Available online at www.sciencedirect.com
SCIENCE DIRECT®

ELSEVIER Electronic Notes in Theoretical Computer Science 110 (2004) 97–113

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

The Grammar Tool Box: A Case Study Comparing GLR Parsing Algorithms

Adrian Johnstone[1] Elizabeth Scott[2] Giorgios Economopoulos

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

Abstract

The Grammar Tool Box is a toolset for manipulating Context Free Grammars and objects associated with them such as parsers, languages and derivations. GTB has three main rôles: as a pedagogic tool; as an experimental platform for novel algorithms and representations; and as a production tool for translator front end generation. In this paper we give an overview of GTB and its companion Java-based animator tool PAT. We illustrate the use of the toolset in the construction of a comparative study of three variants of the Tomita-style GLR parsing algorithm running on LR(0), SLR(1) and LR(1) tables for ANSI-C, ISO-Pascal and IBM VS-COBOL, and give results showing the size of the structures constructed by these parsers and the amount of searching required during the parse, which abstracts their runtime.

Keywords: GLR parsing, grammar types, context free languages

The computing literature contains plenty of experimental studies, but computer science, unlike traditional experimental science, is poor at providing genuinely repeatable results. This arises partially from the rapid changes in the underlying technology and partly from a natural desire of researchers to explore new fields rather than comprehensively mapping the territory opened up by pioneers. Furthermore the complexity of computer systems can make it difficult to define problems in a manner that allows comparative experimental approaches.

Compilers and formal language translators in general are perhaps our best candidate for combining formal rigour with engineering practicality. It is

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

doi:10.1016/j.entcs.2004.06.008

forty years since the links were noted between Chomsky's formalisms and the engineering practice of language design and translator implementation; and nearly twenty years since the engineering practice for parsing conventional programming languages settled on the use of LR style bottom up parsers, usually based on LALR(1) tables which are automatically generated, and the simpler LL(1) parsers which are often hand written.

LALR techniques admit a large subset of the deterministic languages [5] and, with a little judicious use of prioritisation to disambiguate table conflicts, the well known generators such as YACC [9] and its derivatives GNU Bison and BTYACC [6] have reduced parser generation to a mostly clerical task. Nevertheless, users of these tools have to grapple with the lack of generality of deterministic parsers, and it is our experience that both neophyte and experienced language designers experience nasty surprises. Bottom up parsers, in particular, display behaviour that can be hard to interpret in terms of the grammar. For example, YACC-like tools allow semantics to be specified within a production even though semantics execution is in reality associated with the reduction of an entire rule. YACC will silently split a rule to support semantics execution, but in doing so may generate rules with LALR(1) conflicts. The effect from a users' point of view is that a working parser breaks when semantics are added.

In the last decade the computing community has shown an increasing interest in parsing techniques that go beyond the standard approaches. There are a plethora of parser generators that extend both top-down and bottom-up approaches with backtracking and lookahead constructs. As we have noted elsewhere [11] such parsers can display surprising pathologies: in particular parser generators such as PRECC [3], PCCTS [19], ANTLR and JAVACC are really matching strings against *ordered* grammars in which the rule ordering is significant, and it can be hard to specify exactly what language is matched by such a parser. In any case, backtracking yields exponential parse times in worst case.

A safer approach is to use one of the truly general context free parsing algorithms such as Earley [7], CYK [29] or a variant of Tomita's GLR algorithms of which more in section 3 below. Many of these algorithms are primarily *recognisers* which return data structures from which derivations may be extracted one at a time. For ambiguous grammars this may be unacceptable since the number of derivations is not necessarily finite, and no guarantees can be made about the order in which they are extracted, hence the practical time bound may be dominated by the examination (and rejection) of semantically unacceptable derivations.

Practical general algorithms display at least cubic worst case time behaviour but on modern hardware this need not preclude their use. Several Eiffel compilers use Earley parsers to support their relaxed use of expression separators. Tools such as ASF+SDF [27] stress generality and can parse ambiguous languages with support for *shared packed parse forests* which efficiently encode all possible derivations along with sophisticated techniques for disambiguating the forest. ASF+SDF uses GLR parsing to accomplish this: as a final indicator that general techniques are entering the mainstream consider that even GNU Bison had recently acquired a GLR mode, although the implementation is perhaps not yet industrial strength.

In recent years we have studied a wide variety of general parsing techniques along with mechanisms for extracting derivation forests; semi-automatically generating abstract syntax trees by pruning such derivations; and dataflow analysis and code generation techniques based on the resulting structures. We have applied these techniques to conventional compiler-oriented translators and to reverse compilation and reverse synthesis of hardware description languages [13]. We have also used them for searching databases of biological sequence data. We ended up with a set of only loosely comparable tools: in particular it was hard to answer what-if questions about parse-time performance and the size of the structures required for particular techniques in particular application niches.

