Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science 141 (2005) 143–160
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Recursion Engineering for Reduction Incorporated Parsers

Adrian Johnstone[1] Elizabeth Scott [2]

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

Abstract

Reduction Incorporated (RI) recognisers and parsers deliver high performance by suppressing the stack activity except for those rules that generate fully embedded recursion. Automaton constructions for RI parsing have been presented by Aycock and Horspool [3] and by Scott and Johnstone [7] but both can yield very large tables. An unusual aspect of the RI automaton is that the degree of stack activity suppression can be varied in a fine-grained way, and this provides a large family of potential RI automata for real programming languages, some of which have manageable table size but still show high performance. We give examples drawn from ANSI-C, Cobol and Pascal; discuss some heuristics for guiding manual specification of stack activity suppression; and describe work in progress on the automatic construction of RI automata using profiling information gathered from running parsers: in this way we propose to optimise our parsers' table size against performance on actual parsing tasks.

Keywords: RI parsing, recursion analysis, context free languages

1 Introduction

Reduction Incorporated (RI) parsers in principle allow regular parts of a grammar to be parsed using regular automata with stack activity only being triggered for rules that generate embedded recursion. This can yield fast parsers that still construct derivations in terms of the source grammar.

The basic idea is that where possible we effectively back-substitute rules and represent reductions as ϵ-transitions: thus directly incorporating reduc-

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

doi:10.1016/j.entcs.2005.02.060

tions into the parsing automaton rather than treating them as actions associated with state labels, as we do for traditional Knuth style LR-parsing. In general, we can only do this if the grammar is regular. To handle embedded recursion, we construct a family of regular sub-automata and use a stack to manage nested calls between them: a call graph in the form of a Graph Structured Stack can be used to manage the (potentially many) stacks required, giving a general parsing algorithm that has some affinity with Tomita's generalised LR (GLR) parser.

An unexpected aspect of RI parsers is that automaton size can be traded for run time performance in quite a fine grained way, and in fact we believe that practical adoption of this technology will require engineering trade-offs because for real programming languages the fastest RI parsers have extremely large automata. It turns out that there are much smaller automata that are 'nearby' to these very large automata that have sizes commensurate with Knuth style automata, but whose performance that is close to the best RI automaton.

This paper is about exploring the space of RI automata for a given grammar. We shall give examples from ANSI-C, IBM VS-COBOL and Pascal which show that making small compromises in the amount of stack activity suppression can drastically reduce automaton size whilst not significantly affecting run time stack activity. We shall describe some heuristics that may be used to guide the specification of these automata but our main goal is to develop a tool that automatically optimises RI automaton size for a given application. We shall report on work-in-progress in the automatic derivation of these parsers giving algorithms and some preliminary indications of computational feasibility.

An interesting aspect is that call graph size is, of course, a function of both the grammar and the string to be parsed, so trading automaton size for run time performance can be improved if we have statistics on the relative frequency of rule activations and reductions: rules for rarely used parts of a language can be allowed to generate stack activity at low average run-time, and this may allow major reductions in automaton size.

Our approach to fully automatic generation of optimised RI automata falls into three phases.

(i) Automatic discovery of minimal and near-minimal terminalisations, with the potential for exhaustive enumeration of terminalisations for moderately sized grammars.

(ii) Static characterisation of extensions to minimal terminalisations by measuring the size of the associated automaton.

(iii) Dynamic characterisation of terminalisations by feeding back profiling data generated from instrumented versions of the RI recogniser which allow automata that are potentially far from minimal to be used without in practice impacting performance.

We also present results concerning so-called *scannerless parsing* which is used for instance in ASF+SDF. We expect the RIGLR algorithm to have an advantage over the RNGLR algorithm on grammars which contain large sub grammars without self embedding, and grammars for scannerless parsers form a class of such grammars.

2 Why not use regular languages directly?

It is merely a convenient fiction that computer languages have context free grammars: any statically typed language has context-sensitive dependencies that establish correct type equivalence in expressions; and both static and dynamically typed languages must check that a function's call matches the signature of its definition. Fortunately (perhaps because of the limitations of our parsing technology) real languages limit these context dependencies to those that can be checked simply by noting the attributes of individual identifiers in a symbol table. We do not allow context sensitivities that require multi-word phrases to matched.

This kind of context sensitivity is a rather poor thing compared to the phenomena we see in natural languages, but many languages such as Pascal and C impose further restrictions on the ordering of type and signature dependencies by requiring identifiers to be declared before use, as opposed to just being declared within the program text. This arises from the observation that a declaration is more likely to reflect the programmer's intent than an instantiation; and that if the declaration is to be the 'gold-standard' then a single pass compiler must see it before any instances of the identifier.

