Automatic recursion engineering of reduction incorporated parsers

Adrian Johnstone*, Elizabeth Scott

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

Received 1 December 2004; received in revised form 24 March 2006; accepted 12 April 2006
Available online 29 April 2007

Abstract

Reduction Incorporated (RI) parsers deliver high performance by suppressing the stack activity except for those rules that generate embedded recursion. Automaton constructions for RI parsing have been presented by Aycock and Horspool [J. Aycock, N. Horspool, Faster generalised LR parsing, in: Compiler Construction, 8th Intl. Conf, CC’99, in: Lecture Notes in Computer Science, vol. 1575, Springer-Verlag, 1999, pp. 32–46] and by Scott and Johnstone [A. Johnstone, E. Scott, Generalised regular parsers, in: G. Hedin (Ed.), Compiler Construction, 12th Intl. Conf, CC’03, in: Lecture Notes in Computer Science, vol. 2622, Springer-Verlag, Berlin, 2003, pp. 232–246] 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 by choosing different grammar terminalisation sets, 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. In this paper we describe automatic construction of minimal grammar terminalisation sets, giving examples drawn from ANSI-C, Cobol and Pascal; we describe the use of profiling to inform the choice of RI automaton; we investigate the use of RI parsers for scannerless parsing; and we explain some of the phenomena that influence the time/space trade-off for RI parsers.

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 original source grammar.

The basic idea is that where possible we effectively back-substitute rules and represent reductions as \( \epsilon \)-transitions: thus directly incorporating reductions 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 handle 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. A list of the points in a grammar where a call is made to a sub-automaton is referred to as a terminalisation. An
unexpected aspect of RI parsers is that, by choosing different terminalisations, 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 ‘nearby’ to these very large automata that have sizes commensurate with Knuth style automata but performance that is close to the best RI automaton.

A further interesting aspect of this approach is that since call graph size is a function of both the grammar and the string to be parsed, the process of 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. We shall show how both automaton size and parse-time performance can be significantly improved by this kind of parse-time profiling.

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

1. Automatic discovery of minimal terminalisations, with the potential for exhaustive enumeration of minimal terminalisations for moderately sized grammars.

2. Static characterisation of extensions to minimal terminalisations by measuring the size of the associated automata.

3. Dynamic characterisation of terminalisations by feeding back profiling data generated from instrumented versions of the RI recogniser which allow more parse-time-efficient automata to be selected.

Reduction Incorporated parsing was originally introduced by Aycock and Horspool [1] with further development described in [3]. Their algorithm does not admit hidden left recursion. Our closely related RIGLR algorithm does allow completely general context free grammars to be used [2,4]. We presented experimental results comparing the performance of the RIGLR algorithm to other general parsing algorithms in [5]. The terminalisations used in [1,4] and [5] were generated by hand. In order for the RIGLR algorithm to be used without the need to understand the underlying algorithm, we need to be able to automatically generate efficient terminalisations. It is informally claimed in [1] that the problem of generating minimum terminalisations for general grammars is NP-complete. In practice we believe grammars for real languages contain relatively limited true self-embedding (as opposed to left and right recursion) and thus will have some relatively small terminalisations. Our belief is borne out in our experimental investigation where we have been able to automatically generate several terminalisation sets for grammars for ANSI C, ISO Pascal and IBM VS-COBOL (see Section 4.3).

