers the technique is straightforward to implement. However, the class of grammars does not include any grammars for 'real' programming languages and as a result implementations 'modify' the technique to extend its applicability. It can be

hard to reason about the correctness or the subtle behaviour of the resulting implementation.

An alternative is to use a general technique such as Earley[2], CYK[10] or GLR[8], [5]. In worst case Earley algorithms are cubic, CYK requires the grammar to be rewritten in 2-form and standard GLR algorithms are unbounded polynomial order, although the typical performance of GLR algorithms approaches linear and a cubic version has been developed [7]. These general algorithms are used in the natural language community but have had relatively limited take up within mainstream computer science. This is, at least to some extent, because they are hard for many implementers to understand and their implementation needs to be done with a great deal of care to get acceptable space and runtime performance. For example, ASF+SDF [9] uses Farshi's version of GLR [4] which achieves correctness in a rather brute force way and hence acquires a performance cost, and Bison includes a GLR mode [1] which does not employ the full details of the GLR approach and hence cannot parse an input of length 20 for some highly ambiguous grammars.

Recently we have introduced the general GLL parsing technique [6], which is based on the linear recursive descent paradigm. Recursive descent (RD) is an attractive technique because the parser's source program bears a close relationship to the grammar of the language and hence is easy to reason about, for instance by tracing the code in a debugger. However, the class of grammars to which RD can be applied is very limited and many extensions have been implemented which use either full or limited backtracking. Full backtracking can result in exponential runtime and space requirements on some grammars and limited backtracking will fail to parse some grammars correctly. GLL models full backtracking by maintaining multiple process threads. The worst-case cubic complexity is achieved by using a Tomita-style *graph structured stack* (GSS) to handle the function call stacks and left recursion (a fundamental problem for RD parsers) is handled with loops in this graph. (A non-terminal A is *left recursive* if there is a string μ such that $A \stackrel{+}{\Rightarrow} A\mu$, and a grammar is left recursive if it has a left recursive nonterminal.)

The GLL technique is based on the idea of traversing the grammar, Γ, using an input string, u, and we have two pointers one into Γ and one into u. We define a *grammar slot* to be a position immediately before or after any symbol in any alternate. These slots closely resemble LR(0) items and we use similar notation, $X ::= x_1 \dots x_i \cdot x_{i+1} \dots x_q$ denotes the slot before the symbol x_{i+1}. The grammar pointer points to a grammar slot. The input pointer points to a position immediately before a symbol in the input string. For $u = x_1 \dots x_p$, i is the position immediately before x_{i+1} and p is the position immediately before the end-of-string symbol, which we denote by \$.

A grammar is traversed by moving the grammar pointer through the grammar. At each stage the grammar pointer will be a slot of the form $X ::= \alpha \cdot x\beta$ or $X ::= \alpha\cdot$ and the input pointer will be an input position, i. There are then four possible cases: (i) If $x = a_{i+1}$ the grammar pointer is moved to the slot $X ::= \alpha x \cdot \beta$ and the input pointer is moved to $i + 1$. (ii) If x is a nonterminal

A then the 'return' slot, $X ::= \alpha x \cdot \beta$ is pushed onto a stack, the grammar pointer is moved to some slot of the form $A ::= \cdot\gamma$ and the input pointer is unchanged. (iii) If the grammar pointer is $X ::= \alpha\cdot$ and the stack is nonempty, the slot $Y ::= \delta X \cdot \mu$ which will be on the top of the stack is popped and this becomes the grammar pointer. (iv) Otherwise if grammar pointer is of the form $S ::= \tau\cdot$ and the input pointer is at the end of the input a successful traversal is recorded, else the traversal is terminated in failure. Initially the grammar pointer is positioned at a slot $S ::= \cdot\alpha$, where S is the start symbol, and the input pointer is 0.

Of course we have not said how the slot $A ::= \cdot\gamma$ in case (ii) is selected. In the most general case this choice is fully nondeterministic and there can be infinitely many different traversals for a given input string. We can reduce the nondeterminism significantly using what we call *selector sets*. For a string α and start symbol S we define $\text{FIRST}_{\mathbf{T}}(\alpha) = \{t | \alpha \stackrel{*}{\Rightarrow} t\alpha'\}$, and $\text{FIRST}(\alpha) = \text{FIRST}_{\mathbf{T}}(\alpha) \cup \{\epsilon\}$ if $\alpha \stackrel{*}{\Rightarrow} \epsilon$ and $\text{FIRST}(\alpha) = \text{FIRST}_{\mathbf{T}}(\alpha)$ otherwise. We also define $\text{FOLLOW}(\alpha) = \{t | S \stackrel{*}{\Rightarrow} \tau\alpha t\mu\}$ if $\alpha \neq S$ and $\text{FOLLOW}(S) = \{t | S \stackrel{*}{\Rightarrow} \tau S t\mu\} \cup \{\$\}$.

Then for any slot $X ::= \alpha \cdot \beta$ we define $select(X ::= \alpha \cdot \beta)$ to be the union of the sets $\text{FIRST}(\beta x)$, for each $x \in \text{FOLLOW}(X)$, and we can modify the traversal case (ii) above to say (ii) If x is a nonterminal A and there is a grammar slot $A ::= \cdot\gamma$ where $a_i \in select(\gamma)$ then the 'return' slot, $X ::= \alpha x \cdot \beta$ is pushed onto a stack, the grammar pointer is moved to the slot $A ::= \cdot\gamma$ and the input pointer is unchanged. The initial grammar pointer is also set to a slot $S ::= \cdot\alpha$ where $a_0 \in select(A ::= \cdot\alpha)$.