So, context-free languages augmented with (possibly parse-time) type checks are good enough and we can design a notation that is sufficiently close to human language to be comfortable for programmers, whilst still being computationally tractable for compiler writers. Why not go further, and dispense with the complexities of context-free parsing, limiting ourselves to regular languages?

It would be perfectly possible to make a general programming language that had a regular syntax: we simply need to ensure that there are no nestable bracketing structures. The first casualty would be Pascal-style nested block structure with functions declared within functions, but this is already absent in ANSI-C and seems not to have been mourned by the software engineering community.

More serious losses would be nested control structures, and nested parenthesised expressions where the brackets are used to override the operator priorities. The workaround would be to pre-declare small function bodies containing the nested elements and to use their formal names as placeholders for the nested actions. This kind of flattening would in practice be a bridge too far: the separation within the program text of the components of an expression would require a great deal of cross referencing to be done whilst trying to understand a program.

A technical compromise would be to place an upper bound on the levels of nesting that may be used. We can write a regular grammar for any bracket nesting language with a finite maximum nesting level simply by enumerating all the possible nestings. The size of such grammars grows rapidly so a low upper bound might need to be imposed.

A much more attractive idea is to somehow separate out the regular parts of a grammar from the parts that are truly context free: i.e. those that include fully embedded recursion in which a recursive call has both non-empty left and right contexts. RI parsing is such a technology.

3 The background to Reduction Incorporated parsing

Reduction Incorporated parsing was introduced by Aycock and Horspool [3] with further development described in [4]. Their algorithm does not admit hidden left recursion. Our closely-related RIGLR algorithm allows completely general context free grammars to be used, and we also describe an alternative automaton construction process [7].

The essential idea in both algorithms is to construct a parsing automaton which performs reductions directly where ever possible. The goal is to increase the efficiency of general LR parsers by decreasing the amount of associated stack activity.

3.1 The Tomita and RNGLR algorithms

The standard LR parsing algorithm introduced by Knuth [10] constructs an LR DFA (usually an LR(1), SLR(1) or LALR DFA) and then uses a stack to traverse the DFA with a given input string (for full details see, for example, [1] or [2]). All types of LR DFA can be constructed for any context free grammar, but for all types of DFA there exist some grammars for which the corresponding push down automaton (PDA) is non-deterministic.

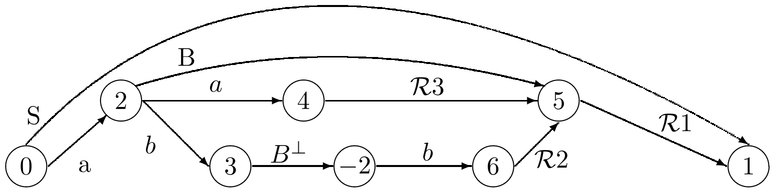
An obvious way to extend the standard LR parsing approach to incorporate non-determinism is to replicate the stack when a point of non-determinism is reached, and to explore all the possible traversals of the DFA. An efficient algorithm for exploring all traversals of a non-deterministic PDA which performs at most one stack pop and one stack push at each step, was given by Lang [11]. Tomita [15] gave an algorithm aimed explicitly at LR DFAs (which in their standard form can pop multiple stack symbols at each step). The core of Tomita's algorithm is the data structure known as a Graph Structured Stack (GSS) which represents the multiple stacks which can be generated. The importance of Tomita's algorithm is the efficient construction of the GSS.

Tomita's algorithm fails to terminate on certain grammars, but Farshi [12] has given a version which does terminate on all grammars. Farshi's algorithm is the recogniser at the heart of the ASF+SDF tool [16] and of Visser's work on 'scannerless' parsing [18]. However, Farshi's algorithm does not have the efficiency of GSS construction that Tomita's original algorithm employed. It turns out that by adding extra reduction items (equivalently extra pop actions) to the LR DFA it is possible to use Tomita's original algorithm correctly with any context free grammar, and furthermore it is possible to use a slightly more efficient algorithm. We call the new DFA's *right nulled* (RN) and the corresponding algorithm the RNGLR algorithm. Detailed comparisons of the RNGLR algorithm with Tomita's original algorithm and Farshi's algorithm, and with Earley's algorithm [6], can be found in [8].