GTB is a unifying framework into which we can implement new theoretical contributions beside those already in the literature so as to allow direct experimental comparisons as well as to act as a production quality translator generator. Presently GTB is very much work in progress, with most of its existing capability focused on generalised parsing. In this paper we give an overview of GTB and illustrate the use of the tool in the construction of a comparative study of three variants of the GLR parsing algorithm running on LR(0), SLR(1) and LR(1) tables for ANSI-C, ISO-Pascal and IBM VS-COBOL, and give results showing the size of the structures constructed by these parsers and the amount of searching required during the parse, which abstracts their runtime.

1 The GTB language and capabilities

GTB is an interpreter for a procedural programming language with facilities for direct manipulation of translator related data structures. A set of built-in functions is provided for creating, modifying and displaying these structures. At the simplest level, the GTB language is used for scripting a standard process such as the generation of an SLR(1) parse table. Unlike a conventional monolithic parser generator, GTB requires the generation process to be specified as a detailed chain of operations on grammars and automata. Our goal is to open up the degrees of freedom in translator implementation in a structured way so that reproducible experiments can be mounted between competing techniques. Naturally, a pre-written GTB script may be used to get the effect of a conventional parser generator like YACC or RDP so GTB can replace those kinds of tools in a production environment.

The following small GTB example illustrates the definition of a (tiny) language. The grammar rules are written in simple BNF, one rule for each non-terminal, terminals are single quoted strings and # represents ϵ. The example shows the generation of all sentential forms in the language and the construction of the LR(1) DFA and parse table *via* the construction of an NFA followed by the application of the subset construction (as described in [8]). The NFA is also output in a format which can be viewed using the VCG tool [21].

```
S ::= 'a' | A B | A 'z'.
A ::= 'a' .
B ::= 'b' | # .
(this_gram := grammar[S]
 generate[this_gram 0 left sentential_forms]
 this_nfa := nfa [this_gram lr 1 nullable_reductions]
 write[open["nfa.vcg"] this_nfa]
 export[open["rnlr1.tbl"] parse_table[dfa[this_nfa]]]
 write[this_gram "\n" CPU_time " CPU seconds elapsed\n\n"]
)
```

The GTB language can also be used to specify semantics to be executed at parse time or indeed during later stages of translator implementation. We do not wish GTB generated translators to necessarily carry a copy of the GTB interpreter with them: the intention is that GTB's language should be sufficiently unsophisticated that GTB expressions can be cleanly translated into a target implementation language such as C++, Java or ML. Semantics may also be specified as fragments of the target language as is usual for compiler-compilers but of course portability is then lost. Our translator generator model is otherwise conventional: we are generating procedural programs with all of their associated flexibility and danger: we are not using term-rewriting as does ASF+SDF nor tree transformers as does Metafront [2] although we do make extensive use of *ad hoc* tree-to-tree transformations within GTB functions. We aim to make GTB accessible to procedural programmers who are not necessarily as comfortable with formal approaches to software construction as we might wish.

As an aid to algorithm analysis and grammar debugging we provide graphical representations of some structures. Static structures are displayed using VCG. Dynamic animations of structures during construction are provided by our Java-based tool PAT (Parser Animation Tool) which we describe in section 2.

The GTB language uses an object oriented style to provide modular specifications with user defined data types and their associated methods packaged into statically type-safe objects analogous to Java and C++ classes. Type inference is used to relieve the user of the need to declare variables, and a loose punctuation style allows specifications without statement and parameter separators although as with Eiffel, this results in a non-deterministic syntax. As in the Ella hardware description language, functions with one or two operands may be written as infix operators. At the lowest level, the semantics of primitive objects and their compositions are defined in terms of symbol enumerations and a primitive *selection* operator. This approach arises from our interest in hardware description languages in which computations might ultimately need to be resolved to the level of simple Boolean operations applied to sets of wires. We make no claims for the appropriateness of this model or indeed the usability of GTB's (rather concise) syntax, but it provides a compact and powerful notation with well-defined translations to standard programming languages.

GTB objects may also include grammar rules written using the Wirth-like syntax we implemented in our RDP LL(1) parser generator. A rule may have embedded GTB expressions describing semantics. Rules which are to be used with parser generator functions may have statements in the target language added between special brackets. There are a set of *promotion operators* for describing the pruning of a derivation into a *Reduced Derivation Tree* (RDT): these are effectively short-hands for semantic expressions that manipulate the derivation tree. The general form is inherited from RDP: the operators work in such a way that they may be implemented on-the-fly in top-down parsers so that single pass translators which directly yield an RDT may be constructed.