Whilst the use of selector sets can significantly reduce the number of possible choices at step (ii), in general there will still be more than one qualifying slot $A ::= \cdot\gamma$ and, in some cases, infinitely many traversals. GLL is a technique designed to cope with this in worst-case cubic time and space.

At step (ii), instead of continuing the traversal each possible continuation path is recorded in a *context descriptor* and ultimately pursued. We would expect a descriptor to contain a slot, an input position and a stack. Then, at step (ii), for each slot $A ::= \cdot\gamma$ such that $a_i \in select(A ::= \cdot\gamma)$ a descriptor is created with that slot, the current stack onto which the return slot is pushed and input position i. A descriptor (L, s, i) is 'processed' by restarting the traversal with the grammar pointer at the slot L, s as the stack and input pointer at i. There are potentially infinitely many descriptors for a given input string because there are potentially infinitely many stacks. The solution, introduced by Tomita for his initial version of the GLR technique, is to combine all the stacks into a single graph structure, merging the lower portions of stacks where they are identical and recombining stack tops when possible. At the heart of the GLL technique are functions for building this graph of merged stacks, which we call the GSS. Instead of a full stack, descriptors then contain a node of the GSS which corresponds to the top of its associated stack(s), thus one descriptor can record several partial traversals provided they restart at the same grammar and input positions.

So far we have addressed only the recognition of a string; we want to capture the syntactic structure to pass on to later stages in the translation process. A

common method for displaying syntactic structure is to use a *derivation tree*: an ordered tree whose root is labelled with the start symbol, leaf nodes are labelled with a terminal or ϵ and interior nodes are labelled with a non-terminal, A say, and have a sequence of children corresponding to the symbols in an alternate of A. This is a derivation tree for $a_1 \ldots a_n$ if the leaf nodes are labelled, from left to right, with the symbols $a_1, \ldots, a_n$ or ϵ. The problem is that for ambiguous grammars one string may have very many different syntactic structures and so any efficient parsing technique must use an efficient method for representing these. The method used by GLR parsers is to build a *shared packed parse forest* (SPPF) from the set of derivation trees. In an SPPF, nodes from different derivation trees which have the same tree below them are shared and nodes which correspond to different derivations of the same substring from the same non-terminal are combined by creating a packed node for each family of children. The size of such an SPPF is worst-case unbounded polynomial, thus any parsing technique which produces this type of output will have at least unbounded polynomial space and time requirements. The GLL technique uses SPPF building functions that construct a binarised form of SPPF which contains additional *intermediate* nodes. These nodes group the children of packed nodes in pairs from the left so that the out degree of a packed node is at most two. This is sufficient to guarantee the SPPF is worst-case cubic size. In detail, a binarised SPPF has three types of nodes: symbol nodes, with labels of the form (x, j, i) where x is a terminal, nonterminal or ϵ and $0 \leq j \leq i \leq n$; intermediate nodes, with labels of the form (t, j, i); and packed nodes, with labels for the form (t, k), where $0 \leq k \leq n$ and t is a grammar slot. We shall call (j, i) the *extent* (j, i are the left and right extents respectively) of the SPPF symbol or intermediate node and k the *pivot* of the packed node.

3 The LC Specification Language

LC is a small object oriented language which provides only a single primitive data type (the enumeration) and a single data structuring mechanism (the table). LC is designed to allow high-level descriptions of set-theory based algorithms whilst also allowing quite fine grained specification of the implementation in terms of the way the algorithm's objects are to be represented in memory. In this respect, LC is an unusual language with elements of both high level specification languages and very low level assembler-like languages. At present, LC is a (mostly) paper notation which we use here to describe data structure refinements. Our intention is that an LC processor will be constructed which can generate executable programs written in C++, Java and so on as well as LaTeX renderings of our algorithms in the style of [6]. (We note in passing that LC's syntax is hard to parse with traditional deterministic parsers, so in fact LC itself needs a GLL or other general parser). In this section we describe the type system of LC along with a few examples of sugared operations and control flow sufficient to understand the description of the GLL parser below.

Lexical conventions. An LC program is a sequence of *tags* and reserved symbols, notionally separated by white space. A tag is analogous to an identifier in a conventional programming language except that any printable character other than the nine reserved characters `( ) [ ] , | : "` and comments delimited by `(* *)` may appear in a tag. Hence `adrian`, `10` and `-3.5` are all valid tags. Where no ambiguity results, the whitespace between tags may be omitted. Character strings are delimited by `"` and may include most of the ANSI-C escape sequences.

Primitive types. The LC primitive data type generator is the enumeration which maps a sequence of symbols onto the natural numbers and thus into internal machine integers. The enumeration is a `|`-delimited list with a prepended tag which is the type name. A boolean type might be defined as `Bool ( false | true )`. Literals of this type are written `Bool::false` and `Bool::true`. Where no ambiguity results, the tag may be written simply as the tag `true` or `false`. Methods may be defined by enumerating the appropriate finite map from the set of input parameters to the output values, so to define a boolean type that contains a logical AND operation we write:

```
Bool ( false | true
       Bool &(Bool, Bool) := ( (false, false),
                               (false, true)   )  )
```

Where no ambiguity results, methods may be invoked using infix notation.