Manually finding terminalisations is time-consuming even if we are only looking for a single terminalisation that breaks all of the embedded recursion. Ideally we should like to automatically find a set of terminalisations, possibly containing all minimal ones, and then weight them according to profile statistics. This paper, then, is about exploring the space of RI automata for a given grammar. The main goal is to describe our approach to automatic generation and optimisation of RI automata for a given application. In addition, we give examples from standard programming language grammars of the ways in which small compromises in the amount of stack activity suppression can drastically reduce automaton size whilst not significantly affecting run time stack activity. We also present results concerning so-called scannerless parsing which is used in the SGLR parsing algorithms [6] in current versions of the ASF+SDF language prototyping environment [7]. 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 are completely specified by 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 [1]. Our closely related RIGLR algorithm allows general context free grammars to be used, and also employs an alternative automaton construction process [4]. 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 [8] 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, [9] or [10]). 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 [12] 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 [13] has given a version which does terminate on all grammars. Farshi’s algorithm is the recogniser at the heart of the ASF + SDF tool and of Visser’s work on ‘scannerless’ parsing [6]. 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 nullled (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 [14], can be found in [5].
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 [1] 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 a terminalisation of the grammar. A terminalisation is minimal if no proper subset of it is also a terminalisation.

Once a terminalised grammar has been generated, a finite state automaton can be constructed which recognises precisely the language of the terminalised grammar. We call this automaton a Reduction Incorporated Automaton (RIA). The method of construction for the RIA is described in detail in [4], 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

\[
\begin{align*}
1. S &::= a B \\
2. B &::= b B \perp b \\
3. B &::= a
\end{align*}
\]

For each pseudo-terminal \( A \perp \) we also construct the RIA, \( \text{RIA}(A) \), 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 \( \text{RIA}(A) \). 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 [4].

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 run time 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 experiments reported in [4], 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 [15] to
detect the strongly connected components (SCC’s), subgraphs in which every node can be reached from every other
node. The result can be examined using the VCG graph visualisation tool [16], and from this instances of non-terminals
to be replaced by pseudo-terminals can be chosen.

In [17] we described an earlier approach to automatic terminalisation in which we ran Tarjan’s algorithm recursively
on each of the graphs resulting from removing one edge from an SCC. In this paper we present a different, and much
more computationally attractive approach that directly locates cycles within each top-level SCC. Terminalisation
then proceeds by enumeration of all minimal terminalisation sets: we can use some of the structural properties of
terminalisation sets to limit the size of the search space; and if run times are still unacceptable we can exclude
terminalisation sets that are larger than some limiting cardinality. In this way we can find terminalisation sets for
real programming language grammars in a few seconds.

The grammars used in the discussion are the grammar for ISO-7185 Pascal extracted from the standard, the
grammar for ANSI-C extracted from [18] and a grammar for IBM VS-COBOL extracted by Steven Klusener and Ralf
Laemmel which is available from http://www.cs.vu.nl/grammars/vs-cobol-ii/. The algorithms have been implemented
in GTB which provides a unifying framework that contains implementations of many generalised parsing algorithms
along with a variety of grammar transformation functions and a scripting language for specifying experiments. A
paper detailing the use and functionality of GTB is in preparation. The timings here are for a Toshiba Satellite Pro
M10 containing a 1.6GHz Pentium-M (Centrino) and 512M bytes of memory. GTB was compiled using the Borland
5.01 C++ compiler environment with the Intel optimising C compiler running under Windows-XP.

4.1. Grammar dependency graphs

Grammars 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
and consider only the 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). A path of
length k in a graph is a sequence of nodes (N_1, . . . , N_{k+1}) such that there is an edge from N_i to N_{i+1}, for 1 ≤ i ≤ k,
and a cycle (from N_1 to itself) is a path (N_1, . . . , N_k, N_1), of length at least 1, such that N_i = N_j if and only if i = j.
Then a non-terminal A is recursive if and only if there is a cycle in the GDG from A to itself.

In our application we need to distinguish between left recursion, A →^* Ay, right recursion, A →^* γ A, and self-
embedding in which A →^* α Aβ 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 μ Bν in which μ ≠ ε then we label the
GDG edge from A to B with L (B appears with a non-trivial left context). Correspondingly if ν ≠ ε 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. We call such paths LR-paths. Similarly, we call a path that
contains at least one L (R) edge an L- (R-)path. (So every LR-path is also an L-path and an R-path.)

In order to remove recursion, we need to identify cycles in the GDG and then remove an edge from each cycle, by
terminalising the corresponding instance of the non-terminal which is the target of the edge.

