Available online at www.sciencedirect.com

SCIENCE @ DIRECT®

ELSEVIER

Science of Computer Programming 61 (2006) 228–244

Science of Computer Programming

www.elsevier.com/locate/scico

Evaluating GLR parsing algorithms

Adrian Johnstone*, Elizabeth Scott, Giorgios Economopoulos

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

Available online 19 May 2006

Abstract

We describe the behaviour of three variants of GLR parsing: (i) Farshi's original correction to Tomita's non-general algorithm; (ii) the Right Nulled GLR algorithm which provides a more efficient generalisation of Tomita and (iii) the Binary Right Nulled GLR algorithm, on three types of LR table. We present a guide to the parse-time behaviour of these algorithms which illustrates the inefficiencies in conventional Farshi-style GLR parsing. We also describe the tool GTB (Grammar Tool Box) which provides a platform for comparative studies of parsing algorithms; and use GTB to exercise the three GLR algorithms running with LR(0), SLR(1) and LR(1) tables for ANSI-C, ISO-Pascal and IBM VS-COBOL. We 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.
© 2006 Elsevier B.V. All rights reserved.

Keywords: GLR parsing; Grammar types; Context free languages; LR tables

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 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 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 and, with a little judicious use of prioritisation to disambiguate table conflicts, the well known generators such as YACC [11] and its derivatives GNU Bison and BTYACC [7] 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

* Corresponding author. Tel.: +44 0 1784 443425; fax: +44 0 1784 439786.
E-mail addresses: a.johnstone@rhul.ac.uk (A. Johnstone), e.scott@rhul.ac.uk (E. Scott).

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2006.04.004

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 conflicts. The effect from a user's 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 [12] such parsers can display surprising pathologies: in particular parser generators such as PRECC [4], PCCTS [22], ANTLR [21] and JAVACC [1] 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 the worst case.

A safer approach is to use one of the truly general context free parsing algorithms such as Earley [8], CYK [33] or a variant of Tomita's GLR algorithm [30]. 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 [31] stress generality and can parse ambiguous languages with support for *shared packed parse forests* (SPPF) 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 has recently acquired a general parsing mode. (It turns out that the present Bison 'GLR' implementation merely splits stacks when conflicts are encountered, so it displays exponential growth in memory requirements which makes it impractical; but the addition of this mode at least demonstrates the demand for general parsing.)

In recent years we have studied a wide variety of general parsing techniques along with mechanisms for extracting derivation forests; semi-automatic generation of abstract syntax trees; and dataflow analysis and code generation techniques based on the resulting structures [14]. 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. The Grammar Tool Box (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. In this paper we shall use GTB to describe the behaviour of three variants of GLR parsing: (i) Farshi's original correction to Tomita's non-general algorithm; (ii) the Right Nulled GLR algorithm which provides a more efficient generalisation of Tomita and (iii) the Binary Right Nulled GLR algorithm which achieves cubic performance on any context free grammar whilst retaining linear time efficiency when parsing deterministic grammars. We present a guide to the parse-time behaviour of these algorithms which illustrates the inefficiencies in conventional Farshi-style GLR parsing and exercise the three GLR algorithms running with LR(0), SLR(1), LALR and LR(1) tables for ANSI-C, ISO-Pascal and IBM VS-COBOL. We 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 Grammar Tool Box (GTB) and Parser Animation Tool (PAT)

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 Parser Animation Tool (PAT) [9] is an accompanying visualisation tool that has its own implementations of our parser algorithms which run from tables generated by GTB. PAT is written in Java, and GTB in C++ so they necessarily require separate implementations, but we have the added benefit of two authors constructing independent programs from the theoretical treatment. As might be expected, reconciliation of results has sometimes revealed bugs. In use PAT dynamically displays the construction of parse-time structures. PAT can also be used in batch mode to generate statistics on parser behaviour: the results given in Section 4.3 were obtained in this way.

2. LR parsing fundamentals

The standard LR parsing approach was introduced by Knuth [17]. The idea is to construct a deterministic finite state automaton (DFA) whose transitions are labelled with terminals and non-terminals from a context free grammar and whose states are labelled with sets of grammar rules. The standard LR parser then uses a stack and an input string to traverse the DFA. The state on the top of the stack is the current state. If there is a transition from this state labelled with the current input symbol then the target of this transition is pushed onto the top of the stack and the current input symbol is updated. This is referred to as a *shift* action. If the state at the top of the stack is labelled with a grammar rule $A ::= x_1 \dots x_j$ then the top j states are popped off the stack, leaving state h say on top, and then the target of the transition labelled A from h is pushed onto the stack (it is a property of the DFA that such a transition will always exist). This is referred to as a *reduction* action. The input is accepted if, when all the input has been read, the state on the top of the stack is the accepting state of the DFA. (See, for example, [2] for a full discussion of standard LR parsing.) Although the DFA itself is deterministic, the LR parsing process may not be. It is possible for there to be both a shift and a reduction associated with a DFA state, and a DFA state may be labelled with more than one rule, yielding a choice of reduction actions. These choices are called LR-parsing *conflicts*.

There are several classical classes of DFA which can be used as the basis of an LR parser, each constructed with different uses of input symbol lookahead. The LR(0) DFA does not use lookahead, the SLR(1) DFA uses one symbol of lookahead in a global fashion using the so-called *follow sets*, and the LR(1) DFA uses the lookahead symbols in a more detailed fashion which takes account of the local properties of the grammar. The LALR DFA [6] is obtained by merging some of the LR(1) DFA states. The classes of grammar whose LR(0), SLR(1), LALR, and LR(1) tables, respectively, are conflict-free form a strict hierarchy under inclusion. The standard LR parsing algorithm requires that there be no conflicts, thus using LR(1) DFAs permits the largest class of context free grammars to be parsed with the technique. However, while for a given grammar the LR(0), SLR(1) and LALR DFAs are the same size, the LR(1) DFA can be substantially larger.

2.1. LR tables in GTB

We follow Grune's treatment [10] of LR DFA 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 and a lookahead set representing those terminals which a parser may legally encounter during a parse. Setting F appropriately gives an LR(0), SLR(1) or LR(1) automaton. In [10] these relationships are emphasised by writing the grammar rules as an NFA in a natural way. 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 in GTB for production use.