Every primitive type has an extra value `()` read as *empty*. Newly declared variables are initialised to `()` and variables may be 'emptied' by assigning `()` to them.

We can declare some integer types as

```
Int3 ( -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 )
Unsigned3 ( 0 | 1 | 2 | 3 | 5 | 6 | 7 )
```

We shall assume the existence of a type `Int` which has been defined in this way and which contains sufficient values to allow our computations to proceed without overflow and the usual arithmetic operations.

For each enumeration, the first element maps to 0, the second to 1 and so on. We use $|x|$ to denote the cardinality of set x and $||x||$ to represent the memory space required to represent an element of x. If $|T|$ is the number of explicitly declared enumeration elements then there are $|T|+1$ elements in the type (allowing for the extra value `()`) and so a value of type T occupies at least $||t|| = \lceil \log_2(|T|+1) \rceil$ binary digits in memory.

Enumerations may be composed: `BothBoolAndUnsigned3 (Bool | Unsigned3)` is a shorthand for

```
BothBoolAndUnsigned3 ( false | true | 0 | 1 | 2 | 3 | 5 | 6 | 7 )
```

It is an error to compose enumerations which share tag values.

Tables. Each variable in LC is a table with zero or more dimensions each of which has an index set defined by an LC enumeration. So, `Bool x(Int3)(Bool)` declares a two dimensional array of 16 boolean variables, called `x(-3)(false)`, `x(-3)(true)`, `x(-2)(false)`, ..., `x(3)(true)`. We may write `Bool x(Int3, Bool)` as a shorthand declaration.

Representing types with LC tables. Since LC's type system is so simple, it is reasonable to wonder whether it is sufficient. It clearly is complete in fundamental engineering terms because nearly all modern computers use a single virtual address space of binary memory locations corresponding to a one dimensional LC table. All data structures ultimately are mapped to this representation. Hoare's classic survey of datastructuring techniques listed five mechanisms by which primitive types are combined: Cartesian product, discriminated union, map, powerset and recursive data type. LC provides each of these: a comma-delimited list of type and field names, for example `CartProd (Int3 i, Bool b)`, denotes a Cartesian product and corresponds to a record structure in Pascal or a `struct` in C named `CartProd` with two fields named `i` and `b`; a `|`-delimited list of type and field names, for example `DiscUnion (Int3 i | Bool b)`, denotes a discriminated union and plays a similar role to a C `union`; maps are directly represented by tables of functions; powersets are represented by tables indexed by the type of the powerset whose cells are either empty or contain the element of the unusual enumeration type `Set (isMember)`; and recursive types by (impractically) large tables: for instance the edges of a graph of nodes containing a `CartProd` field is specified as `Set g(CartProd, CartProd)`. The process by which such extensive tables is implemented is described in a later section.

Assignments. In LC, assignment is central. Simple assignment is written `x := 2`. Structurally type compatible assignments may be done is a single statement as in `(Int3 x, Bool y, Unsigned3 z) := (3, true, 3)`. We provide some higher level assignment operations which are used as hints by the datastructure refinement stage:

`x addTo s` is shorthand for `s(x) := isMember`
`x deleteFrom s` is shorthand for `s(x) := ()`
`y selectDelete s` is shorthand for: nondeterministically select an index `i` of an occupied cell in `s` then execute `y := s(i) s(i) := ()`.
`y selectNewestDelete s` is like `selectDelete` except that the most recently assigned cell is guaranteed to be selected (leading to stack-like behaviour)
`y selectOldestDelete s` is like `selectDelete` except that the occupied cell whose contents has been unchanged the longest is guaranteed to be selected, leading to queue-like behaviour.

Control flow. An LC label may appear before expressions or statements, labels are denoted by a tag followed by a colon. Labels may be assigned to variables and

passed as parameters; their type is `CodeLabel` and there is an implicit definition of an enumeration comprising all of the labels in a program in the order in which they are declared. Control may be transferred to a literal label or the contents of a `CodeLabel` variable with the `goto` statement. LC also includes the usual `if` and `while` statements. LC provides some syntactically sugared predicates such as `x in s` which is shorthand for `s(x) != ()`, i.e. that the cell in `s` indexed by `x` is non-empty.

The `for` statement provides a variety of higher level iteration idioms.

> `for x in T` and `for x in Y` each execute once for each member of the enumeration in type T or non-empty member of table `Y` respectively; on each iteration `x` will have a different element but there is no guarantee of the order in which elements are used.
> `for x over T` and `for x over Y` each execute once for each member of the enumeration in type T or non-empty member of table `Y` respectively, with the elements being used in the order in which they are declared in `T` or, respectively, `Y`. In the case of a table, where the index is a tuple, the rightmost element varies most rapidly.

Text strings and output. LC strings are delimited by `"` and accept most ANSI-C escape sequences. An LC program can produce textual output via a method `write()` which, by analogy with ANSI-C's `printf()` function takes a string with embedded place holders %. No type need be supplied, since LC values carry their class with them, but most ANSI-C formatting conventions are supported.

4 An Example GLL Parser

In this section we discuss an LC GLL parser for the grammar

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

In the listing below, data types are declared in lines 1–19 and variables in lines 20–30. The GLL parser body is in lines 42–71 and the support routines (which are grammar independent) are appended in lines 73–111. The dispatcher, which dictates the order in which contexts are computed is at lines 35–40.