We follow Grune's treatment [8] of bottom up parser generation, both in our theoretical results and in the implementation. Traditional treatments of LR parsers focus on the definition of the LR handle-finding automaton as a closure over sets of items. In general an item is an ordered pair (S, F) where S is a 'dotted' position in the grammar (which in GTB we call a *slot*) and F is a lookahead set representing those terminals which a parser may legally encounter during a parse. LR(0), SLR(1) and LR(1) variants are obtained for different kinds of lookahead: setting each F to be the complete set of terminals in the grammar gives an LR(0) automaton; setting each F to be a local follow set computed separately for each *instance* of a nonterminal gives an LR(1) automaton; and setting F to be the appropriate global follow set gives the SLR(1) automaton. In [8] these relationships are emphasised by writing a grammar as an NFA in a way which captures these relationships directly. The handle-finding DFA is then derived from this NFA by applying the subset construction. We have found this approach useful both pedagogically and as a basis for theoretical analysis of LR parsers. As an implementation method it is less attractive since the LR(1) NFA's for some languages can be large and in our present scheme must be co-resident with the DFA. Some ingenuity has been required to design internal representations for the NFA that optimise the use of memory, and we do not rule out more direct construction techniques for production use.

GTB can handle multiple grammars simultaneously, and extract multiple grammars from a rule set by using different start symbols (a facility which is useful for some techniques that segment grammars, such as the Aycock and Horspool trie-based automaton [1]). The built-in function `grammar` constructs a grammar object from a particular start rule and as a side effect calculates first and follow sets. Grammar augmentation is automatically applied, if necessary. The following is an extract of the text generated by running GTB with the example script given above.

```
Grammar alphabet
   0 !Illegal     6 A
   1 #            7 B
   2 $            8 S
   3 'a'          9 S!augmented
   4 'b'
   5 'z'
nullable_reductions = {S ::= 'a' . , S ::= A . B , S ::= A B . ,
      S ::= A 'z' . , A ::= 'a' . , B ::= 'b' . , B ::= . #,
      S!augmented ::= S . }
first(S) = {'a', A}
follow(S) = {$}
rhs_follow(S, 0) = {#, 'b', B}
rhs_follow(S, 1) = {#}
rhs_follow(S, 2) = {'z'}
...
0.010000 CPU seconds elapsed
```

The set `rhs_follow(S, 0)` is the instance follow set for the first nonterminal on the right hand side of S.

The `generate` function uses breadth first search to output either sentential forms or sentences of the grammar. We can specify leftmost, rightmost or random selection of the nonterminal in a sentential form for expansion. Of course, most interesting languages are infinite, so we can specify an upper bound on the number of outputs. The parameter of zero in the above example specifies no upper bound, so all sentential forms will be generated.

Nondeterministic automata are constructed using the `nfa` function. The types presently supported are LL, LR and the 'unrolled'-LR automata used in our version of Aycock and Horspools parsing algorithm [12]. Follow sets are