4.2. Basic cycle breaking

When Tarjan’s algorithm is run on a graph it returns 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. Any LR-path
from a node to itself in a GDG is contained in a maximal SCC, and thus we begin by using Tarjan’s algorithm to find
the SCCs.

Each SCC that contains at least one edge labelled L and at least one edge labelled R is then considered. In any
graph there is a path from a node to itself if and only if the node belongs to a cycle. Thus to remove all recursion from a
grammar it is necessary and sufficient for one edge to be removed from each cycle in the GDG. However, because we
only want to remove self-embedding the situation is more complicated. It is possible for there to be an LR-path from a
node to itself that is comprised of two cycles, one containing no R edges and the other containing no L edges. Thus it
is not enough simply to remove the LR-cycles. On the other hand, removing edges from all of the L-cycles and all of
the R-cycles will, in some cases, result in non-minimal terminalisations. Removing an edge from every L-cycle will
result in a minimal terminalisation, but not all minimal terminalisations will be found. Clearly, in any terminalisation
all of the L-cycles or all of the R-cycles must have been removed, and once all of the L-cycles (or R-cycles) have been
removed there can be no remaining self-embedding. Thus to find precisely all the minimal terminalisations we run the
process twice, once to find all possible minimal terminalisations that can be obtained by removing edges from every
L-cycle and then again to find the minimal terminalisations by removing edges from every R-cycle. (Of course, the
two processes will find many of the same terminalisations.) In the rest of this discussion we shall describe the process
of generating terminalisations by removing L-cycles, the process for R-cycles is identical except that only R-cycles
are considered.

We illustrate the process using the following small grammar as a running example.

\[
\begin{align*}
S & ::= A \\
A & ::= E E B C \mid d \\
B & ::= A \mid E \\
C & ::= a D \mid c \\
D & ::= C \mid D a \\
E & ::= A S \mid F \mid e \\
F & ::= a F \mid E b
\end{align*}
\]

The grammar has GDG

First we run Tarjan’s algorithm. Since we are removing L-cycles, all L-loops (L-edges from a node to itself) must
be included in any terminalisation and no other loops are of interest. To reduce the size of the SCC’s we remove all
loops. For the above example this gives the following two LR-SCC’s (whose edges have been numbered for later
reference and whose L, R labels have been removed as they are only needed to construct the LR-SCC’s).

We then consider each LR-SCC in turn.

We begin by finding all the cycles, and then we consider all the L-cycles. For the first SCC in our example this
results in five L-cycles.
We could form terminalisation sets by simply taking one edge from each cycle. However, in general such a set would not be minimal. To find all the minimal sets we would need to construct all such sets and then test them for mutual inclusion. Of course the number of such sets is the product of the sizes of each of the cycles, a very large number for real grammars. There are graphs, and corresponding grammars, for which the edge sets of all the cycles are disjoint. In this case the number of minimal terminalisations is the product of the sizes of the cycles. However, it is our belief that for grammars for real languages there is considerable overlap between the cycles and our approach is designed to exploit this to improve the average case performance of the algorithm.

The basic version of our algorithm lists the cycles in some order and then works recursively down this list selecting one edge from the next cycle, unless an edge from that cycle has already been chosen.

In our example, suppose that we choose edge 1 from the first cycle. Since 1 is not in the second cycle we choose another edge, 3 say. Since 1 is in the third cycle we pass over it. Then we choose an edge, 2 say, from the fourth cycle. Edge 2 is in the fifth cycle, so this exploration path is complete and has produced the terminalisation

\{1, 2, 3\}.

The algorithm now recurses out to re-consider the fourth cycle, and this time we choose edge 5. The currently constructed set is \{1, 3, 5\}, which does not contain an edge from the fifth cycle. Thus an edge, 2 say, is selected resulting in the set \{1, 2, 3, 5\}. The algorithm then re-consider the fifth cycle and selects the other possible edge, resulting the terminalisation set \{1, 3, 4, 5\}.