There are primitive types `GSSLabel`, `SPPFLabel` and `ContextLabel` whose elements are certain grammar slots together with, in the case of `SPPFLabel`, the grammar terminals and nonterminals, and `#` (epsilon). There are explicit maps `contextLabel`, `codeLabel`, and `sppfLabel` from `GSSLabel` to `contextLabel` and from `contextLabel` to `CodeLabel` and `sppfLabel`. We also define the selector sets for each grammar slot, and *abort* sets which are the complements of the selector sets. The maps `isSlot1NotEnd` and `isSlotEnd` take a `ContexLabel` and return a Boolean. The former returns true if the corresponding slot is of the form $X ::= x \cdot \alpha$ where x is a terminal or nonterminal and $\alpha \neq \epsilon$, and the latter returns true if the corresponding slot is of the form $X ::= \gamma\cdot$. The map `lhsSlot` takes a ContextLabel and returns the left hand side nonterminal of

the corresponding slot. For readability we have left the explicit definitions of all these types out of the listing.

The methods `findSPPFSymbolNode`, `findSPPFPackNode` and `findGGSNode` return a node with the specified input attributes, making one if it does not already exist. Their definitions have also been omitted so as not to pre-empt the data structure implementation discussion presented in the later sections of this paper.

```
1   Set (isMember)
2   N ( A | S )
3   T ( a | b )
4   Lexeme (T | EndOfString)
5   GSSLabel (...)
6   SPPFLabel (...)
7   ContextLabel (...)
8   CodeLabel codeLabel(ContextLabel) := (...)
9   ContextLabel contextLabel(GSSLabel) := (...)
10  SPPFLabel sppfLabel(ContextLabel) := (...)
11  Set abortSet... := (...)
12  Set selectorSet... := (...)
13  GSSNode (GSSLabel s, Int i)
14  GSS (Set GSSEdge (GSSNode src, SPPFSymbolNode w, GSSNode dst))
15  SPPFSymbolNode (SPPFLabel s, Int leftExtent, Int rightExtent)
16  SPPFPackNode (SPPFLabel s, Int pivot)
17  SPPF (SPPFSymbolNode symbolNode, SPPFPackNode packNode,
18        SPPFSymbolNode left, SPPFSymbolNode right)
19  Context (Int i, ContextLabel s, GSSNode u, SPPFSymbolNode w)
20  GSS gss
21  SPPF sppf
22  Context c_C                 (* Current context *)
23  Int c_I                     (* Current input pointer *)
24  GSSNode c_U                 (* Current GSS node *)
25  SPPFSymbolNode c_N          (* Current SPPF node *)
26  SPPFSymbolNode c_R          (* Current right sibling SPPF node*)
27  GSSnode u_0                 (* GSS base node *)
28  Set U(Context)              (* Set of contexts encountered so far *)
29  Set R(Context)              (* Set of contexts awaiting execution *)
30  Set P(GSSNode g SPPFSymbolNode p) (* Set of potentially unfinished pops *)
31  gll_S(
32    SPPF parse(Lexeme I(Int))(
33      goto L_S
34
35      L_Dispatch:
36      c_C selectDelete R
37      if c_C = () return sppf
38      c_I := c_C(i)
39      c_N := c_C(w)
40      goto codeLabel(c_C(s))
41
42      L_A:
43        if I(i) in selectorSet_A_1 addContext(A_1, c_U, c_I, ())
44        goto L_Dispatch
45
46      L_A_1:
47        c_R := (a, c_I, c_I + 1) c_N := getSPPFNodeP(A_1_1, c_N, c_R) c_I := c_I + 1
48        pop
49        goto L_Dispatch
50
51      L_S:
52        if I(i) in selectorSet_S_1 addContext(S_1, c_U, c_I, ())
53        if I(i) in selectorSet_S_2 addContext(S_2, c_U, c_I, ())
54        goto L_Dispatch
55
56      L_S_1:
57        c_U := updateGSS(S_1_1) goto L_A
58      L_S_1_1:
59        if I(c_I) in abortSetS_1_2 goto L_Dispatch
```

```
60          c_U := updateGSS(S_1_2) goto L_S
61        L_S_1_2:
62          if I(c_I) in abortSetS_1_3 goto L_Dispatch
63          c_R := (b, c_I, c_I + 1) c_N := getSPPFNodeP(S_1_3, c_N, c_R) c_I := c_I + 1
64          pop
65          goto L_Dispatch
66
67        L_S_2:
68          c_R := (#, c_I, c_I) c_N := getSPPFNodeP(S_2_1, c_N, c_R)
69          pop
70          goto L_Dispatch
71      )
72
73      Void addContext(ContextLabel s, GSSNode u, Int i, SPPFSymbolNode w)
74        if (s, u, w) not in U(i) (
75          (s, u, w) addto U(i)
76          (s, u, w) addto R(i)
77      )
78
79      Void pop() (
80        if c_U = u_0 return
81        (c_U, c_N) addto P
82        for (w, u) in gss(c_U)
83          addContext(contextLabel(c_U(s)), u, c_I,
84                     getSPPFNodeP(contextLabel(c_U(s)), w, c_N))
85      )
86
87      GSSNode updateGSS(ContextLabel s) (
88        w := c_N
89        v := findGSSNode(s, c_I)
90        if (v, w, c_U) not in gss (
91          (v , w, c_U) addto gss
92          for z in P(v) addContext(s, c_U, z(rightExtent), getSPPFNodeP(s, w, z))
93        )
94        return v
95      )
96
97      SPPFSymbolNode getSPPFNodeP(ContextLabel s, SPPFSymbolNode z, SPPFSymbolNode w) (
98        if isSlot1NotEnd(s) return w
99        SPPFLabel t
100       if isEndSlot(s) t := lhsSlot(s) else t := sppfLabel(s)
101       ( , k, i)  := w
102       if z != SPPFSymbolNode::()  (
103         y := findSPPFSymbolNode(t, z(leftExtent), i)
104         findSPPFPackNode(y, s, k, z, w)
105       )
106       else (
107         y := findSPPFSymbolNode(t, k, i)
108         findSPPFPackNode(y, s, k, (), w)
109       )
110       return y
111     )
112   )
```