The following GTB example illustrates the definition of a (tiny) language. In the GTB input format 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. The NFA and DFA are also output in a format which can be viewed using the VCG tool [24], as shown in Figs. 1 and 2.

```
S ::= 'a' | A B | A 'z'.
A ::= 'a'.
```

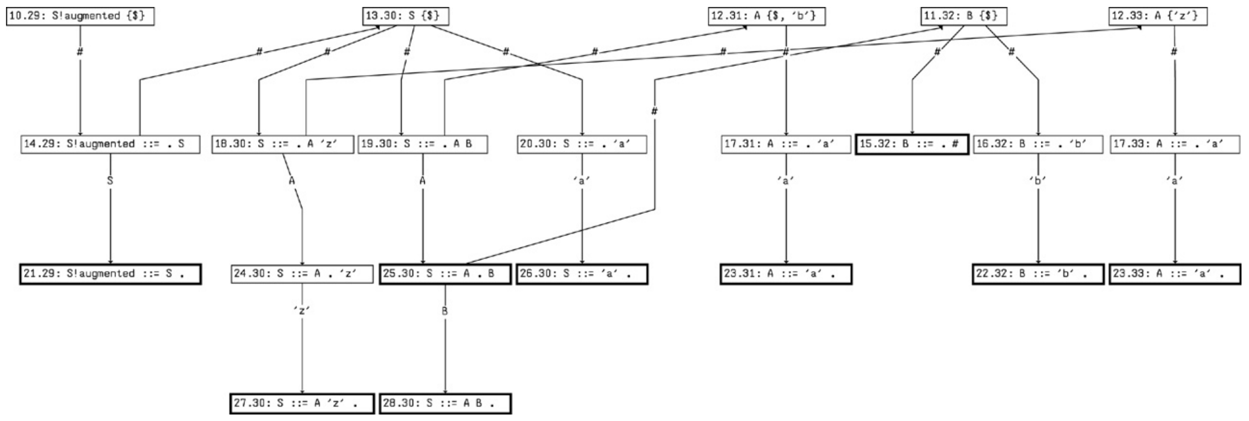


Fig. 1. Nondeterministic LR(1) automaton.

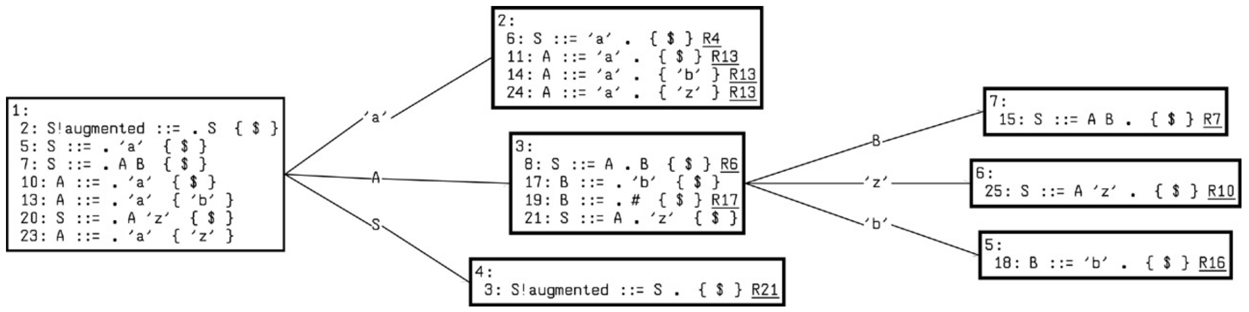


Fig. 2. Deterministic LR(1) automaton.

```
B ::= 'b' | #.

(
this_grammar := grammar[S]
this_nfa := nfa[this_grammar lr 1 nullable_reductions]
render[open["nfa.vcg"] this_nfa]
lr1_dfa := dfa[this_nfa]
lalr_dfa := la_merge[lr1_dfa]
this_derivation := rnglr_parse[lalr_dfa "a b"]
render[open["ssg.vcg"] this_derivation]
write["\n" CPU_time " CPU seconds elapsed\n\n"]
)
```

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 [3]). The built-in function `grammar` constructs a grammar object from a particular start rule and as a side effect calculates first and follow sets. 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 Horspool's parsing algorithm [26]. Follow sets are 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.) An LR DFA is obtained by running the subset construction on one of these NFAs. An LALR DFA can be constructed from an LR(1) DFA using the `la_merge` function. Once the DFA has been constructed the required parsing algorithm can be run. For example, `rnglr_parse` runs an RNGLR parser, and outputs a GSS in a format that can be viewed using VCG.

3. GLR parsing

GLR parsing, an extension of LR parsing, was introduced by Tomita [30] as a means for natural language parsers to capture all possible derivations when parsing potentially highly ambiguous grammars. Tomita-style GLR algorithms extend the LR parsing technique to cope with conflicts, thus GLR parsers can use any of the LR DFAs. However, we expect the efficiency of the GLR parser to be different on different types of DFA. The SLR(1) parser typically encounters more conflicts and thus may generate more parser activity, but the LR(1) DFA contains more states and thus the size of the runtime structures may be larger. In Section 4.3 we use GTB and PAT to investigate the effects of different DFA types on GLR parsers. First we give an outline of the GLR algorithms we use.

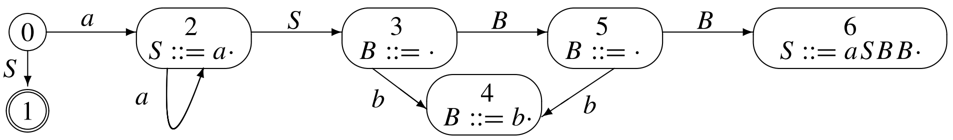
Nondeterminism in an LR parser gives rise to multiple current stacks. Tomita's algorithm constructs a graph structured stack (GSS) that efficiently represents these stacks. Tomita's approach centres around the *processing* of states currently on the tops of the stacks. We think of the GSS nodes corresponding to these states as the current *frontier* of the GSS, and the stacks are lock-stepped on shift actions so the GSS nodes fall into levels associated with each input symbol. In detail, as each node u on the frontier is processed each reduction $A ::= x_1 \dots x_j$ in the corresponding state (if there are any) is applied down all paths of length j from u. This can cause new states to be added to the frontier. When all the nodes on the frontier have been processed, a new frontier is begun by applying the shift actions associated with the next input symbol.

We describe the GLR algorithm using as a running example the grammar, Γ_1, whose rules are

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

Here S and B are non-terminals, a and b are terminals, ϵ is the empty string, and S is the start symbol.

We shall use the LR(0) DFA for Γ_1, shown below, but our discussion applies equally well to the SLR(1), LALR and LR(1) DFAs. Traditionally DFA states are labelled with sets of items (see Section 3.5) but we have only shown the reduction items. The start state is state 0 and the accepting state is state 1.

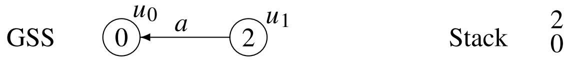


The DFA will be traversed in the standard way, reading an input string, in this example `aab`, and pushing states onto a stack. The first three steps of the algorithm, given in Section 3.1, illustrate the basic approach, including the stack sharing in the GSS. The last step generates multiple pending actions and at that point we discuss, Section 3.2, the use of *work lists* to manage the action application. The example has been chosen to illustrate the basic ideas, but there are problems with the approach. We shall illustrate these, and their solutions, in Section 3.3 using the same grammar with a different input string, `aaa`.