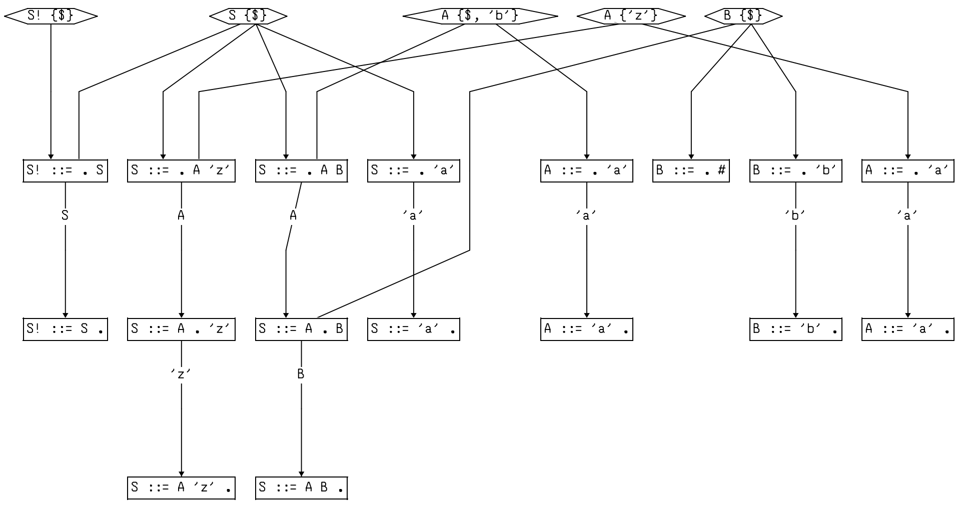

Fig. 1. Nondeterministic LR(1) automaton

specified with positive integers for right-hand-side (instance-level) follow sets and negative integers for left-hand-side follows, so an unrolled SLR(1) table is obtained with `nfa[this_gram unrolled -1]`. (In practice, we allow some sugared calls so `nfa[this_gram slr 1]` has the expected effect.) Figure 1 shows a VCG rendering of an LR(1) automaton for the example grammar. Some other specialised automata, such as Aycock and Horspool's trie-based PFA can be constructed using function specific calls.

A Knuth-style parse table is obtained by running the subset construction on one of these NFA's. Figure 2 shows the LR(1) DFA generated from our example. Tables may be written in a form convenient for human consumption or exported to other tools.

2 Parsers and parser animation

As well as outputting generated parsers, GTB has table driven parser functions built in so that a batch of experiments can be conducted in a single run. A separate Parser Animation tool (PAT) has its own implementations of our parser algorithms which run from tables generated by GTB. PAT is written in Java so that animations may be run as a Web applet. GTB is written in C++ for efficiency.

In operation, the PAT user must read in a table generated by GTB and then select a parsing algorithm and a candidate string. A full parse is performed so that all of the intermediate data structures (such as Graph Struc-

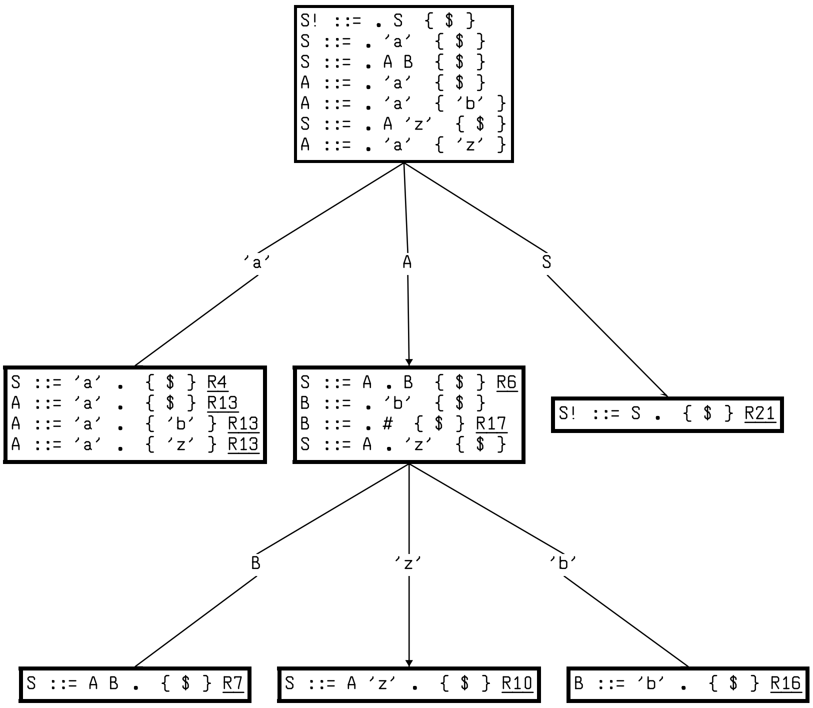

Fig. 2. Deterministic LR(1) automaton

tured Stacks or Shared Packed Parse Forests) required by the algorithm are constructed in their entirety. The algorithm is then animated by interactively displaying the history of the run allowing the user to step forwards and backwards in time and watch the structures build up.

The rest of this paper concerns some experiments comparing the performance of three GLR parsing variants working with LR(0), SLR(1) and LR(1) tables on grammars for Pascal, C and Cobol. We begin by reviewing the operation of GLR parsers and the development of the core ideas. We then discuss the provenance of our grammars and give some results.

3 GLR parsing

GLR parsing was introduced by Tomita [26] as a means for natural language parsers to capture all possible derivations when parsing potentially highly ambiguous grammars. If a grammar contains cycles the number of derivations of a string may be infinite, but even for cycle-free grammars there may be exponentially many derivations. Tomita uses a graph structured stack (GSS) to efficiently compute all of these derivations.