5 The Impact of Language Size

Programming languages can vary by factor of seven or more in the cardinality of key enumerations for GLL parsers. Table 1 shows the size of the GSSLabel, SPPFLabel and ContextLabel enumerations for a range of languages. In each case we have used the most authoritative available grammar: language standards for ANSI-C, Pascal and C++; the original report for Oberon and the VS COBOL II grammar recovered from IBM documentation by Ralf Lämmel and Chris Verhoef.

Table 1. GLL measures for programming languages

Grammar	Enumeration extents			Ring	Sets		
	GSS	SPPF	context	length	Defined	Unique	Saving
Oberon 1990	240	645	481	3	442	204	54%
C ANSI X3.159-1989	277	665	505	5	507	176	65%
Pascal: ISO/IEC 7185:1990	393	918	755	3	626	234	63%
Java JLS1.0 1996	384	949	755	4	668	227	66%
C++ ISO/IEC 14882:1998	722	1619	1287	4	1351	294	78%
COBOL (SDF)	1767	4530	3702	5	3502	863	75%

We also show the grammar's *ring length*. This is the longest sequence of terminals present in any alternate in the grammar. It turns out that the dimensions of some tables in a GLL parser may be reduced in size from O(the length of the input string) to O(ring length).

Finally, we show the effect of *set merging*. The number of defined sets is the number of unique sets referenced by a GLL parser: that is the selector sets and the abort sets. It turns out that many of these sets have the same contents, so we can alias names together and only store one table of `isMember` for two or more set names.

6 Process Management in GLL

The description of the GLL technique in Section 2 is essentially declarative. Here we focus on possible implementations of the control-flow in GLL parsing. The heart of GLL parsing from an operational point of view is the task scheduler. GLL contexts (line 11) comprise a `CodeLabel L`, at which to resume execution, and the input pointer, GSS and SPPF nodes that were current at the time the context is created. Each of these specifies an instance of the parser process. Whenever a GLL parser encounters potential multiple control flow paths it creates a process context for each path and then terminates the current process. This happens in two places: (i) whilst processing a nonterminal when the selected productions are added as new processes and (ii) when rules which called a nonterminal are restarted after a pop action (in either the `pop` or `updateGSS` functions).

Now, what is to stop the number of processes growing without limit? The key observation is that our parsing process is *context free*, and this means that all instances of a nonterminal matching a particular substring are equivalent, or to put it another way, if a nonterminal A has already been matched against a particular substring α then we do not have to rerun the parse function. Instead, we merely need to locate the relevant piece of SPPF and connect the SPPF node currently under construction to it. As result, each GLL context created within a run of the parser need execute only once. To ensure this, we maintain a set of contexts that have been seen on a parse `U` along with a set `R` which holds the subset of `U` which currently awaits processing.

Whenever a process terminates, either because it reached the end of a production or because the current input lexeme is invalid, control returns to `L_Dispatch` where a new process is scheduled for execution. In our example, processes are removed non-deterministically (line 36). If we force stack removal by changing the `selectDelete` operator to `selectNewestDelete` then we can simulate the behaviour of a traditional depth-first RD parser except that GLL accepts left recursive grammars. This can be very useful for debugging grammars. It turns out that if we force queue removal using `selectOldestDelete` then we can make significant memory savings, and reduce the maximum size of `R`.

7 Modelling GLL Data Structure Refinement

So far, our LC programs have been based on the notion of a flexible tables which can change their size and even their dimensionality as required. From the theoretician's point of view, we assume that the tables are 'large enough'. Table elements may be accessed in unit time, and on the occasions that we need to iterate over the contents of a table in time proportional to the number of used elements, we assume that a parallel linked list has been constructed in tandem with the table. This model (which we call Big-Fast-Simple or BFS) is sufficient to reason about the asymptotic space and time performance of an algorithm, but is likely to be over-simplistic for real problems, and indeed that is the case for GLL. It turns out that directly implementing GLL data structures such as the SPPF and GSS as arrays is practical for small examples, and is in fact very useful for experimenting with pathological highly ambiguous grammars because we get maximum speed that way, but for realistic inputs to parsers for even small languages such as ANSI-C the direct implementation requires huge memory.

We shall now describe the procedure we use to optimise the space required by our data structures with compromising asymptotic behaviour.

7.1 Address Calculation and Pointer Hopping

We might attempt to implement an LC table as a straightforward multi-dimensional array, or as a sparse array made up of a tree of lists, or as some sort of hybrid. An LC table, like any other kind of data structure, is a container for other data structures and primitive data types; any instance of a type is ultimately just a collection of filled cells along with access paths to each cell, expressed in LC's case as tuples of indexing expressions.