3.1. The basic approach

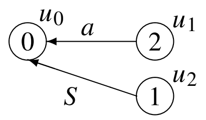
We traverse the above DFA for Γ_1 with the input string `aab`.

We read the first input symbol, a, begin in the start state, 0, and construct a GSS node u_0 labelled 0. There are no reductions from state 0 so we traverse the a-labelled arrow, perform the shift action and push onto the stack the state, 2, which is the target of the a-arrow from 0. Thus we create a new GSS node u_1 labelled 2 and an arrow from this to u_0 to indicate that 2 is above 0 in the stack.



(The label a on the arrow is not really part of the GSS but it has been included to make the exposition clearer.) We then read the next input symbol, a, and apply the reduction actions. State 2 includes a reduction using the rule $S ::= a$. To apply this reduction we need to trace back along all paths of length 1 from the node u_1, in this case we just get node u_0, and then create a new node u_2 labelled 1 and an edge (u_2, u_0). This corresponds to duplicating the stack and performing the corresponding pop and push operations.

GSS

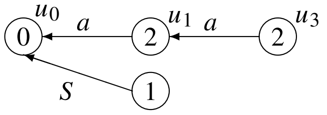


Stacks

2	1
0	0

State 1 has no reductions so the frontier is complete and we apply the shift actions. There is an a-arrow from DFA state 2 to itself, so we create a corresponding GSS node. There is no a-arrow from state 1 so the stack whose top is 1 dies.

GSS

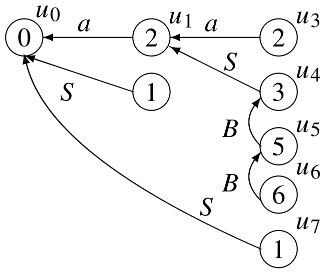


Stacks

2
2
0

Next we read the last input symbol, b. Applying the reduction from state 2 results in a GSS node u_4 labelled 3. State 3 includes a reduction using the rule $B ::= \epsilon$, and the path of length 0 ends at u_4, so we create a GSS node u_5 labelled 5 and an edge (u_5, u_4). State 5 also includes a reduction using $B ::= \epsilon$, so we create a GSS node u_6 labelled 6. State 6 has a reduction using $S ::= aSBB$, so we find all paths of length 4 from u_6. There is one such path which ends at u_0, and so we create a GSS node u_7 labelled 1.

GSS

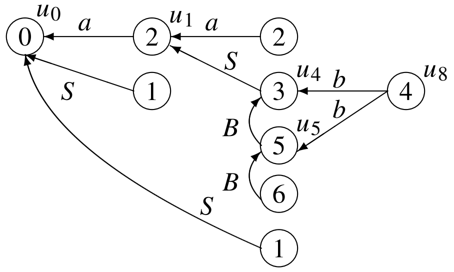


Stacks

			6	
		5	5	
2	3	3	3	
2	2	2	2	1
0	0	0	0	0

We have now applied all the reductions from all of the current stack tops, so we apply the shift actions and read the next input symbol which is the end-of-file symbol \$. When we apply the shift action from state 3 we create a new GSS node u_8 labelled 4. When we apply the shift action from state 5 we find that there is already a GSS node labelled 4 on the current frontier so we reuse this node. This is an example of the stack re-merging that we mentioned above.

GSS



Stacks

	4
4	5
3	3
2	2
0	0

So far in this example we have only had one frontier node to be processed at a time and each reduction has been applied down only one path, so we have not needed to address the book-keeping aspects of the process. To cope efficiently with multiple reductions we use a work list.

3.2. Work lists

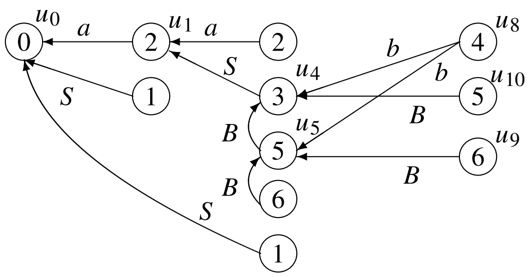
There is an important cost associated with the use of a GSS: in an LR parser a reduction is performed by popping j states from the stack where j is the length of the rule being reduced. This can be performed with a single modification of a stack pointer. In the GLR case a reduction involves a search down all pathways of length j, and in worst case this can require $O(2^j)$ time. Thus it is important that the searches are only applied once, and this implies that the algorithm must incorporate 'work lists' to ensure that previously carried out searches are not repeated. It is this *efficient construction of the GSS* which is the important contribution of Tomita's work.

Reductions of length greater than zero are applied down paths. We could keep a list of the nodes that have been created and then process each node, applying any reduction associated with the node down all the paths from the node. The problem is that, as we shall see later, additional edges can be added to a node after it has been processed. If we simply re-process the node we will retrace all the paths that we have already explored, as well as the new paths, leading to an unnecessary explosion in the searching cost of the algorithm. Tomita's approach is to keep a work list of the reductions together with the first edge down which they are to be applied. Then when a new edge is created the reductions are only applied down the new paths. In fact all we need to record is the length of the rule associated with the reduction action, the non-terminal on the left hand side of the rule, and the node at the end of the first edge down which the reduction is to be applied.

Thus we have a work list $\mathcal{R}$ which consists of triples (u, A, m) where u is a GSS node, A is the left hand side of a grammar rule and m is the length of the rule. When a node v is created, if there are any rules $B ::= \epsilon$ associated with this node then we add $(v, B, 0)$ to $\mathcal{R}$. When an edge (v, u) is created, if there are any rules $A ::= x_1 \dots x_m$, where $m \geq 1$, associated with v then we add (u, A, m) to $\mathcal{R}$.

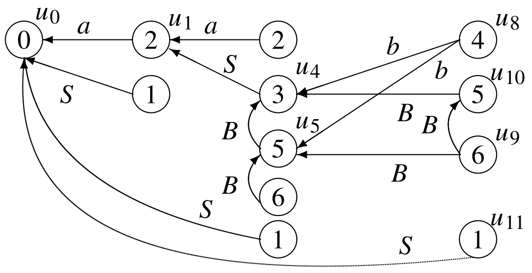
In addition we also keep a work list $\mathcal{Q}$ for the shift actions. The elements of $\mathcal{Q}$ are pairs (v, k) where v is a GSS node and k is a (integer) DFA state number. When a node v labelled h is created, if there is an a-transition from h to k, where a is the next input symbol, then (v, k) is added to $\mathcal{Q}$. The algorithm operates by creating the start node u_0 labelled 0 and then adding any appropriate elements to $\mathcal{R}$ and $\mathcal{Q}$.