The algorithm carries on in this way, exploring all choices until all the edges from the first cycle have been considered. The algorithm then terminates, having constructed a list of terminalisation sets. This list will include all the minimal terminalisations, but it may also include some non-minimal sets, because edges added later may make edges selected earlier redundant. Thus as a final step the algorithm compares the sets for inclusion.

The actual list of terminalisations constructed before the test for subsets will depend on the order in which the cycles are visited. In our example, selecting the edges in numerical order, the algorithm generates the following set of terminalisations before testing for subsets. (Duplicate sets are not added to the list.)

\[
\begin{align*}
\{1, 2, 3\} & \quad \{1, 4, 6\} & \quad \{1, 2, 5, 6\} & \quad \{2, 3, 7\} \\
\{1, 2, 3, 5\} & \quad \{2, 3\} & \quad \{1, 4, 5, 6\} & \quad \{2, 3, 5, 7\} \\
\{1, 3, 4, 5\} & \quad \{2, 3, 6\} & \quad \{2, 3, 5, 6\} & \quad \{3, 4, 5, 7\} \\
\{1, 2, 3, 6\} & \quad \{2, 6, 7\} & \quad \{3, 4, 5, 6\} & \quad \{2, 3, 6, 7\} \\
\{1, 3, 4, 6\} & \quad \{2, 3, 5\} & \quad \{2, 5, 6, 7\} & \quad \{3, 4, 6, 7\} \\
\{1, 2, 6\} & \quad \{3, 4, 5\} & \quad \{4, 5, 6, 7\} & \quad \{4, 6, 7\}
\end{align*}
\]

After subset comparison, we are left with six terminalisations. Each of these must have the L-loop from \(F\) to itself added. If we number this edge 12 we have the following (L-generated) terminalisations for the first SCC in our example.

\[
\{2, 3, 12\} \quad \{1, 2, 6, 12\} \quad \{1, 4, 6, 12\} \quad \{2, 6, 7, 12\} \quad \{3, 4, 5, 12\} \quad \{4, 6, 7, 12\}
\]

We then repeat the process for the R-cycles. Once the terminalisations for R-cycles are constructed we run a final test to see whether any of the \(L\)-terminalisations are subsets of the \(R\)-terminalisations, or vice versa. Then the resulting sets are the minimal terminalisations for the SCC.

In our example the \(R\)-cycles are the five \(L\)-cycles above and the cycle \{8, 9\}. As this last cycle is disjoint from the others, the terminalisations constructed will be those constructed as above with the addition of either 8 or 9. Thus we
get 48 interim terminalisation sets and the following twelve final sets
\[ \{2, 3, 8\} \{1, 2, 6, 8\} \{1, 4, 6, 8\} \{2, 6, 7, 8\} \{3, 4, 5, 8\} \{4, 6, 7, 8\} \]
\[ \{2, 3, 9\} \{1, 2, 6, 9\} \{1, 4, 6, 9\} \{2, 6, 7, 9\} \{3, 4, 5, 9\} \{4, 6, 7, 9\} \]

As none of the L-generated sets are subsets or supersets of the R-generated ones, the eighteen terminalisations are the minimal ones for this SCC.

We then compute the terminalisation sets for the other SCC's, and combine them to form terminalisation sets for the whole grammar.

In our example the other SCC has three minimal singleton terminalisations
\[ \{10\} \{11\} \{13\} \]
(Where 13 is the edge from D to itself.) This results in a total of 54 minimal terminalisation sets for the grammar.

Whilst this basic approach may seem somewhat cumbersome it works reasonably well in practice. For example, our grammar for Pascal has five LR-SCCs and these SCCs have 660, 1026, 18, 117 and 12 minimal terminalisation sets respectively; and using only this basic approach the tool found all these in about 60 user CPU seconds (CPUs).