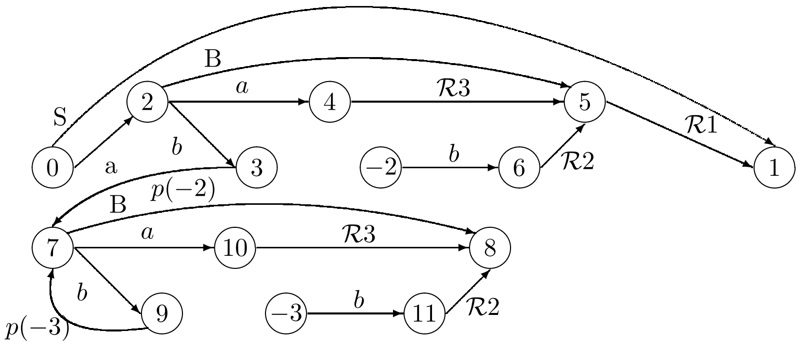
3.2 The Aycock and Horspool and RIGLR algorithms

Despite its relative efficiency, much of the work of the RNGLR algorithm is in the stack activity involved in constructing the GSS. It is well known that regular languages can be parsed without the need for a stack, and the Aycock and Horspool [3] algorithm uses a finite state automaton alone to parse the regular parts of a grammar and only uses a stack to deal with self embedding and right recursion. For both Aycock and Horspool's algorithm and our RIGLR algorithm the first step is to take the input grammar and to replace instances of non-terminals with pseudo-terminals, written $A^{\perp}$ where A is a non-terminal, until the resulting grammar has no self embedding. We call the result a *terminalised* grammar, and we call the particular set of instances of non-terminals which have been replaced with pseudo-terminals a *terminalisation* of the grammar. Once this is done a finite state automaton can be constructed which recognises precisely the language of the terminalised grammar. We call this automaton a *Recursion Incorporated Automaton* (RIA). The method of construction for the RIA is described in detail in [7], but for this paper it is sufficient to know that it has three types of edges: symbol edges labelled with terminals of the grammar; push edges labelled with pseudo-terminals and reduction edges labelled with grammar rules from the original grammar. For example, the following is the RIA for terminalised grammar

$$1.S ::= a\ B \qquad 2.B ::= b\ B^{\perp}\ b \qquad 3.B ::= a$$



For each pseudo-terminal $A^{\perp}$ we also construct the RIA, RIA(A), for the grammar obtained by taking as the start rule the rule for A. Then we replace each push edge labelled $A^{\perp}$ in all the RIAs with an edge to the start state of RIA(A). These new edges are labelled $p(k)$, where k labels the target of the corresponding push edge. This results in a push down automaton which we call a *recursive call automaton* (RCA) for the original grammar. For example, the following is an RCA for the above grammar.



As for LR DFAs, for some grammars the RCA will be non-deterministic. The RIGLR algorithm traverses any RCA with any input string and determines whether or not the string can be accepted by the RCA [7].

3.3 Trading time for space

One of the features of the RIGLR algorithm over other parsing algorithms is that it can be 'tuned' in a natural way to trade parse automaton size for runtime performance. In order to construct the underlying automaton all instances of self embedding in the grammar must first be detected and removed by introducing pseudo-terminals. Enough instances of pseudo-terminals must be introduced to create a grammar which has no self embedding, but additional terminalisations can also be introduced if desired. In general the more of these instances there are the smaller the size of the automaton but the more run-time stack activity there is.

4 Automatic computation of terminalisations

For the work reported in [8], the removal of self embedding from a grammar in order to generate the underlying RCA automaton was done by hand with some tool support. Our Grammar Tool Box (GTB) tool can construct a 'grammar dependency graph' which shows which non-terminals appear on the right hand side of the rule for a given non-terminal, and it can perform standard principal component analysis using Tarjan's algorithm [14] to detect the strongly connected components (SCC's), sub graphs in which every node can be reached from every other node. The result can be examined using the VCG graph visualisation tool [13], and from this instances of non-terminals to be replaced by pseudo-terminals can be chosen.

Grammar non-terminals can be thought of as 'calling' the non-terminals on the right hand side of their rules. These non-terminals then call the non-terminals on the right hands sides of their rules, and so on. If there is a path through the grammar that causes a non-terminal to call itself then we have recursion. We can abstract away from the grammar by combining the instances of non-terminals on the right hand sides of a grammar rule in a way that summaries a *dependency relation*; A depends on B if B appears on the right hand sides of the rule for A. It is helpful to display this relation as a directed graph, which we call the *Grammar Dependency Graph* (GDG). Then a non-terminal A is recursive if there is a non-empty path in the GDG from A to itself.

In our application we need to distinguish between left recursion, $A \overset{*}{\Rightarrow} A\gamma$, right recursion, $A \overset{*}{\Rightarrow} \gamma A$, and self embedding in which $A \overset{*}{\Rightarrow} \alpha A \beta$ where neither α nor β is ϵ, the empty string. To this end we label the edges of the GDG with the symbols L and R as follows. If the rule for A has an alternate $\mu B \nu$ in which $\mu \neq \epsilon$ then we label the GDG edge from A to B with L (B appears with a non-trivial left context). Correspondingly if $\nu \neq \epsilon$ then the edge is labelled with R. So edges are labelled with subsets of the set {L,R}.

It is easy to see that A displays self embedding if and only if there is a GDG path from A to itself in which at least one edge is labelled L and at least one edge is labelled R.