Continuing with our running example above, when the shift actions are applied which create the edges (u_8, u_4) and (u_8, u_5), the elements $(u_4, B, 1)$ and $(u_5, B, 1)$ are added to $\mathcal{R}$. When we remove $(u_5, B, 1)$ from $\mathcal{R}$ we find all paths of length $1 - 1 = 0$ from u_5. This, of course, ends at u_5 so we create a new GSS node, u_9, labelled 6 and an edge (u_9, u_5). State 6 has an associated reduction on the rule $S ::= aSBB$ so on the creation of the edge (u_9, u_5) we add $(u_5, S, 4)$ to $\mathcal{R}$. Similarly removing $(u_4, B, 1)$ we create a new GSS node, u_{10}, labelled 5, an edge (u_{10}, u_4) and, since state 5 has a reduction on $B ::= \epsilon$, on the creation of u_{10} we add $(u_{10}, B, 0)$ to $\mathcal{R}$.



$\mathcal{R} = (\ (u_5, S, 4),\ (u_{10}, B, 0)\ )$

Removing $(u_5, S, 4)$ we look for paths of length 3 from u_5, and there is one which ends at u_0. From this we create a new node u_{11} labelled 1. Since there are no actions associated with state 1 no elements are added to $\mathcal{R}$. We then remove $(u_{10}, B, 0)$ from $\mathcal{R}$. There is already a node on the frontier labelled 6, u_9, so we add an edge (u_9, u_{10}) and the creation of this edge causes $(u_{10}, S, 4)$ to be added to $\mathcal{R}$.



$\mathcal{R} = (u_{10}, S, 4)$

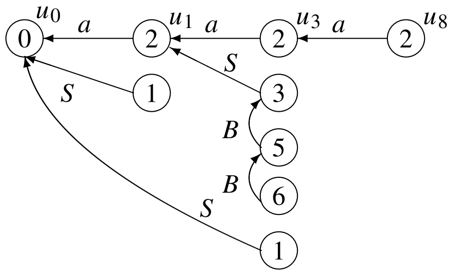
When $(u_{10}, S, 4)$ is removed from $\mathcal{R}$ we look for paths of length 3 from u_{10}. There is one which ends at u_0. Since there is already a frontier node, u_{11}, labelled 1 and an edge (u_{11}, u_0) no new nodes or edges are created, and the list $\mathcal{R}$ is empty. Since the input symbol is the end-of-string symbol, and there is a stack with the accepting state 1 on the top (i.e. there is a node labelled 1 on the current frontier), the input string is (correctly) accepted.

3.3. The failure of Tomita's algorithm

Tomita's basic algorithm [30], which we call Tomita-1, can be applied to all ϵ-free grammars. What we have described above is essentially an extension of Tomita-1 to include ϵ-rules in the obvious way, and we call this algorithm Tomita-1e. (Our formulation is slightly different from Tomita's in that Tomita's GSS included nodes for the grammar symbols as well as the DFA states. Also, Tomita did not use the optimisation of starting the reduction path searches from the second node on the path.)

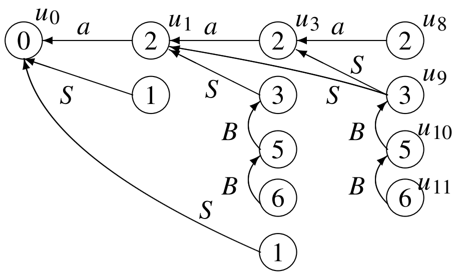
The problem is that Tomita-1e does not always work correctly on grammars with hidden *right* recursion. Consider again the above grammar Γ_1, this time with the input string aaa.

Reading the first three input symbols and proceeding as above, after we perform the shift action associated with the third a we have the following GSS and work list $\mathcal{R}$.



$$\mathcal{R} = (u_3, S, 1)$$

Removing $(u_3, S, 1)$ from $\mathcal{R}$ results in the creation of a GSS node u_9 labelled 3 and an edge (u_9, u_3), and $(u_9, B, 0)$ is added to $\mathcal{R}$. When this element is removed we create u_{10} labelled 5 and the element $(u_{10}, B, 0)$, and then in turn we create u_{11} labelled 6 and the element $(u_{10}, S, 4)$. Processing $(u_{10}, S, 4)$ we find the path of length three from u_{10} to u_1 and we create an edge (u_9, u_1).



Since state 3 has only a reduction of length 0, there are no elements added to $\mathcal{R}$ as a result of the creation of the edge (u_9, u_1) thus the work lists $\mathcal{R}$ and $\mathcal{Q}$ are both empty and the process terminates. Since there is no node labelled 1 on the current frontier, the input string is (incorrectly) rejected.

The problem is that the edge (u_9, u_1) creates a new path of length 4 from u_{11} and so the reduction $S ::= aSBB$ should be applied down this path. The method of storing reductions on a work list with the first edge of the path down which they are to be applied only works if new paths are only created by adding edges at the front of existing paths. Tomita was aware of this and modified his algorithm to try to ensure this property held, but the modification caused the construction of an infinite GSS in certain cases. A different solution was proposed by Farshi [20].

3.4. Farshi's algorithm

Farshi extended Tomita-1 to include ϵ-rules and added a brute force search to ensure that rules are correctly applied down newly created paths. The algorithm uses slightly different work lists from those we have described. There is a work list $\mathcal{A}$ of frontier nodes waiting to be processed, and when a node is created it is added to $\mathcal{A}$. When a node v, whose label is h say, is removed from $\mathcal{A}$, if there is a DFA transition from h to k labelled with the current input symbol then, as for Tomita-1e, the pair (v, k) is added to a work list $\mathcal{Q}$.

Reduction actions are treated slightly differently. For each rule $D ::= x_1 \dots x_m$ in state h (if there are any) all the GSS nodes at the ends of paths of length m from v are found and for each such node w the pair (w, D) is added to $\mathcal{R}$.

There is a state, t say, which is the target of an D-transition from l, where l is the label of w, and when (w, D) is removed from $\mathcal{R}$, if there is no frontier node labelled t, then one is created and added to $\mathcal{A}$, and an edge is created from this node to w.

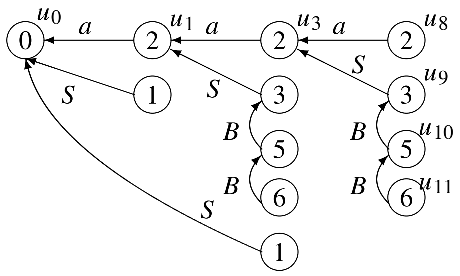
Farshi's approach results in the frontier nodes being created in a different order to Tomita's but the method is essentially the same and it does not solve the basic problem with Tomita's algorithm. To address this Farshi added an additional step to the processing part of his algorithm. A new edge in the middle of an existing path can only be created when processing an element (w, D) from $\mathcal{R}$. Thus there is an additional instruction when such an edge, (u, w) say, is created. This instruction says:

If an edge (u, w) is created from an existing node u, then for each node v on the current frontier which is not in $\mathcal{A}$, for each rule $B ::= x_1 \dots x_j$ associated with v, and for each path of length j from v which includes the edge (u, w), add (y, B) to $\mathcal{R}$, where y is the node at the end of the path.