A naïve approach to generalising Knuth's LR parser would be to simply clone the parser whenever a conflict is encountered in the table, creating a complete copy of the parser. An LR parser's configuration is completely encoded by the contents of its state stack, so rather than constructing a copy of the stack as these clone processes are created we could use a graph representation and share the common prefix, so for instance the three stacks on the left below could be represented by the middle graph.

$$\begin{array}{ccc} 5 & 5 & 9 \\ 3 & 4 & 4 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{array}$$

$$\begin{tikzcd} 2 & 1 \arrow[l] & 3 \arrow[l] & 5 \arrow[l] \\ & & 4 \arrow[ul] & 5 \arrow[l] \\ & & & 9 \arrow[ul] \end{tikzcd}$$

$$\begin{tikzcd} 2 & 1 \arrow[l] & 3 \arrow[l] & 5 \arrow[l] \arrow[dl] \\ & & 4 \arrow[ul] & \\ & & & 9 \arrow[ul] \end{tikzcd}$$

This scheme saves space by sharing stack prefixes. However Tomita noted that the context-free nature of rules by which the GSS is built mean that at most one stack top needs to be maintained for each state in the table, so we can also share stack postfixes as represented by the righthand GSS above.

There is an important cost associated with the use of a GSS: in an LR parser a reduction is performed by popping j states from the stack where j is the length of the rule being reduced. This can be performed with a single modification of a stack pointer. In the GLR case a reduction involves a *search* down all pathways of length j, and in worst case this can require $O(2^j)$ time. Space precludes a full exposition of the algorithm here: see [24] for a more detailed discussion.

Tomita initially presented his algorithm in five stages [26]. We are interested in the second stage, which we call Tomita-1, which can be applied to all ϵ-free grammars. Tomita extended this algorithm to include grammars with ϵ-rules but this extension is complex, and turned out to fail to terminate on grammars with hidden left recursion.

If we extend Tomita-1 to include ϵ-rules in the obvious way, we call such an algorithm Tomita-1e, we find that it works incorrectly on grammars with hidden right recursion.

Farshi, [18], extended Tomita-1 to include ϵ-rules and added a brute force search to ensure that rules with hidden right recursion are correctly processed. Farshi also introduced loops into the GSS which allowed the algorithm to cope with grammars containing cycles. Farshi described only a recogniser version of his algorithm. Rekers [20] turned Farshi's algorithm into a parser in which the GSS edges are labelled with SPPF nodes. Visser [28] uses a generally similar approach but incorporates lexical processing directly into the parser and its associated GSS and SPPF.

A separate line of development is represented by our right-nulled (RN) parsers, in which we redefine a reducible item as any item with a nullable suffix after the dot position. Using these right-nulled tables Tomita-1e can directly handle all context free grammars [10,24] without the brute force search required by Farshi's approach. We refer to this as Right-Nulled GLR parsing (RNGLR). We have also described a resolution process that may be applied to remove some conflicts in RN tables [23] and *binary* BRNGLR variants that run in $O(n^3)$ time as opposed to $O(n^j)$ where j is the length of the longest production [22]. Below, we use GTB to compare the performance of the Farshi, RNGLR and BRNGLR algorithms.

4 Grammars for ISO-Pascal, ANSI-C and COBOL

Pascal and C typify the top down and bottom up schools of language design. In the folklore at least, Pascal is thought of as being designed for LL(1) parsing and C for LALR parsing. In practice, Pascal is indeed reasonably close to LL(1) notwithstanding the IF-THEN-ELSE ambiguity and the need for lexical backtracking to distinguish between a real literal (`2.3`) and an integer range `2..3`. C is essentially parsable by LALR parsers, but was not initially designed that way. The LALR(1) ANSI-C grammar was only written by Tom Penello in about 1983. Bjorn Stroustrup described at some length the difficulties involved in attempting to build parsers for early versions of C++ using a hybrid of YACC and a lexer containing 'much lexical trickery relying on recursive descent techniques' [25, pp 68–69,103]. Those of us interested in generalised parsing should perhaps be grateful for C++'s nondeterministic syntax since it has clearly stimulated the development of tools such as ANTLR and the GLR mode of Bison. COBOL's development was contemporary with that of Algol-60 and thus pre-dates the development of deterministic parsing techniques. The language has a large vocabulary (some 400 terminals depending on variant) which can challenge any table based parsing method.

For these experiments we have used the grammar for ISO-7185 Pascal extracted from the standard, the grammar for ANSI-C extracted from [14] and a grammar for IBM VS-Cobol developed in Amsterdam. The original extraction of this grammar is described at length in [16] and some interesting work on techniques to generate tolerant variants is described in [15]. A version of the grammar is available as a hyperlinked browsable HTML file at `http://www.cs.vu.nl/grammarware/browsable/vs-cobol-ii/`: we used a version prepared for ASF+SDF from which we extracted the context free rules.