In order to remove recursion, we need to identify cycles in the GDG and then remove an edge, by terminalising the corresponding instance of the non-terminal which is the target of the edge. We now describe how GTB uses Tarjan's strongly connected component algorithm on the GDG to construct terminalisation sets for a grammar.

Application of Tarjan's strongly connected component algorithm

When Tarjan's algorithm is run on a graph it returns strongly connected components (SCC's) as sets of nodes. The sets are maximal with respect to the property that every node can be reached from every other node in the set. When run on a GDG the node sets represent maximal sets of non-terminals which are mutually dependent. If we consider the set of edges that have both their source and their destination within a particular SCC then we can interpret Tarjan's algorithm as returning all of the paths from a node to itself, which in general will include nested and intersecting families of cycles. In order to find a terminalisation we need the minimal cycles, that is SCC's that do not contain non-singleton SCC's, or equivalently those SCC's which have the same number of nodes and edges.

The approach we take is to recursively remove edges from SCC's until we are left with minimal cycles. This results in the construction of a family of SCC's which are nested by inclusion. If we represent this nesting as a directed acyclic graph (DAG), the SCC's we need, the mostly deeply nested ones, are the leaves of the DAG.

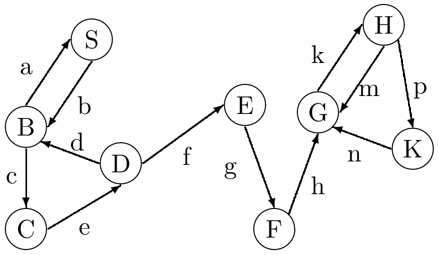
In detail, we run Tarjan's algorithm on the GDG and record each SCC generated. For each edge in each SCC we remove that edge and run Tarjan's algorithm on the resulting graph to generate a new family of SCC's which lie within the first SCC. This process is repeated until all the SCC's in the GDG have been found. The sets of nodes from each of the SCC's are presented as a DAG under inclusion. The leaf nodes of this DAG are disjoint sets, and any terminalisation of the grammar will have to include one edge between nodes in each of these sets. We construct all of the sets of edges which can be obtained by taking one edge from each leaf node set. Each of these sets is part of a terminalisation scheme for the grammar. (Note, in the limit this exhaustive search process is computationally infeasible for general graphs. However, we believe that the process is practical for graphs which are GDG's of real grammars. The process runs in a few seconds for the GDG's from our grammars for ANSI-C, Pascal and even Cobol.)

To get a full terminalisation set for the grammar we take the original GDG and remove from it the edges in the partial set, and run the whole process again on the reduced GDG. We continue in this way until there are no cycles left in the GDG.

Since all minimal cycles in the GDG must be removed to remove recursion, our procedure will find all minimal terminalisations of a grammar. Because the removal of an edge may later be superseded by the removal of another edge, the process may also return some terminalisations which are not minimal. But the minimal ones can be selected if required by running an inclusion test.

Example of automatic terminalisation

We illustrate this process using the following graph



which can be generated as the GDG for the grammar

$$S ::= B \quad B ::= S \mid C \mid aSS \quad C ::= D \quad D ::= B \mid E$$

$$E ::= F \quad F ::= G \quad G ::= H \quad H ::= G \mid K \quad K ::= G \mid \#$$

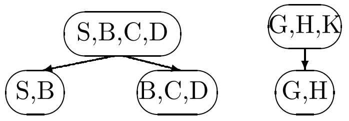
(Note the edges of the graph have been given names for reference in the text, and the L and R labels are not shown. In this case the only non-empty label is $\{L, R\}$ on the edge B to S.)

On the first run Tarjan's algorithm finds two non-empty SCC's whose edges are $\{a, b, c, d, e\}$ and $\{k, m, n, p\}$. We then remove each of the edges from each set in turn and run Tarjan's algorithm again.

Removing a from $\{a, b, c, d, e\}$ finds $\{c, d, e\}$ as does removing b. Removing each of c, d, e in turn finds $\{a, b\}$. Removing p or n from $\{k, m, n, p\}$ finds $\{k, m\}$ and removing m finds $\{k, n, p\}$.

Removing edges from the four new SCC's and then running Tarjan's algorithm does not generate any further non-empty SCC's, so the first step of the process is complete.

To find the minimal cycles we consider the DAG produced by taking the set of nodes from each of the SCC's and ordering them under inclusion. (Note, $\{k, m, n, p\}$ and $\{k, n, p\}$ have the same node set, but the leaf nodes correspond to unique SCC's.)