This does solve the problem with Tomita's algorithm, but at a very high cost. Farshi's algorithm does not say how paths which include a particular edge are to be found, and in the limit, all paths of length j must be re-traced to determine whether or not they contain the edge. This is exactly the potentially explosive repeated searching that Tomita's algorithm was designed to avoid.

We illustrate Farshi's approach by returning to the above example and the string aaa, at the point where the shift on the third a is applied. The node u_8 is created and added to $\mathcal{A}$. This node is removed and, for the rule $S ::= a$ which is associated with state 4, Farshi's algorithm finds the nodes at the end of paths of length 1 from u_8, in this case u_3, and adds (u_3, S) to $\mathcal{R}$. Next (u_3, S) is removed from $\mathcal{R}$ and processed. This results in the creation of u_9 labelled 3, which is added to $\mathcal{A}$, and the creation of the edge (u_9, u_3).

Now u_9 is removed from $\mathcal{A}$ and, because $B ::= \epsilon$ is associated with state 3, we add (u_9, B) to $\mathcal{R}$. When this element is removed a node u_{10} labelled 5 is created. When u_{10} is processed (u_{10}, B) is added to $\mathcal{R}$, and then processing this results in the creation of u_{11} labelled 6 and the edge (u_{11}, u_{10}). When u_{11} is processed, because $S ::= aSBB$ is associated with state 6, we find the nodes at the end of paths of length 4 from u_{11} and hence add (u_1, S) to $\mathcal{R}$.



$\mathcal{R} = (u_1, S)$

When (u_1, S) is processed we find that there is already a node u_9 labelled 3, so we just create an edge (u_9, u_1). At this point the additional step in Farshi's process is triggered. The nodes u_8, u_9, u_{10} and u_{11} are no longer in $\mathcal{A}$, thus paths of length 1 from u_8 and of length 4 from u_{11} are looked for again. The path from u_8 and one of the paths from u_{11} do not contain the new edge, so no elements are added to $\mathcal{R}$. But we also find an additional path of length 4 from u_{11} which does contain (u_9, u_1). The end of this path is u_0 so (u_0, S) is added to $\mathcal{R}$. Thus we have found the extra reduction application that Tomita's algorithm missed, but note that in the process we have had to trace one of the paths of length 4 from u_{11} twice.

Finally removing (u_0, S) from $\mathcal{R}$ we create a new frontier node u_{12} labelled 1 and an edge (u_{12}, u_0). At this point all of the work lists are empty so the algorithm terminates and, since there is a frontier node labelled 1, the algorithm (correctly) accepts the input string.

In this example, Farshi's modification only required two paths, one of length 4 and one of length 1, to be traversed twice. But it is not hard to see that on the input string aaaa a similar pattern will appear but two extra edges will be added to the node labelled 3 on the last frontier. The creation of the second edge will cause a path of length 4 to be retraced, and the creation of the third will cause two paths of length 4 to be retraced. Thus we will have a total of four extra length 4 path traversals. For input aaaaa the total is 10, and so on.

In the above discussion we have described Farshi's algorithm as it is described in [20]. Of course we can limit the length of the re-searching carried out using some observations about the creation of new edges. As written, Farshi's algorithm requires us to search for all paths containing the new edge. However, the structure of the algorithm ensures that any new edge will have its source node on the current frontier. Thus when searching for new paths which contain the edge we can terminate the search down each path when we reach a node which is not on the current frontier.

In our experiments we have implemented Farshi's algorithm exactly as it is described in [20], and we have also implemented a more efficient version which truncates the searches as described above. We refer to the former as Farshi-naïve and the latter as Farshi-opt, and we have presented the results for both of these algorithms.

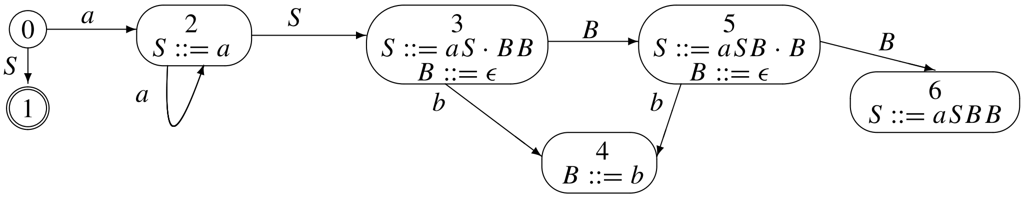
3.5. The RNGLR algorithm

The RNGLR algorithm [27] provides a different solution to the problem with Tomita-1e, which involves changing the DFA rather than the algorithm.

If an edge (u, w) is added to an existing GSS node u, then u is on the current frontier. So if there is a path $u_j, \ldots, u_p, u, w, u_{p-3}, \ldots, u_1$ in the GSS then $u_j, \ldots, u_p$ are also on the current frontier. It is not too hard to see that if a reduction $A ::= x_1 \ldots x_j$ has to be applied down this path then we must have $x_r \overset{*}{\Rightarrow} \epsilon$, for $p \leq r \leq j$. Thus rather than applying the reduction $A ::= x_1 \ldots x_j$ from the node u_j we apply the reduction $A ::= x_1 \ldots x_{p-1}$ from u. One of the beauties of this approach is that it can be achieved simply by adding some extra reductions to the DFA states.

The standard DFA construction process labels the DFA states with *items* which are grammar rules with a 'dot' on the right hand side. The rules that we have associated with the states in the above description have the dot at the end, $A ::= x_1 \ldots x_j \cdot$, and our additional rules have the dot at the point where the limited reduction is to be applied, $A ::= x_1 \ldots x_p \cdot x_p \ldots x_j$. We call the DFAs with these additional reductions *right nulled* (RN) DFAs. They are constructed in exactly the same way as traditional LR DFAs, and then additional items are nominated as reduction items.

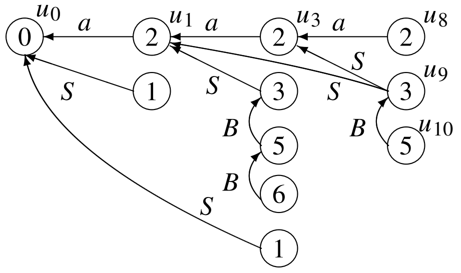
The RN LR(0) DFA for the grammar Γ_1 in our running example is



To illustrate the effect of the RN DFA we consider again the above example and the string aaa, and take up the process at the point where the shift on the third a is applied.

Again the node u_8 and edge (u_8, u_3) are created, causing $(u_3, S, 1)$ to be added to $\mathcal{R}$. When this element is removed from $\mathcal{R}$ a node u_9 labelled 3 and an edge (u_9, u_3) are created and, because both $B ::= \epsilon$ and $S ::= aS \cdot BB$ are associated with state 3, $(u_9, B, 0)$ and $(u_3, S, 2)$ are added to $\mathcal{R}$. (It is the addition of the latter element which allows the parser to eventually terminate correctly.)