There are essentially only two mechanisms from which to construct access paths: pointer hopping (i.e. linked data structures based on cells which contain the names of other cells) and direct address calculation in which the access path is a computation over the index expressions and some constants. The distinguishing feature, then, is that with address calculation, a data cell's address is a function of its index expression only, but with linking a data cell's address is a function of its index expression and the contents of at least one other cell in the data type.

Multi-dimensional arrays are the most common example of data structures accessed solely by address calculation, but we include hash tables in this category too. Space precludes consideration of hash tables in this paper although we shall introduce our notation for refining LC tables to hash table implementations.

We can characterise the two styles of implementation in terms of their impact on (i) space, (ii) access time for a particular index value, and (iii) iteration time, i.e. the time taken to access all elements.

Consider a one dimensional table x of type Y indexed by type T in which $d \leq |T|$ elements are utilised, the others being set to `()`. So, for example, this LC declaration `Y x(T) := ( (), u1, u1 )` creates a table in which d is 2 since only the second and third elements are in use.

- For a linked implementation, the space required is $O(d)$, the time to access a particular element indexed by $t \in T$ is $O(d)$ and the time taken to iterate over all elements is also $O(d)$.
- For an implementation based on address calculation, the space required is $O(|T|)$, a particular element indexed by $t \in T$ can be accessed in constant time and the time taken to iterate over all elements is $O(|T|)$.

We *refine* an LC table definition by annotating the dimensions to specify either linking (indicated by parentheses) or address calculation indicated by square brackets. We call these possibilities the *dimension modes*.

For a two dimensional table `U x(T,S)` indexed by types $T(1|2|3|4)$ and $S(a|b|c|d)$ we might choose

1. `U x( (T), (S) )` a fully linked representation (using a list of lists),
2. `U x( [T],  [S] )` a two dimensional address calculation,
3. `U x( [T], [S] )` two one-dimensional (vector) address calculations
4. `U x( [T], (S) )` a vector of linked lists, or
5. `U x( (T), [S] )` a linked lists of vectors.

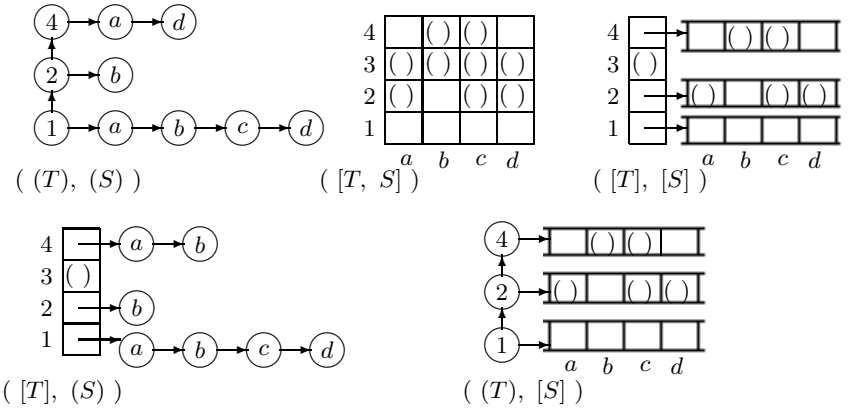

Fig. 1. Table refinement for 2-D structures

These cases are illustrated in Figure 1 for the case of an 2D array indexed by unsigned two-bit integers containing the following ordered pairs: $(1, a)$,$(1, b)$,$(1, c)$, $(1, d)$, $(2, b)$, $(4, a)$ and $(4, d)$.

For multi-dimensional tables implemented as anything other than a full array, the dimension modes of a table are not the only things that affect performance since both the size of a table and its access time for particular elements can be dependent on the *ordering* of the dimensions. We should emphasise that this is purely an implementation matter: the semantics of a table do not change if one permutes the indices in its declaration as long as the indices in accesses to that table are changed to match. This means that we have an opportunity to improve performance through refinement without affecting the analysis of the algorithm as specified in its original, unrefined, form.

Consider the set of ordered pairs $(1, a), (3, a), (4, a)$ and $(4, b)$. The leftmost dimension uses three distinct values, but the rightmost dimension uses only two. If we use the list-of-lists organisation, indexing as `((T), (S))` we can have two structures, one listing the leftmost dimension first and the other the rightmost:

The leftmost-first table requires more space than the rightmost. The driver for this is the *utilisation count* for a dimension of a table. If we can order a table so that, on average, the dimensions with the lowest utilisation counts are used first, then we will on average reduce the size of the table. This effect, in which using the dimension with the highest utilisation count first increases the overall size of the structure can have a very significant effect on memory consumption. Consider the case of a table `x([Unsigned16], [Unsigned16])`. This will comprise vectors of length 2^{16} elements. Let us imagine an extreme case in which only one row of this table is in use, i.e. that we are using cells `x(0,0)`, `x(1,0)`, ... `x(65535, 0)`. If we use the leftmost dimension as the first (column) index, then we need 65 536 row vectors within which only one element is in use. If, on the other hand, we use the rightmost dimension for the column vector then we need only one row vector, all elements of which are in use. We can see that the space allocation for a table indexed as `([T], [T])` can vary between $2|T|$ and $|T|^2$.