4.3. Pruning the search space—limiting set size

The problem with the basic approach above is that the number of interim terminalisations sets constructed prior to the minimality testing can become very large. For Pascal there were 241408, 21654, 84, 705, and 42 sets respectively for the five SCCs before minimisation pruning, relatively small numbers well within the scope of our system. However, for our grammar for C the corresponding numbers are much larger. The GDG for our C grammar has three SCC's, one of which has over 8,000,000 interim terminalisation sets. Furthermore, for some grammars the number of cycles, and the final number of minimal terminalisation sets, means that the prospect of finding all minimal terminalisation sets is impractical.

As discussed in Section 4, our first goal is simply to construct at least one minimal terminalisation set for a given grammar, and terminalisation sets with fewer elements are likely to be better than ones with more elements since stack activity is associated with and only with terminalisation instances. This suggests that we modify the basic algorithm so that it only constructs terminalisation sets of size less than some given integer, \(N\) say, for each SCC. This is achieved by simply terminating the construction of a set when the \(N + 1\)st element is added. As a result precisely the minimal terminalisation sets of size at most \(N\) will have been constructed.

We call a terminalisation set minimum if it has the smallest possible number of elements. One affect of this modification is that when the algorithm terminates, if any terminalisation sets have been found then all the minimum ones will have been found. We have used this modified approach on our C, Cobol and Pascal grammars.

Our C grammar has three LR-SCCs. Running the algorithm with \(N = 1\) we find two minimum terminalisation sets for the first SCC. (In fact these are the only minimal terminalisations for this SCC.) To find terminalisation sets for the second SCC we have to set \(N = 4\), at which point the algorithm finds 36 minimum terminalisations. Finally, for the third SCC (which contains 75 cycles) we have to set \(N = 12\), at which point 14 minimum terminalisation sets are found.

Thus we have found the \(2 \times 36 \times 14\) minimum terminalisation sets for our C grammar. Each of these sets contains 17 elements and the algorithm, with \(N = 12\), ran in 0.3CPUs.

Our Cobol grammar has four LR-SCCs. Running the algorithm with \(N = 1\) we find three minimum terminalisation sets for the first SCC. Running the algorithm with \(N = 2\) we find ten minimum terminalisation sets for the second SCC. Running the algorithm with \(N = 3\) we find six minimum terminalisation sets for the third SCC. Finally for the fourth SCC (which contains 254 cycles) we have to set \(N = 5\), at which point 2 minimum terminalisation sets are found.

Thus we have found the \(3 \times 10 \times 6 \times 2\) minimum terminalisation sets for our Cobol grammar. Each of these sets contains 11 elements and the algorithm, with \(N = 5\), ran in 3.8CPUs.

For Pascal, as we have seen above, we can quickly find all the minimal terminalisation sets, but we have used the limit functionality to find the ones of minimum size. There are \(4 \times 2 \times 18 \times 9 \times 2\) minimum sets, and they each contain nine elements.
As we have said, the terminalisations used for the experiments reported in [4] were calculated by hand. For Pascal the grammar had 21 terminalisation instances and for C the grammar had 46 instances. The number of terminalisation instances can be higher than the size of the corresponding terminalisation set because a grammar rule may contain more than one instance of a given non-terminal on its right hand side. To remove the recursion, all instances in a particular rule must be terminalised. For our Pascal grammar there was only one instance of each non-terminal to be terminalised in any one rule, so 21 is also the size of the terminalisation set, illustrating that it is not easy to find minimum terminalisations by hand. For C, the smallest instance weighted value of our minimum terminalisations sets is 26, but the hand terminalised grammar for C used in [4] was deliberately non-minimal as part of a time/space trade-off to reduce the automaton size.

4.4. Profiling for terminalisation selection

At first glance, it might seem that simply selecting the smallest terminalisation set that breaks all self-embedding is all that is required to build an RIGLR parser. After all, the aim is to reduce stack activity and fewer non-terminal terminalisations provide fewer opportunities for stack calls. However, the relationship between terminalisation set size and stack activity is not straightforward. One of the most interesting aspects of RIGLR parsers is that there are usually multiple terminalisation sets which are sufficient (in the sense of clearing all self-embedding) but which display different time/space tradeoffs at parse time on particular classes of input string. It turns out that selecting terminalisation sets larger than one of the minimal sets may yield better performance. More terminalisations in a little-used part of the grammar may be preferable to fewer terminalisations in other parts.