As in the previous cases, processing $(u_9, B, 0)$ causes a node u_{10} labelled 5 to be created, and as a result $(u_{10}, B, 0)$ and $(u_9, S, 3)$ are added to $\mathcal{R}$. Processing $(u_3, S, 2)$ requires us to look for paths of length one from u_3, and there is one which ends at u_1. Since there is already a frontier node labelled 3 we just add a new edge (u_9, u_1) and then an element $(u_1, S, 2)$ to $\mathcal{R}$.



$\mathcal{R} = (\ (u_{10}, B, 0), (u_9, S, 3), (u_1, S, 2)\ )$

Processing $(u_{10}, B, 0)$ creates a node, u_{11}, labelled 6 and causes $(u_{10}, S, 4)$ to be added to $\mathcal{R}$. Processing $(u_9, S, 3)$, we look for nodes at the end of paths of length 2 from u_9. We find u_1 and u_0. An edge to u_1 of the required type already exists so we take no further action. For u_0 we create a frontier node u_{12} labelled 1. Finally, processing $(u_1, S, 2)$ and $(u_{10}, S, 4)$ does not result in any new GSS edges, so the process is complete and has constructed the same final GSS as Farshi's algorithm did.

In fact it turns out that because we have added the additional reductions it is not necessary to apply any of the reductions down paths whose first edge lies between two nodes on the current frontier. This is because a limited version of the reduction will be applied at an earlier point in the algorithm. This allows us to present a slightly different version of Tomita-1e which does not require any additional searching, and in fact is even slightly more efficient than Tomita-1e. (Specifically, in the above example the elements $(u_9, S, 3)$ and $(u_{10}, S, 4)$ will not actually be added to $\mathcal{R}$.) We call this algorithm the RNGLR algorithm, and the RNGLR approach is discussed in detail in [27] and [28].

For completeness we include, without further discussion, the formal version of the RNGLR algorithm.

Input: an RN DFA written as a table $\mathcal{T}$, an input string $a_1 \dots a_d$

```
PARSER {
 **if** $d = 0$ { **if** $acc \in T(0, \$)$ report success  **else** report failure }
 **else** {
   create a node $v_0$ labelled with the start state 0 of the DFA.
   set $U_0 = \{v_0\}$, $R = \emptyset$, $Q = \emptyset$, $a_{d+1} = \$$, $U_1 = \emptyset, ..., U_d = \emptyset$
   **if** $pk \in T(0, a_1)$ add $(v_0, k)$ to $Q$
   **forall** $r(X, 0) \in T(0, a_1)$ add $(v_0, X, 0)$ to $R$
   **for** $i = 0$ to $d$  **while** $U_i \neq \emptyset$ { **while** $R \neq \emptyset$ { REDUCER($i$) }
                                              SHIFTER(i) }
   **if** the DFA final accepting state is in $U_d$ report success **else** report failure }}

REDUCER($i$) {
 remove $(v, X, m)$ from $R$
 find the set $\chi$ of nodes which can be reached from $v$ along a path of
      length $(m - 1)$, or length 0 if $m = 0$
 **for** each node $u \in \chi$  do {
   let $k$ be the label of $u$ and let $pl \in T(k, X)$
   **if** there is a node $w \in U_i$ with label $l$  {
     **if** there is not an edge from $w$ to $u$  {
       create an edge from $(w, u)$
       **if** $m \neq 0$ { **forall** $r(B, t) \in T(l, a_{i+1})$ where $t \neq 0$, add $(u, B, t)$ to $R$ }}}
   **else** {
     create a node $w \in U_i$ labelled $l$ and an edge $(w, u)$
     **if** $ph \in T(l, a_{i+1})$ add $(w, h)$ to $Q$
     **forall** $r(B, 0) \in T(l, a_{i+1})$ add $(w, B, 0)$ to $R$
     **if** $m \neq 0$ { **forall** $r(B, t) \in T(l, a_{i+1})$ where $t \neq 0$, add $(u, B, t)$ to $R$ }}}}

SHIFTER($i$) {
 **if** $i \neq d$ {
   $Q' = \emptyset$
   **while** $Q \neq \emptyset$ {
     remove an element $(v, k)$ from $Q$
     **if** there is $w \in U_{i+1}$ with label $k$  {
          create an edge from $(w, v)$
          **forall** $r(B, t) \in T(k, a_{i+2})$ where $t \neq 0$ add $(v, B, t)$ to $R$}
     **else** { create a node, $w \in U_{i+1}$ labelled $k$ and an edge $(w, v)$
            **if** $ph \in T(k, a_{i+2})$ add $(w, h)$ to $Q'$
            **forall** $r(B, t) \in T(k, a_{i+2})$ where $t \neq 0$ add $(v, B, t)$ to $R$
            **forall** $r(B, 0) \in T(k, a_{i+2})$ add $(w, B, 0)$ to $R$}}
  copy $Q'$ into $Q$ }}
```

3.6. The BRNGLR algorithm

The Tomita-1e and RNGLR algorithms are of worst case order $O(n^{k+1})$ where k is the length of the longest rule in the grammar, and the order of Farshi's algorithm can be even higher. The run-time costs come from the fact that in the limit there can be m^k paths of length k from a node on the mth frontier and these paths must all be traversed when a reduction is carried out.

It is possible to turn the RNGLR algorithm into a cubic algorithm by performing the searches for paths of length k in $k-1$ steps of length 2. For example, in the basic RNGLR algorithm if an edge is the first edge of m^{k-1} paths from a node then it can be traversed m^{k-1} times. Traversing the paths in steps of length two ensures that each edge is only traversed at most $O(n)$ times for an input sting of length n. We call this two step version the (Binary) BRNGLR algorithm, and it is described in detail in [25]. Below, we use GTB to compare the performance of the Farshi, RNGLR and BRNGLR algorithms.

3.7. From recognition to parsing

Tomita designed his algorithms to allow straightforward generation of an SPPF, a representation of the (potentially infinite) set of derivation trees for an input string. It is possible for derivation tree generation to significantly increase the run-time cost and space requirements of a recognition algorithm, but Tomita designed the SPPF generation to be as efficient as the recogniser part of the algorithm. In particular, Tomita's GSS nodes have intervening 'symbol' nodes which correspond to nodes in the SPPF.

Farshi described only a recogniser version of his algorithm. Rekers [23] turned Farshi's algorithm into a parser in which the GSS edges are labelled with SPPF nodes. Visser [32] uses a generally similar approach but incorporates lexical processing directly into the parser and its associated GSS and SPPF. The RNGLR algorithm uses Rekers' approach to SPPF generation because, although requiring a little more searching, Rekers' approach results in more compact SPPFs, see [27] for more discussion of this. In Section 4.3 we report on the both the GSS and SPPF sizes.