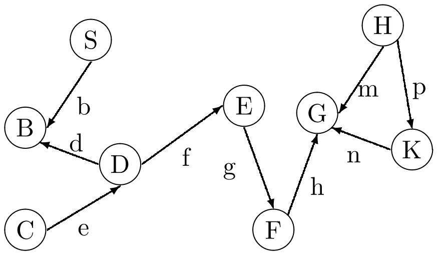
The next step of the algorithm is to form the partial terminalisation sets by removing one edge from each of the SCC's at the leaves of the DAG. This gives twelve sets

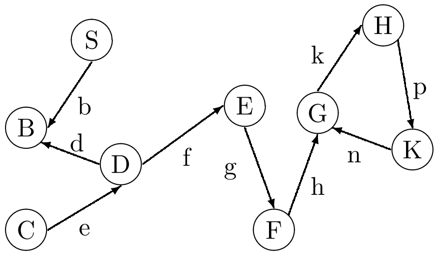
$$\{a,c,k\},\{a,c,m\},\{a,d,k\},\{a,d,m\},\{a,e,k\},\{a,e,m\},$$

$$\{b,c,k\},\{b,c,m\},\{b,d,k\},\{b,d,m\},\{b,e,k\},\{b,e,m\}$$

We then remove the edges from each set in turn from the GDG and run the whole process again.

Removing the first set above results in the graph of the left below, and removing the second set results in the graph on the right.





Running Tarjan's algorithm on the left graph finds no non-empty SCC's so we have a minimal terminalisation in which the edges a, c and k have been removed. Edge a represents instances of S on the right hand side of rules for B, and thus to remove this edge we need to terminalise all instances of S in all rules for B. Similarly, edge c represents instances of C in B and edge k instances of H in G. This generates the terminalised grammar

$$S ::= B \quad B ::= S^{\perp} \mid C^{\perp} \mid aS^{\perp}S^{\perp} \quad C ::= D \quad D ::= B \mid E$$

$$E ::= F \quad F ::= G \quad G ::= H^{\perp} \quad H ::= G \mid K \quad K ::= G \mid \#$$

Running Tarjan's algorithm on the right graph finds the SCC $\{k,n,p\}$. Removing each edge from this in turn and running Tarjan's algorithm generates nothing else, so we get single node DAG $\{7,8,9\}$. This requires us to remove one edge from a choice of three to add to the set we already have, giving the partial terminalisations

$$\{a,c,m,k\},\{a,c,m,n\},\{a,c,m,p\}$$

Removing these sets in turn and running Tarjan's algorithm shows that the graph has no further recursion, so these are terminalisation sets. (Notice the first set is non minimal.)

Carrying out this process on the other 10 partial terminalisations above ultimately results in 24 different terminalisations, 18 of which are minimal.

To complete this section we note that the above process removes all recursion from the grammar not just self embedding. To remove only instances of self embedding, each time Tarjan's algorithm is run, the SCC's produced are examined to see if they contain at least one edge labelled L and at least one edge labelled R. The SCC's which do not have this property corresponding to recursion which is not self embedding, so they are suppressed and treated as though they were empty SCC's.

5 Investigations

In [9] we compared the RNGLR and RIGLR algorithms for ANSI-C, Pascal and Cobol, showing that both techniques are practical for real grammars. However, there is very little true engineering experience of these techniques, particularly the RIGLR algorithm.

In the rest of this paper we present a variety of experiments that show features of RIGLR behaviour that motivate our automation project. The main message is that small changes to grammar terminalisation schemes can yield big reductions in the size of RI automata without significantly impacting parse times. These experiments give at best glimpses of the overall picture: a fuller characterisation of the space of RI automata will be possible when our automated tools are complete.

We shall first look at grammars for scannerless parsers for ANSI-C, COBOL and Pascal in which the normal lexical level rules have been directly incorporated. We shall then show some results concerning the breaking of long chains of rules in ANSI-C, and finally show some results from COBOL and ANSI-C that support our conjecture that parse-time profiled selection of terminalisations will be beneficial.

5.1 Character level, (scannerless) parsing

In principle we could define programming language grammars in terms of individual ASCII characters, but a traditional compiler usually comprises a lexical analyser that consumes *tokens* defined as regular sets over characters; and a parser which performs context free matching on the resulting token stream. This arrangement is attractive for several reasons: a regular lexer will usually be faster than a context free parser; segmenting the input stream into meaningful tokens can aid error reporting; and the terminal set of the parser can be large with respect to the underlying alphabet which can reduce the number of non-determinisms in the grammar. (Consider, for instance an LL(1) parser for Pascal which was attempting to work with individual characters: the keywords `do` and `downto` would generate a left-factoring conflict.) In addition, it is convenient to allow white space and comments to be quietly discarded by the lexer. A full character-level context free grammar for, say, C would have to make a call to a rule to match comments or white space after every keyword, which would cause considerable clutter.