Access time can be affected too: if we use a `([Int], (Int))` indexing style then the arrangement with the shortest average row list is fastest. This militates in favour of placing the dimension with the highest utilisation rate first. These two effects are occasionally in tension with each other, but for sparse tables it turns out that the best organisation is to move all of the `[]` dimensions to the left, and then sort the `[]` left-to-right by reducing utilisation counts and then to sort the `()` dimensions left-to-right by reducing utilisation count.

8 The Modelling Process

Having looked at notation for, and effects of, different organisations we shall now look at a real example drawn from the GLL algorithm.

Consider the declaration of the GSS:

```
GSSNode (GSSLabel s, Int i)
SPPFSymbolNode (SPPFLabel s, Int leftExtent, Int rightExtent)
GSS (Set GSSEdge (GSSNode src, SPPFSymbolNode w, GSSNode dst))
GSS gss
```

We begin by flattening the declarations into a signature for table `gss` by substitution:

```
Set gss ((*src*) GSSLabel, Int,
         (*w*)   SPPFLabel, Int, Int,
         (*dst*) GSSLabel, Int
        )
```

This is discouraging: on the face of it the signature for `gss` demands a seven dimensional table, of which four dimensions are indexed by `Int`. We can reduce the extent of those dimensions to `Natural(n)` (a natural number in the range 1 to n) since we know that extents and indices are bounded by the length of the parser's input string. Extents for the other dimensions may be found in Table 1: for ANSI-C there are 277 elements in the `GSSLabel` enumeration and 665 in the `SPPFLabel` enumeration. It is not unreasonable to expect a C compiler to process a string of 10,000 (10^4) tokens: our GSS table would then require at least $10^{16} \times 277^2 \times 665 \approx 5.1 \times 10^{22}$ bits, which is clearly absurd.

Now, the signature for a data structure may contain dependent indices. For instance, in the GSS we can show that the left extent of a GSS edge label `w(leftExtent)` is the same as the index of the destination GSS node, and the rightExtent is the same as the index of the source node. These *repeated dimensions* can be removed, at which point our signature is reduced to

```
Set gss ((*src*) GSSLabel, Int,
         (*w*)   SPPFLabel,
         (*dst*) GSSLabel, Int
        )
```

We must now identify dimensions that are candidates for implementation with address calculation. If we have sufficient runtime profile information, we may choose to ignore asymptotic behaviour and use hash tables tuned to the behaviour of our parser on real examples. In this paper, we restrict ourselves to array-style address mapping only.

For dimensions that require constant time lookup we *must* use address mapping. For other dimensions, we *may* use address mapping if the improvement in performance merits the extra space required.

To find out if an indexing operation must be done in constant time, we must analyse the behaviour of our `gll()` function. The outer loop is governed by the removal of contexts from `R` at line 36. From our analysis in [6], we know that there are quadratically many unique contexts in worst case so the outer loop is $O(n^2)$. We also know that there are $O(n)$ GSS nodes, and thus each node has $O(n)$ out-edges (since the number of edge labels is the number of SPPF labels which is constant).

The parser function itself has no further loops: after each context is extracted from `R` we execute linear code and then jump back to `L_Dispatch`. Only the `pop()` and `updateGSS()` functions have inner loops: in `pop()` we iterate over the $O(n)$ out-edges of a GSS node performing calls to `addContext()` and `getSPPFNode()` and in `updateGSS()` we iterate over the $O(n)$ edges in the unfinished pop set, `P`, again performing calls to `addContext()` and `getSPPFNode()`.

Clearly, `addContext()` and `getSPPFNode()` are executed $O(n^3)$ times, and so must themselves execute in constant time. `addContext()` involves only set tests and insertions which will execute in constant time if we use address calculation. `getSPPFNode()` looks up an SPPF symbol node, and then examines its child pack nodes: these operations must execute in constant time and they turn out to be the most space demanding parts of the implementation.

We are presently concerned only with the GSS implementation so we return to function `updateGSS()`. Lines 81-91 implement the updating of the GSS structure: at line 89 we look for a particular GSS node which will be the source node for an edge, and in lines 90 and 91 we conditionally add an edge to the destination node. This operation is done $O(n^2)$ times, so the whole update can take linear time without undermining the asymptotic performance. We choose to implement the initial lookup (line 89) in constant time and allow linear time for the edge update. By this reasoning, we reach a GSS implementation of

```
Set gss ((*src*) (GSSLabel), [Int],
         (*w*)   (SPPFLabel),
         (*dst*) (GSSLabel), (Int)
        )
```

Finally, we consider ordering of dimensions. We need an estimate of the utilisation counts for each dimension. For long strings, utilisation counts of the `Int` dimensions will be greater than the others, because the extent of the `GSSLabel` and `SPPFLabel` enumerations is constant and quite small (277 and 665 for ANSI-C). It is hard to reason about the utilisation rates for the GSS and SPPF labels: experimentation is required (and some initial results are given below). We note, though, that there are more SPPF labels than GSS labels.

We also need to take account of the two-stage access to the GSS table. We first need to find a particular source node, and then subsequently we use the other indices to check for an edge to the destination node. This means that the indices used in the first-stage query must be grouped together at the left.