4. Experiments

The complete tool chain for these experiments comprises `ebnf2bnf`, a tool for converting extended BNF grammars to BNF under the control of annotations that specify the type of expansion to be applied; GTB which is used to analyse grammars and construct tables and PAT which is used to animate algorithms and generate statistics on parse-time structure size. We report results for Pascal, C and COBOL along with small grammars that display pathological behaviour with some algorithms.

4.1. 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 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' [29, 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 [15] and a grammar for IBM VS-COBOL developed in Amsterdam. The original extraction of this grammar is described at length in [18] and some interesting work on techniques to generate tolerant variants is described in [16]. 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 5000 tokens; for ANSI-C a Boolean equation minimiser of about 4500 tokens; and for COBOL, strings of approximately 2000 tokens extracted from the test set supplied with the grammar.

4.2. 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* [5] 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. 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 [19]: the authors describe EBNF to BNF conversion as Yacc-ification. Our tool emphasises ease-of-use and traceability of the basic operations.

4.3. Results

We run our experiments on grammars as described above for Pascal, C, and COBOL, and on the grammar Γ_1 from Section 3. The latter grammar displays much worse behaviour on the LR(0) DFA than on the SLR(1) and LR(1) DFAs, illustrating the potential effects of using lookahead to improve efficiency, even when the parsing technique does not *require* lookahead to ensure determinism. We also consider two minor variants of Γ_1 which illustrate situations in which LR(1) is a much better choice than SLR(1), and in which all three DFA types are equally bad.

$$\begin{array}{llll} \Gamma_2 : & S ::= T \mid b\, T\, a & T ::= a\, T\, B\, B \mid a & B ::= b \mid \epsilon \\ \Gamma_3 : & S ::= T\, a & T ::= a\, T\, B\, B \mid a & B ::= b \mid \epsilon \end{array}$$

Table 1
The size of LR and RNLR parse tables

	Pascal	ANSI-C	COBOL
LR(0)/SLR(1)/LALR	434 × 286	383 × 158	2692 × 1028
LR(1)	2608 × 286	1797 × 158	–

Table 2
The number of conflict cells in LR and RNLR parse tables

	Pascal	ANSI-C	COBOL
LR(0)/RNLR(0)	768/4097	414/414	131,506/167,973
SLR(1)/RNSLR(1)	1/242	88/88	65,913/73,003
LALR/RNLALR	1/233	75/75	–
LR(1)/RNLR(1)	2/1104	421/421	–

Table 3
The size of GSS and SPPF structures

	RNGLR/Farshi GSS edges	BRNGLR GSS edges	RNGLR/Farshi SPPF edges	BRNGLR SPPF edges
Pascal LR(0)	31,015	34,441	35,350	38,770
Pascal SLR(1)	21,258	23,826	23,325	25,891
Pascal LALR	21,258	23,826	23,325	25,891
Pascal LR(1)	21,135	23,655	23,050	25,572
C LR(0)	39,389	41,748	40,454	42,800
C SLR(1)	28,604	30,670	29,033	31,092
C LALR	28,476	30,542	28,906	30,956
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

We consider first the size of the parse tables required for our three large grammars. For a given combination of grammar and LR 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 shows the sizes of the tables given as the number of rows (states) by the number of columns (grammar symbols), and Table 2 shows the number of table cells with conflicts (multiple entries). As an aside, the LR(1) tables sometimes show more conflict cells than the SLR(1). This is because the LR(1) DFA contains multiple copies of a given SLR(1) DFA state, corresponding to different local follow sets. The tables for ANSI-C show a perhaps unexpectedly high number of conflicts for a grammar that is known to produce only one conflict with YACC: however that result relies on the use of the so-called C lexer hack which maps type names defined by `typedef` statements to a special token. Our figures here relate to the grammar used as-is without any lexical trickery.

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 [18].

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 3 shows expansions of between 5 and 20% in the size of the structures.

Table 4
GSS size for pathological grammars

	Γ_1, a^{20}	Γ_2, a^{20}	Γ_3, a^{20}	Γ_1, a^{1000}	Γ_2, a^{1000}	Γ_3, a^{1000}
LR(0)	268	288	306	503,498	504,498	505,496
SLR(1)	42	269	266	2,002	503,499	503,496
LR(1)	44	45	300	2,004	2,005	505,490

Table 5
Performance

GSS edge visits	Farshi-naïve	Farshi-opt	RNGLR	BRNGLR
Pascal LR(0)	41,460	38,964	8,556	8,550
Pascal SLR(1)	25,753	24,100	5,665	5,663
Pascal LALR	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 LALR	30,096	29,801	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
Γ_1 LR(0)	334,831,502	168,664,502	499,500	499,500
Γ_1 SLR(1)	1,001,998	503,497	999	999
Γ_1 LR(1)	1,000,001	502,498	999	999
Γ_2 LR(0)	334,832,502	168,665,502	499,500	499,500
Γ_2 SLR(1)	334,831,503	168,664,503	499,500	499,500
Γ_2 LR(1)	1,000,001	502,449	999	999
Γ_3 LR(0)	335,831,500	169,165,999	500,499	500,499
Γ_3 SLR(1)	333,829,507	168,161,008	498,502	498,502
Γ_3 LR(1)	332,833,504	167,662,508	498,502	498,502

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 GSSs 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) GSSs 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.

However, relatively simple grammars can show pathological behaviours. Table 4 shows the different sizes of GSS for the three types of DFA on the grammars Γ_1, Γ_2 and Γ_3 for two input strings a^{20} and a^{1000}. The figures are the number of edges in the GSS generated by the RNGLR and Farshi algorithms, and we see that while for Γ_1 the SLR(1) DFA is comparable with the LR(1) DFA, for Γ_2 the LR(1) DFA is better, and for Γ_3 the LR(0) DFA based parser is comparable with the other two.

Now we turn to performance. As we have already discussed in Section 3, the asymptotic time order of the 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. Table 5 presents figures for four algorithms: BRNGLR, RNGLR and two variants of Farshi's algorithm described in Section 3.4. The figures for Γ_1, Γ_2 and Γ_3 are for the string a^{1000}.

We see clear performance advantages arising from the RNGLR algorithm, both for the small grammars and for COBOL where there is an order of magnitude difference in the search costs. We also see that the difference between SLR(1), LALR and LR(1) is insignificant for Pascal and C, but Γ_2 shows the massive differences in run time costs that can be made by using an LR(1) grammar in some circumstances.

5. Conclusions and acknowledgements

GTB has allowed us to make direct comparisons between three different GLR-style algorithms and four types of LR table. To summarise: for typical (mostly deterministic) grammars the results show that performance is rather insensitive to the choice between LR(1), LALR and SLR(1) tables. The RNGLR algorithm shows very significant improvements over the traditional Farshi approach, and (although not demonstrated here) BRNGLR offers further improvements for grammars which trigger supra-cubic behaviour in the RNGLR algorithm [13].