In all three cases we have suppressed lexical level productions and used separate tokenisers to convert source programs into strings of terminals from the main grammar, so for instance the Pascal fragment

```
program tree_file(input, output) ;

const MAX_INTSETS = 200 ; MAX_TREES = 200 ;

function intset_create(var s : intset) : intset_sig ;
```

is tokenised to

```
program ID ( ID , ID ) ;
const ID = INTEGER ; ID = INTEGER ;
function ID ( var ID : type_ID ) : type_ID ;
```

For Pascal, our source was a tree viewer program which has approximately 5,000 tokens; for ANSI-C a Boolean equation minimiser of about 4500 tokens; and for COBOL, strings of approximately 2,000 tokens extracted from the test set supplied with the grammar.

4.1 EBNF to BNF conversion issues

Our source COBOL and Pascal grammars are specified using variants of EBNF, that is, BNF supplemented by regular expressions. The browsable COBOL grammar also includes *permutation phrases* [4] which are used to express free-order constructs in which each phrase may appear at most once: in the ASF+SDF version these are mapped to Kleene-closure over the permutation phrases with the assumption that semantic checks will weed out strings with multiple occurrences of a phrase.

The standard LR table construction algorithms do not support regular expressions within rules, so EBNF rules must be converted to BNF. In general there are many ways to do this and not all approaches yield the same size of table or the same amount of parse time stack activity. As a trivial example consider the expansion of Kleene closure to BNF. Left recursive rules are handled efficiently by LR parsers but right recursive rules generate extra stack activity. On the other hand, left recursive rules cause nontermination in LL parsers, so in that case right recursion is mandatory.

One sometimes sees discussions on the size of parse tables for languages (see for instance http://compilers.iecc.com/comparch/article/00-02-072). To make realistic comparisons we need to know both the full provenance of the grammar, and the way that it has been massaged into a form acceptable to the tools from which measurements have been taken.

The IBM VS-COBOL grammar is salutary. Our first attempt expanded all closures by creating a new left-recursive nonterminal for each closure. GTB ran out of memory when trying to create an LR(1) NFA for this grammar. Simple back substitution for head and tail recursive rules generated an NFA with 1.4 million nonterminal headers. Applying a broader set of substitutions yields an NFA with only 21,000 headers.

Clearly, a mechanism for applying transformations in a way that can be audited by grammar users is needed: starting with a standard grammar is helpful but not if we then perform a series of *ad hoc* operations as we remove the EBNF syntactic sugar which can cause the size of our automata to vary by nearly two orders of magnitude.

We use a separate tool `ebnf2bnf` to perform these conversions which takes as input an (E)BNF grammar annotated with expansion operators and outputs an (E)BNF grammar. The tool constructs a rules tree from the original grammar and then performs tree transformations under the control of the annotations, before writing the grammar back out. An ambitious environment that supports this kind of operation has been prototyped in ASF+SDF [17]: the authors describe EBNF to BNF conversion as Yacc-ification. Our tool emphasises ease-of-use and traceability of the basic operations.

We provide five operations: `!^` (substitute); `!|` (expand head or tail closure with simple recursion); `!<` (expand closure using left recursion); `!>` (expand closure using right recursion); and `!*` (multiply out brackets). So, for instance a rule of the form

```
S::= A ( B | C )!* D
```

will be expanded to

```
S ::= A B D | A C D
```

and rules of the form

```
S ::= A X!^ D        X ::= B | C
```

will be expanded to

```
S ::= A ( B | C ) D
```

5 Experiments

We consider first the size of the parse tables required for our three grammars. For a given combination of grammar and LR(0), SLR(1) or LR(1) NFA the deterministic handle finding automata, and thus the tables, will be the same size for both RN- and conventional Knuth-style reductions. In general the RN tables will contain far more conflicts, which might be expected to generate more searching during GSS construction. It turns out that the RNGLR algorithm short-circuits these extra searches, as we shall see. Table 1 gives the number of rows (states) and the number of table cells with conflicts (multiple entries) for Pascal, ANSI-C and COBOL. The number of columns in a table

Pascal	LR(0)	SLR(1)	LR(1)	RNLR(0)	RNSLR(1)	RNLR(1)
states	434	434	2608	434	434	2608
conflicts	768	1	2	4,097	242	1,104
C	LR(0)	SLR(1)	LR(1)	RNLR(0)	RNSLR(1)	RNLR(1)
states	383	383	1797	383	383	1797
conflicts	391	88	421	391	88	421
COBOL	LR(0)	SLR(1)	LR(1)	RNLR(0)	RNSLR(1)	RNLR(1)
states	2692	2692	–	2692	2692	–
conflicts	131,506	65,913	–	167,973	73,003	–

Our Pascal grammar has 286 symbols; C has 158 symbols and COBOL has 1028 symbols.