Although this is conventional, there are several infelicities that arise. Probably best known is the so-called ANSI C *lexer hack*: a `typedef` statement in C defines a new type identifier which can subsequently be used as the first word of a statement. If the lexer has only a single token available for alphanumeric identifiers, then a one-token lookahead parser will be unable to distinguish between a declaration starting with an identifier that has been *typedef*'ed and a variable name that might be the start of an assignment statement. This is usually resolved by allowing the lexer to look in the compiler's symbol table for `typedef` identifiers, in which case a special token is returned. ANSI C also presents another oddity: real C programs are a mixture of two languages – the main C language and its pre-processor which uses a line-oriented syntax. As a result C compilers with integrated pre-processors need *two* scanners and must switch between them based on whether the first character of a line is a `#` character.

More serious problems arise in language prototyping environments such as ASF+SDF where parsers need to be constructed for mixed languages; or in production systems for mixed mode languages such as embedded assembler statements or mixed COBOL/SQL texts. A particular identifier may be a keyword in one language context and not in another, yet a traditional lexer cannot know which language context it is operating in. *Ad hoc* solutions involving parser to lexer feedback are required in these cases. A much cleaner solution is to simply specify the grammar right down to character level so that the full context free state is available as each character is consumed.

Once we incorporate the character level regular lexer rules into the context free grammar we have an excellent candidate for RIGLR parsing because the RI automaton construction will effectively 'recover' the regular lexer automata, so we might expect RIGLR to perform better than RNGLR on character level grammars. We shall examine RIGLR behaviour on grammars for ANSI-C, Pascal and Cobol to which we have added grammar rules which specify identifiers, integers and real numbers at character level. We have also written the keyword as strings of character tokens rather than as single tokens. These conversions were carried out automatically using our EBNF2BNF tool.

For the character level C grammar we have run the RIGLR and RNGLR algorithms on eight strings of varying lengths. In Table 1 we compare speed by noting the number of RCA pops performed by RIGLR and the number of GSS edge visits performed by RNGLR; and we compare the size of the parse time structures by showing the number of RIGLR call graph edges with the number of RNGLR GSS edges. It turns out that all four statistics grow essen-

string	12,207	16,202	18,546	21,032	23,330	24,631	26,858	27,748
RCA pops	3,734	5,348	5,993	6,885	7,648	8,285	9,107	9,527
GSS edge visits	15,397	20,803	23,625	26,885	29,668	31,537	34,402	35,599
Call graph edges	2,980	4,251	4,741	5,353	5,879	6,355	6,937	7,188
GSS edges	65,423	89,418	101,949	116,026	128,799	138,069	150,925	156,198

Table 1
Scannerless parsing of C programs

RCA1 pops	RCA2 pops	GSS edge visits	RCA1 edges	RCA2 edges	GSS edges
3,583	4,648	32,795	3,559	4,950	49,586

Table 2
Scannerless parsing of Cobol

tially linearly with the string length, as might be expected for an essentially deterministic grammar like ANSI-C. However, the RNGLR algorithm has to perform around four edge visits for each RIGLR pop action and the RNGLR GSS is between 21 and 22 times larger than the RIGLR call graph. RIGLR parsing appears preferable to Visser-style parsing for scannerless applications.

Now, it is possible that these effects arise from the highly deterministic nature of the ANSI-C grammar. A nondeterministic grammar for IBM VS-COBOL is available from `http://www.cs.vu.nl/grammars/vs-cobol-ii/`. The RCA obtained using the same minimal terminalisation as was used for the non-character level grammar is too large for the current version of GTB to construct. However, by adding three extra non-terminal instances to the terminalisation we can derive an RCA which has 9,736,820 edges. We refer to this as RCA1. If we further add all instances of the non-terminal `Cobword` to the terminalisation we get an RCA2 which has only 1,005,754 edges. The SLR(1) DFA for the grammar contains 144,584 edges and 466,428 reduction entries.

Table 2 shows the results of comparing the RIGLR algorithm running with RCA1 and RCA2 to the RNGLR algorithm using the SLR(1) DFA. We see the same general effect as for ANSI-C: RNGLR has to perform seven edge visits for each RCA2 pop and more than nine times as many visits as the closer-to-minimal RCA1. The size of the RNGLR GSS is more than ten times the size of the RCA2 call graph and nearly fourteen times the size of the RCA1 call graph.

As an aside, this table also shows that by adding the three extra terminalisations we increase the size of the call graph by 40%, but we have decreased the size of the automaton from nearly 10 million to around 1 million states: strong evidence for the kinds of useful performance trade-offs that we are seeking.

Identifier length	4	6	8	10
String length	16,434	19,180	21,926	24,672
GSS visits	40,213	42,959	45,705	48,451
GSS edges	118,744	126,982	135,220	143,458

Table 3
The effect of varying identifier size in Pascal RNGLR scannerless parsers

The impact of identifier length

The part of a character level grammar which specifies identifiers, and indeed the parts which specify numeric and string literals, are usually regular. Thus the stack activity associated with an RIGLR parse of a program is independent of the lengths of the identifier names. This is not the case for an RNGLR parse, because all of the symbols in the identifier name have to be pushed onto the stack. Intuitively, for scannerless parsing we might expect the size of an RNGLR GSS and the cost of its construction to increase linearly with average identifier length, all else being equal.