For Pascal the RNGLR algorithm shows a factor greater than 4 improvement over Farshi's algorithm regardless of table type; for ANSI-C better than 6.5 and for COBOL better than 10. These are very encouraging results although they represent only a few sample input strings. The BRNGLR begins to show further significant improvements over RNGLR in the case of the COBOL grammar; but we should note that the overheads for this algorithm are higher and so the improvement may only be real for very long input strings.

With regard to table choice, in all cases the LR(1) table resulted in smaller and faster run-time parsers, but the improvement over LALR is marginal, and indeed even the SLR(1) tables yield results within a few percent of the LR(1) figures. The LR(1) tables are, of course, significantly larger.

The size of the structures is only one indication of performance. We expect SLR(1) GLR parsers to perform more reduction searches than LR(1) because of the reduced lookahead resolution. In fact, for the limited examples here we find that LR(1), LALR and SLR(1) parsers perform very similar numbers of edge visits. The Pascal and ANSI-C grammars are mostly deterministic whatever the parsing model, so we should not be too surprised by this. We would expect more non-deterministic grammars to show divergent results, favouring the LR(1) model as shown clearly by the results for Γ_2.

Given the relative insensitivity to table type of parse-time space and time requirements for current conventional programming language grammars, the evidence presented here indicates that users should deploy either SLR(1) or LALR tables depending on which is most convenient for them, and only use LR(1) tables for very long strings where the (small) improvement in the size of the parse time structures outweighs the increased table size.

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 many helpful suggestions.

References

[1] JAVACC project home page, https://javacc.dev.java.net/, Last visited: December 2004.
[2] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles Techniques and Tools, Addison-Wesley, 1986.
[3] John Aycock, Nigel Horspool, Faster generalised LR parsing, in: Compiler Construction, 8th Intnl. Conf, CC'99, in: Lecture Notes in Computer Science, vol. 1575, Springer-Verlag, 1999, pp. 32–46.
[4] Peter T. Breuer, Jonathan P Bowen, A PREttier compiler-compiler: Generating higher-order parsers in C, Software Practice and Experience 25 (11) (1995) 1263–1297.
[5] Robert D. Cameron, Extending context-free grammars with permutation phrases, ACM Letters on Programming Languages and Systems 2 (1–4) (1993) 85–94.
[6] Franklin L. DeRemer, Simple LR(k) grammars, Communications of the ACM 14 (7) (1971) 453–460.
[7] Chris Dodd, Vadim Maslov, http://www.siber.com/btyacc, June 2002.
[8] J. Earley, An efficient context-free parsing algorithm, Communications of the ACM 13 (2) (1970) 94–102.
[9] Giorgios Robert, Economopoulos. Generalised LR parsing algorithms. Ph.D. Thesis, Royal Holloway, University of London, 2005.
[10] Dick Grune, Ceriel Jacobs, Parsing Techniques: A Practial Guide, Ellis Horwood, Chichester, England, 1990, See also http://www.cs.vu.nl/~dick/PTAPG.html.
[11] S.C. Johnson, Yacc—yet another compiler-compiler, Technical Report 32, AT&T Bell Laboratories, 1975.
[12] Adrian Johnstone, Elizabeth Scott, Generalised recursive descent parsing and follow determinism, in: Kai Koskimies (Ed.), Proc. 7th Intnl. Conf. Compiler Construction, CC'98, in: Lecture Notes in Computer Science, vol. 1383, Springer, Berlin, 1998, pp. 16–30.
[13] Adrian Johnstone, Elizabeth Scott, Giorgios Economopoulos, Generalised parsing: Some costs, in: Evelyn Duesterwald (Ed.), Compiler Construction, 13th Intnl. Conf, CC'04, in: Lecture Notes in Computer Science, vol. 2985, Springer-Verlag, Berlin, 2004, pp. 89–103.

[14] Adrian Johnstone, Elizabeth Scott, Tim Womack, Reverse compilation of digital signal processor assembler source to ANSI-C, in: Proc Internat. Conference on Software Maintenance, ISCM'99, IEEE, 1999, pp. 316–325.
[15] Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language, second edn., Prentice Hall, 1988.
[16] S. Klusener, 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.
[17] Donald E. Knuth, On the translation of languages from left to right, Information and Control 8 (6) (1965) 607–639.
[18] R. Lämmel, C. Verhoef, Semi-automatic grammar recovery, Software—Practice & Experience 31 (15) (2001) 1395–1438.
[19] Ralf Lämmel, Guido Wachsmuth, Transformation of SDF syntax definitions in the ASF + SDF Meta-Environment, in: Mark van den Brand, Didier Parigot (Eds.), Proceedings of the First Workshop on Language Descriptions, Tools and Applications, LDTA'01, Genova, Italy, 7 April 2001, Satellite event of ETAPS'2001, in: ENTCS, vol. 44, Elsevier Science, 2001.
[20] Rahman Nozohoor-Farshi, GLR parsing for ϵ-grammars, in: Masaru Tomita (Ed.), Generalized LR Parsing, Kluwer Academic Publishers, Netherlands, 1991, pp. 60–75.
[21] Terence Parr, ANTLR home page. http://www.antlr.org, Last visited: December 2004.
[22] Terence John Parr, Language translation using PCCTS and C++, Automata Publishing Company, 1996.
[23] Jan G. Rekers, Parser generation for interactive environments. Ph.D. Thesis, Universty of Amsterdam, 1992.
[24] Georg Sander, VCG Visualisation of Compiler Graphs, Universität des Saarlandes, 66041 Saarbrücken, Germany, February 1995.
[25] E.A. Scott, A.I.C. Johnstone, G.R. Economopoulos, BRN-table based GLR parsers, Technical Report TR-03-06, Computer Science Department, Royal Holloway, University of London, London, 2003.
[26] Elizabeth Scott, Adrian Johnstone, Generalised bottom up parsers with reduced stack activity, The Computer Journal 58 (5) (2005) 565–587.
[27] Elizabeth Scott, Adrian Johnstone, Right nulled GLR parsers, ACM Transactions on Programming Languages and Systems (in press).
[28] Elizabeth Scott, Adrian Johnstone, Shamsa Sadaf Hussain, Tomita-style generalised LR parsers, Technical Report TR-00-12, Computer Science Department, Royal Holloway, University of London, London, December 2000.
[29] Bjarne Stroustrup, The Design and Evolution of C++, Addison-Wesley Publishing Company, 1994.
[30] Masaru Tomita, Efficient Parsing for Natural Language, Kluwer Academic Publishers, Boston, 1986.
[31] M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier, Compiling language definitions: The ASF + SDF compiler, ACM Transactions on Programming Languages and Systems 24 (4) (2002) 334–368.
[32] Eelco Visser, Syntax definition for langauge prototyping. Ph.D. Thesis, Universty of Amsterdam, 1997.
[33] D.H. Younger, Recognition of context-free languages in time n^3, Information and Control 10 (2) (1967) 189–208.