Table 1
Parse tables sizes and conflict counts

is the number of symbols in the grammar.

COBOL has a large alphabet and requires more than seven times as many states as ANSI-C even for LR(0) and SLR(1) tables. In fact, our LR(1) table generator ran out of memory when processing COBOL, so we leave those entries empty. We can also see that this COBOL grammar is highly nondeterministic, reflecting the construction process described in [16].

We now consider the size of the GSS and SPPF structures. The Farshi and RNGLR algorithms generate the same structures. The binary BRNGLR algorithm achieves cubic run times but at the cost of a worst-case constant factor increase in the size of the structures, that is the asymptotic space requirements of RNGLR and BRNGLR algorithms are the same, but for any grammar with productions greater than two symbols long the BRNGLR algorithm introduces additional nodes into both the GSS and SPPF. Table refgss:sppf:size shows expansions of between 5 and 20% in the size of the structures. As we apply stronger parsing techniques, there is a potential trade off between the amount of nondeterminism in the table and the size of the GSS. Some early reports from the natural language processing community suggest that LR(1) based GSS's would be much larger than SLR(1) ones because the number of states is so much larger, and in the limit the number of nodes in a GSS is bounded by the product of the number of table states and the length of the string. However, not all states will be populated, and our figures show that in practice LR(1) GSS's are a little smaller than their SLR(1) equivalents. Of course, the LR(1) tables themselves are usually much bigger than SLR(1) tables (by a factor of 4.6 for ANSI-C) so the rather small reduction in GSS size might only be justified for very long strings. ASF+SDF uses SLR(1) tables.

	RNGLR/Farshi GSS edges	BRNGLR GSS edges	RNGLR/Farshi SPPF edges	BRNGLR SPPF edges
Pascal LR(0)	31,015	34,441	26,743	30,387
Pascal SLR(1)	21,258	23,826	21,045	21,938
Pascal LR(1)	21,135	23,655	18,147	20,740
C LR(0)	39,389	41,748	40,454	42,800
C SLR(1)	28,604	30,670	29,033	31,092
C LR(1)	28,477	30,512	28,761	30,797
COBOL LR(0)	23,002	26,461	24,792	29,342
COBOL SLR(1)	13,512	14,517	12,204	13,167

Table 2
The size of GSS and SPPF structures

Now we turn to performance. The asymptotic time order of these algorithms is dominated by the search time in the GSS associated with reductions. We count the number of times each edge is visited during a search, and present here the sum over all these individual edge counts, which abstracts the total amount of searching performed in each run. We present figures for four algorithms: BRNGLR, RNGLR and two variants of Farshi's algorithm. Farshi's solution to the problem with Tomita's algorithm requires that each time a reduction edge r is added to an already processed GSS node, all paths from state nodes at the same level are searched to see if r can be reached, in which case reductions must be re-queued. A naïve reading of Farshi's paper would suggest that all paths from all active state nodes must be searched in their entirety, but an obvious optimisation is to abort any path search once it reaches a node on a lower level, since edge r must be reached *via* the most recent level. We refer to these two variants as Farshi-naïve and Farshi-opt. Table 3 shows clear performance advantages arising from the RNGLR and BRNGLR algorithms, especially in the case of COBOL where there is an order of magnitude difference in the search costs.

6 Conclusions and acknowledgements

GTB has allowed us to make direct comparisions between three different GLR-style algorithms and three types of LR table. In all cases the LR(1) table resulted in smaller and faster run-time parsers, but the improvement over SLR(1) is not very great, while the increase in the size of the table is significant. The RNGLR algorithm performs better than the Farshi algorithm, but comparably with the BRNGLR algorithm. This latter result is not surprising

GSS edge visits	Farshi-naïve	Farshi-opt	RNGLR	BRNGLR
Pasacal LR(0)	41,460	38,964	8,556	8,550
Pascal SLR(1)	25,753	24,100	5,665	5,663
Pascal LR(1)	23,305	22,459	5,572	5,570
C LR(0)	42,707	42,251	5,184	5,180
C SLR(1)	30,235	29,940	4,502	4,498
C LR(1)	30,754	30,484	4,450	4,446
COBOL LR(0)	139,187	103,120	10,056	9,554
COBOL SLR(1)	47,464	38,984	3,581	3,487

Table 3
Run time performance

as the test grammars are for real languages and do not trigger supra-cubic behaviour in the RNGLR algorithm.