We distinguish between static properties of the parser, which are a function of the input grammar only, and dynamic properties which are a function of both the input grammar and the string to be parsed. Using a procedure described in Section 4.4.1 we shall compute the probability of a particular instance of a non-terminal generating stack activity in an RIGLR parser; thus generating dynamic profile weights for our sets. We contrast this with a static equal probability weighting that is computed by assuming that all non-terminal stack activities have equal probability.

The number of grammar instances corresponding to each element of a terminalisation set is a property of the grammar. Thus we can think of the GDG as being a weighted graph, with the weight on the edge, from A to B say, being the number of instances of B on the right hand side of the grammar rule for A. We can then weight terminalisation sets, which are sets of GDG edges, by adding together the weights on the elements. A low weight set is likely to yield a more efficient parser than a high weight one. These weights assume that every instance of a non-terminal is equally likely to be activated during a parse, although in practice we shall recommend using profiling information to provide a more realistic weighting. We refer to these two cases as the equal probability and profiled weightings.

As we have already mentioned, our C grammar has three LR-SCC’s. SCC 1 has two minimum terminalisation sets of size one, SCC 2 has 36 minimum sets of size four, and SCC 3 has 14 minimum sets of size 12. Figs. 1 and 2 show the terminalisation sets for SCC 2 and SCC 3 respectively plotted according to their equal-probability weights and their profiled weight derived using the procedure described below.
There is a wide range, from 4 to 19, in equal probability weighted size amongst the minimum terminalisations for SCC 2. For SCC 3, the equal probability weights are all about the same: there are 7 sets with equal probability weight 20; 5 sets with weight 21 and 2 sets with weight 22. It is not surprising that it is hard to find the most efficient terminalisation by hand.

4.4.1. The profiling procedure

Using equal probability weight values to choose between minimum terminalisations does not address the first point we mentioned above, that it may be preferable to choose more terminalisations in seldom used parts of a grammar. One way to gather information on the potential stack activity associated with particular terminalisations is to build up data on how often instances of non-terminals are encountered when strings are parsed. To do this we can profile the parser on sample input programs.

The structure of RIGLR parsers makes it easy to gather this type of profiling information. It is, of course, possible to construct an RIGLR parser from a grammar by terminalising every instance of every non-terminal. We call such a parser a maximally terminalised RIGLR parser. Such a parser generates maximum stack activity, because every instance of every non-terminal triggers a call to the automaton for that non-terminal, but these calls allow us to count the number of times instances are encountered. The method is not perfect because occasionally a push edge in the automaton is shared by more than one corresponding instance of a terminalised non-terminal. (This can happen only when both instances occur in the same place in the same ‘viable prefix’ and as we are only dealing with probabilities anyway we can accept this small inaccuracy. In fact the situation does not occur at all in the automaton for our C grammar.)

Parsing input strings with a maximally terminalised grammar generates a frequency score for each instance of each non-terminal. For each terminalisation set constructed by the process described above we can add together the frequency scores for each element of the set, generating a profiled weight for the set. The set with the lowest score is likely to produce the parser with lowest average stack activity.

4.4.2. Some profiling results

As proof of concept, we have built the maximally terminalised RIGLR parser for our C grammar and run it on 3 input strings of lengths 4291, 26551 and 36,827 tokens. The total length of the original source code is of the order of 12,000 lines. Most of the ANSI-C code is the source code for our RDP [19] and GTB tools. We have calculated the profiled weights based on this data for each of the minimum terminalisation sets for our C grammar. The example code has all been written by the same person so it cannot be taken as representative of typical usage, but the experiment illustrates the approach.