We illustrate this effect with a grammar for Pascal which is specified at the character level. We take a fixed Pascal program, and then change all of the identifier names so that they have the same length. We run both the RIGLR and RNGLR algorithms on the program with identifier lengths from 4 to 10. The table does indeed show the expected linear increase in cost for the RNGLR algorithm, while the RIGLR parser executes a constant 3,061 pops and builds a call graph with 2,357 edges for all of these cases.

To summarise: character level parsing is attractive for some applications tools but places great burdens on generalised parsers. The RIGLR algorithm, by 'recovering' the underlying regular parts of the grammar generates smaller run time structures which require commensurately less searching and is insensitive to the length of identifiers, numeric constants and string constants just as a traditional compiler with attached regular lexer would be.

5.2 Long chains in ANSI C

In [8] we noted that for each non-terminalised instance of a nonterminal B in the grammar generates a sub-automaton in the RCA of the size of RIA(B). Thus for a GDG dependency chain $B_0 \rightarrow B_1 \rightarrow \ldots \rightarrow B_n$ the number of copies of RIA(B_n) contributed to the RCA is $c_1 \times c_2 \times \ldots \times c_n$ where c_i is the number of instances of B_{i+1} in the rules for B_i. Thus by constructing a grammar with n nonterminals each of which has two instances of the next nonterminal in its rules and at least one terminal we can construct a grammar of size $O(n)$ which has an RIA of size at least $O(2^{n+2})$.

non-terminal	RCA nodes	call stack	pops
`conditional_expression`	1,499,973	2,615×3,083	3,336
`logical_or_expression`	1,504,750	2,631×4,121	3,384
`logical_and_expression`	770,018	2,638×3,131	3,405
`inclusive_or_expression`	406,954	2,655×3,152	3,152
`exclusive_or_expression`	234,026	2,656×3,154	3,410
`and_expression`	164,770	2,659×3,157	3,438
`equality_expression`	164,558	2,683×3,185	3,521
`relational_expression`	286,785	2,751×3,268	3,568
`shift_expression`	1,100,476	2,790×3,315	3,587
`additive_expression`	3,392,827	2,806×3,334	3,639

Table 4
The effect of chain breaking

We can reduce the size of the automaton by adding extra terminalisations within this chain. If we break a chain of length n in the middle, we convert an exponential in n to the sum of two exponentials in $n/2$. We expect, therefore, that if we try adding one extra terminalisation at all the positions in the chain then we shall see a steadily decreasing size towards the middle and then an increase.

In the GDG for our terminalised grammar for C there is a chain of length 16 in the expression part of the grammar. Table 4 shows this effect breaking this chain in several places, and using the resulting parsers to parse a string of 4,291 tokens. In each case the chain was broken by terminalising all instance of the stated non-terminal in the rule for the non-terminal preceding it in the chain. The non-terminals are listed in the table in order, `conditional_expression` is 3rd and `additive_expression` is 12th in the chain. We see that as predicted the smallest RCA is obtained by breaking the chain near the middle with the instances of `equality_expression`, and furthermore that the corresponding increase in stack activity on our given input string is small. The optimal break is not exactly in the middle because, whilst for most of the non-terminals in the above expression chain the number of right hand side instances which need to be terminalised is two, for `relational_expression` and `add_expression` the number is three, and for `shift_expression` the number is five. We can see that moving the terminalisation past these points rapidly reduces the size saving that is made. This demonstrates the additional fact, discussed in the next section, that GDG nodes with several children deep in a chain are one cause of very large RCAs.

non-terminal	keyword	RCA nodes×edges
`Copy_operand`	BY	5,030,351×5,349,721
`Unstring_statement_simple`	UNSTRING	4,509,150×5,216,413
`When_clause`	WHEN	3,885,498×4,125,327
`When_clauses`	EVALUATE	4,901,871×5,179,718
`When_phrase`	WHEN	4,764,322×5,025,832

Table 5
Some profile-style terminalisations

5.3 Deep rules with high fanout

From our observation of the steep increase in RCA size for deep rules with high fanout (i.e. with GDG nodes that have many children), we might wish to modify our 'break chains in the middle heuristic' in the presence of very high fanout nodes: it may be beneficial to directly terminalise instances of such rules, or their parents, effectively breaking chains immediately above high fanout rules.