We are very grateful to Steven Klusener and Ralf Laemmel for allowing their IBM VS-COBOL grammar to be used here; to Mark van den Brand for helpful discussions on GLR parsing and application to COBOL reengineering; to Georg Sander for his VCG graph visualisation software and for allowing it to be distributed with our toolkits; and to the anonymous referees for their helpful comments.

References

[1] John Aycock and Nigel Horspool. Faster generalised LR parsing. In *Compiler Construction: 8th International Conference, CC'99*, volume 1575 of *Lecture Notes in computer science*, pages 32 – 46. Springer-Verlag, 1999.

[2] Claus Braband, Michael I. Schwartzbach, and Mads Vanggaard. The `metafront` system: Extensible parsing and transformation. In *LDTA-03*, Electronic Notes in Theoretical Computer Science. Elsevier, 2003.

[3] Peter T. Breuer and Jonathan P. Bowen. A PREttier Compiler-Compiler: Generating higher-order parsers in C. *Software Practice and Experience*, 25(11):1263–1297, November 1995.

[4] Robert D. Cameron. Extending context-free grammars with permutation phrases. *ACM Letters on Programming Languages and Systems*, 2(1–4):85–94, March–December 1993.

[5] Franklin L DeRemer. *Practical translators for LR(k) languages.* PhD thesis, Massachussetts Institute of Technology, 1969.

[6] Chris Dodd and Vadim Maslov. `http://www.siber.com/btyacc`. June 2002.

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

[8] Dick Grune and Ceriel Jacobs. *Parsing Techniques: A Practial Guide.* Ellis Horwood, Chichester, England, 1990.

[9] S. C. Johnson. Yacc — yet another compiler-compiler. Technical Report 32, AT&T Bell Laboratories, 1975.

[10] A. Johnstone and E. Scott. Generalised reduction modified LR parsing for domain specific language prototyping. In *Proc. 35th Annual Hawaii International Conference On System Sciences (HICSS02)*, IEEE Computer Society. IEEE, New Jersey, 2002.

[11] Adrian Johnstone and Elizabeth Scott. Generalised recursive descent parsing and follow determinism. In Kai Koskimies, editor, *Proc. 7th Intnl. Conf. Compiler Construction (CC'98), Lecture notes in Computer Science 1383*, pages 16–30, Berlin, 1998. Springer.

[12] Adrian Johnstone and Elizabeth Scott. Generalised regular parsers. In Gorel Hedin, editor, *Compiler Construction, 12th Intnl. Conf, CC2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 232–246. Springer-Verlag, Berlin, 2003.

[13] Adrian Johnstone, Elizabeth Scott, and Tim Womack. Reverse compilation of Digital Signal Processor assembler source to ANSI-C. In *Proc Internat. Conference on Software Maintenance (ISCM'99)*, pages 316–325. IEEE, 1999.

[14] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[15] S. Klusener and R. Lämmel. Deriving tolerant grammars from a base-line grammar. In *Proc. International Conference on Software Maintenance (ICSM'03).* IEEE Computer Society Press, September 2003. 10 pages; To appear.

[16] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[17] Ralf Lämmel and Guido Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In Mark van den Brand and Didier Parigot, editors, *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA'01), Genova, Italy, April 7, 2001, Satellite event of ETAPS'2001*, volume 44 of *ENTCS*. Elsevier Science, April 2001.

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

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

[20] Jan G. Rekers. *Parser generation for interactive environments.* PhD thesis, Universty of Amsterdam, 1992.

[21] Georg Sander. *VCG Visualisation of Compiler Graphs.* Universität des Saarlandes, 66041 Saarbrücken, Germany, February 1995.

[22] E.A. Scott, A.I.C. Johnstone, and G.R. Economopoulos. BRN-table based GLR parsers. Technical Report TR-03-06, Royal Holloway, University of London, Computer Science Department, 2003.

[23] Elizabeth Scott and Adrian Johnstone. Reducing non-determinism in reduction modified GLR parsers. *To appear in: Acta Informatica*, 2004.

[24] Elizabeth Scott, Adrian Johnstone, and Shamsa Sadaf Hussain. Tomita-style generalised LR parsers. Technical Report TR-00-12, Royal Holloway, University of London, Computer Science Department, December 2000.

[25] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley Publishing Company, 1994.

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

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

[28] Eelco Visser. *Syntax definition for langauge prototyping.* PhD thesis, Universty of Amsterdam, 1997.

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