On this basis, a good candidate for implementation is

```
Set gss ((*src(i)*) [Int], (*src(s)*) (GSSLabel),
         (*dst(i)*) (Int),
         (*w(s)*)   (SPPFLabel),
         (*dst(s)*) (GSSLabel)
        )
```

This refinement process has given us a compact representation where we have done a lot to save space without destroying the asymptotic behaviour implied by the original unrefined specification. Depending on the results of experimentation, we may wish to relax the constraints a little so as to increase performance at the expense of space. For instance, what would be the impact of changing to this implementation?

```
Set gss ((*src(i)*) [Int], (*src(s)*) [GSSLabel],
         (*dst(i)*) (Int),
         (*w(s)*)   (SPPFLabel),
         (*dst(s)*) (GSSLabel)
        )
```

This proposes an array of arrays to access the source GSS node label rather than an array of lists. We can only investigate these kinds of engineering tradeoff in the context of particular grammars, and particular sets of input strings. By profiling the behaviour of our parser on typical applications we can extract real utilisation counts.

We ran a GLL parser for ANSI-C on the source code for `bool`, a Quine-McCluskey minimiser for Boolean equations, and measured the number of GSS labels used at each input index. The first 50 indices yielded these utilisation factors:

11, 18, 13, 10, 0, 1, 61, 11, 47, 0, 0, 19, 23, 0, 10, 3, 18, 0, 10, 18, 0, 44, 0, 0,
4, 7, 0, 45, 0, 10, 0, 44, 0, 0, 10, 0, 44, 0, 0, 46, 0, 14, 2, 0, 6, 16, 3, 45, 0, 10

so, there are 11 GSS nodes with labels of the form $(0, l_g \in \texttt{GSSLabel})$, 18 of the form $(1, l_g)$ and so on. The total number of GSS nodes here is 623. The mean number of GSS labels used per index position is 12.46, but 38% of the indices have no GSS nodes at all. This might seem initially surprising, but recall that a GSS node is only created when an input position has an associated grammar slot which is immediately before a nonterminal.

Now, linked list table nodes require three integer words of memory (one for the index, and two for the pointers). If these statistics are typical, for every 100 index positions we would expect 38 to be unpopulated, which means that there would be 62 second level vectors of length 277 (the `GSSLabel` extent for ANSI-C), to a total of 17 174 words. In the linked version, we would expect 1246 nodes altogether, and that would require 3738 words, so the `([], [], ...)` representation requires 4.6 times as much memory as the `( (), (), ...)` version, which may not be too onerous. The performance advantages are clear: the mean list length would be 12.46, leading to an expected lookup time 6–10 times

slower than directly indexing the vectors even without taking into account cache effects. There are also several lists which would be greater than 40 elements long, leading to substantial loss of performance.

9 Conclusion: Prospects for Automatic Refinement

In this paper we have given some details of our implementation of the GLL technique using an approach that separates high level reasoning about algorithm complexity from details of implementation, and a modelling process that allows us to produce compact implementations which achieve the theoretically predicted performance. We summarise our procedure as follows.

1. Flatten declarations to signatures by substitution.
2. Establish upper bounds on dimensions.
3. Remove dimensions that may be mapped from other dimensions and create maps.
4. Remove repeated dimensions.
5. Identify dimensions that govern the time asymptote — these are the *critical* dimensions.
6. For each dimension, compute, measure or guess the average number of elements of the index type T that are used in typical applications: this is the dimension's Likely Utilisation Count (LUC). The ratio ($|T|$/LUC) is called the dimension's load factor (LF).
7. Implement critical dimensions as one dimensional tables.
8. Implement dimensions whose LF is greater than 33% as one dimensional tables.
9. Group dimensions by query level, with outer queries to the left of inner queries.
10. Within query groups, arrange dimensions so that `[]` indices are always to the left of `()` indices.
11. Within query groups and index modes, sort dimensions from left to right so that load factors increase from left to right.

For a given data structure, the available index modes and orderings define a space of potential implementations. As we have shown, the basic procedure minimises space, but nearby points in the implementation space may have better performance, and as long as the application fits into available memory, most users would like to have the faster version.

In the future we propose that semi-automatic systems be constructed to automatically explore these spaces in much the same way that hardware-software co-design systems have successfully attacked the exploration of implementation spaces for hardware oriented specifications. We have in mind the annotation of critical dimensions by the theoretician, allied to a profiling system which will collect statistics from sets of test inputs so as to measure or estimate load factors. We believe that a notation similar to LC's will be suitable for such an optimiser.

References

1. Gnu Bison home page (2003), `http://www.gnu.org/software/bison`
2. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM 13(2), 94–102 (1970)
3. Knuth, D.E.: On the translation of languages from left to right. Information and Control 8(6), 607–639 (1965)
4. Nozohoor-Farshi, R.: GLR parsing for ϵ-grammars. In: Tomita, M. (ed.) Generalized LR Parsing, pp. 60–75. Kluwer Academic Publishers, The Netherlands (1991)
5. Scott, E., Johnstone, A.: Generalised bottom up parsers with reduced stack activity. The Computer Journal 48(5), 565–587 (2005)
6. Scott, E., Johnstone, A.: GLL parsing. Electronic Notes in Theoretical Computer Science (2009)
7. Scott, E., Johnstone, A., Economopoulos, G.: A cubic Tomita style GLR parsing algorithm. Acta Informatica 44, 427–461 (2007)
8. Tomita, M.: Efficient parsing for natural language. Kluwer Academic Publishers, Boston (1986)
9. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: the ASF+SDF compiler. ACM Transactions on Programming Languages and Systems 24(4), 334–368 (2002)
10. Younger, D.H.: Recognition of context-free languages in time n^3. Inform. Control 10(2), 189–208 (1967)