COBOL presents useful examples. We have a (manually generated) minimal terminalisation for COBOL in which 28 instances of 20 non-terminals are terminalised, giving an RCA with 5,251,219 nodes and 5,582,158 edges. Within the COBOL GDG, the nonterminal `Statement` has 42 children and `Statement_non_closed` has 20 children. Both of these are good candidates for special treatment. Terminalising all instances of `Statement`'s parent node reduces the size of the RCA to 4,300,284 edges. A similar transformation applied with respect to `Statement_non_closed` yields an RCA with 4,421,808 edges. These kinds of transformations can be combined to great effect: if we terminalise immediately above both `Statement` and `Statement_non_closed` then the size of the RCA reduces to 2,537,668 edges, essentially cutting the size of the RCA in half from the minimal terminalisation.

5.4 In support of profiling

Our Cobol grammar contains non-terminals that derive only strings that begin with a particular keyword, and terminalising all instances of such a non-terminal will not generate any additional runtime stack activity for input which does not contain this keyword. Table 5 shows the effect of terminalising five nonterminals of this kind in COBOL: the size should be compared to the minimal terminalisation above which gives an RCA with 5,251,219 nodes and 5,582,158 edges.

This is, in a sense, a manual simulation of the kind of analysis that we expect our profiler to perform. COBOL lends itself to manual manipulation because keyword introduced statements are placed into separate rules. This makes it easy to terminalise just those parts of the grammar that relate to specific statements. In block structured languages like C and Pascal, the rules are far more intertwined, rendering manual analysis ineffective. Terminalisations of particular, rather than all, instances of a nonterminal are likely to be effective for C and Pascal and these could also be identified *via* profiling.

6 Conclusions and acknowledgements

We have shown that not all RI automata (and thus parse tables) are created equal, and that it is possible to find using *ad hoc* techniques automata which are small compared to the most parse-time-efficient automaton but which are not much slower on real inputs. We have also described the tools that we are constructing to allow automatic exploration of the space of RI automata and their characterisation in terms of parse-time rule profiles. We have some indications that exhaustive enumeration of a grammar's terminalisations may be computationally feasible.

It is reasonable to ask whether this level of pre-computation of automata will be practically worthwhile. We shall not rehearse here the arguments for generalised parsing in domain specific and prototyping language environments (see, for instance [17]). Even in applications dealing with the rather well behaved grammars that we have for current programming languages, generalised parsing speed is an issue. However, there are broader applications: in the field of bioinformatics searching, comparison and tagging of biological sequence data is almost invariably done with regular language recognisers even though it is well known that such sequences contain context free (and context sensitive) features. Tools such as GenLang [5] and work at the University of Washington represent the current best effort to apply context free searching, but further developments have been blocked by the unavailability of speed-competitive parsing technologies. We hope to develop techniques by which substantial automaton-generation time computation can be used to deliver sufficient improvements to parse times that context free searching of biological data becomes routine.

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; and to Georg Sander for his VCG graph visualisation software and for allowing it to be distributed with our toolkits.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles techniques and tools.* Addison-Wesley, 1986.

[2] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1 — Parsing of *Series in Automatic Computation*. Prentice-Hall, 1972.

[3] John Aycock and Nigel Horspool. Faster generalised LR parsing. In *Compiler Construction, 8th Intnl. Conf, CC'99*, volume 1575 of *Lecture Notes in Computer Science*, pages 32 – 46. Springer-Verlag, 1999.

[4] John Aycock, R Nigel Horspool, Jan Janousek, and Borivo Melichar. Even faster generalised LR parsing. *Acta Informatica*, 37(8):633–651, 2001.

[5] S. Dong and D. B. Searls. Gene structure prediction by linguistic methods. *Genomics*, 23:540:551, 1994.

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

[7] Adrian Johnstone and Elizabeth Scott. Generalised regular parsers. In Gorel Hedin, editor, *Compiler Construction, 12th Intnl. Conf, CC'03*, volume 2622 of *Lecture Notes in Computer Science*, pages 232–246. Springer-Verlag, Berlin, 2003.

[8] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. Generalised parsing: some costs. In Evelyn Duesterwald, editor, *Compiler Construction, 13th Intnl. Conf, CC'04*, volume 2985 of *Lecture Notes in Computer Science*, pages 89–103. Springer-Verlag, Berlin, 2004.

[9] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. The grammar tool box: a case study comparing GLR parsing algorithms. In Gorel Hedin and Eric Van Wick, editors, *Proc. 4th Workshop on Language Descriptions, Tools and Applications LDTA2004*, also in Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[10] Donald E Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[11] Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In *Automata, Lanugages and Programming: 2nd Colloquium*, volume 14 of *Lecture Notes in Computer Science*, pages 255 – 269. Springer-Verlag, 1974.

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

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

[14] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146:160, 1972.

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

[16] 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.

[17] M.G.J. van den Brand, C. Verhoef, and P. Klint. Re-engineering needs generic programming language technology. *ACM SIGPLAN notices*, 32(20):54:61, 1997.

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