For ANSI-C, SCC 1 has only two sets. The first set has a profiled weight of 7 and the second of 17. The results for the other two SCC’s are displayed graphically in Figs. 1 and 2 in which the x-axis of each graph is labelled with the equal-probability weights of the sets; and the y-axis is labelled with the profiled weight. For SCC 2 these scores range from 2,774 to 18,522; and for SCC 3 the scores range from 21,583 to 78,053.

As we have already said, the code we used for profiling was all written by the same person and thus this data is only useful for illustration purposes. However, it is interesting to note that for both SCC 2 and SCC 3 the smallest equal...
probability minimum terminalisations are not those that have the lowest profiled weights. In fact for SCC 3 the second lowest profiled weight belongs to one of the sets with the highest equal probability weight. This shows that, at least for one programmer, it is not always the case that the lowest weighted terminalisation set will generate the most efficient parser. Clearly some engineering experience is needed before we can determine in general which terminalisation sets to choose.

It is clear, though, that not all of these effects need be artefacts of the particular source code sample. Consider the following GDG in which the edge from R to S is both part of the cycles R–S–T–X and R–S–W as well as being part of the non-recursive paths R–S–U… and R–S–V…

If we are only concerned to minimise the number of terminalised edges in the GDG then clearly the edge R–S should be terminalised since it breaks both loops. Doing so, however, associates stack activity with the non-recursive paths too, and thus it would be more efficient to termnalise R–W and either X–R or T–X. We would expect profiling to discover these relationships automatically, without any need to examine the structure of the GDG.

It turns out that SCC 3 in the ANSI-C grammar displays this behaviour: it corresponds to that part of the grammar which describes statements. Some statements nest (such as if and for) corresponding to the recursive paths, and some (such as assignment, break and continue) do not. Whether to terminalise the ‘spine’ of the loops (edge R–S above) or the back edges such as W–R can be decided on the relative frequency of nested instances of the nestable statements compared to that of un nestable statements and non-nested instances of nestable statements. We know from Knuth [20] (and see also [21]) that real programs rarely show deep nesting of control structures, so it is not surprising that terminalising back edges rather than the spine is a good strategy in SCC 3. Programs written by programs, such as ANSI-C output by compilers that use C as an intermediate language, or parsers written by our own RDP parser generator [19] may display deep nesting.

In this section we have discussed the construction and potential variation in efficiency of minimal terminalisation sets. In the following sections we describe some investigations into specific RIGLR parsers. We consider examples of so-called scannerless parsers for which RIGLR parsers present potentially significant advantages. We also consider the trade-off between parser size and efficiency, showing that we may not always wish to choose the most efficient RIGLR parser even when we can find one.

5. The application of RIGLR parsers to 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 ASF + SDF uses a special rule to handle whitespace rather than requiring explicit handling in the source grammar).

Although this is conventional, there are several difficulties 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. 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 essentially 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. The IBM VS-COBOL grammar is highly non-deterministic. 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 on a single string. 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.

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

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

<table>
<thead>
<tr>
<th>RCA1 pops</th>
<th>RCA2 pops</th>
<th>GSS edge visits</th>
<th>RCA1 edges</th>
<th>RCA2 edges</th>
<th>GSS edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,583</td>
<td>4,648</td>
<td>32,795</td>
<td>3,559</td>
<td>4,950</td>
<td>49,586</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Identifier length</th>
<th>String length</th>
<th>GSS visits</th>
<th>GSS edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>16,434</td>
<td>40,213</td>
<td>118,744</td>
</tr>
<tr>
<td>6</td>
<td>19,180</td>
<td>42,959</td>
<td>126,982</td>
</tr>
<tr>
<td>8</td>
<td>21,926</td>
<td>45,705</td>
<td>135,220</td>
</tr>
<tr>
<td>10</td>
<td>24,672</td>
<td>48,451</td>
<td>143,458</td>
</tr>
</tbody>
</table>

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.

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. Table 3 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 an attached regular lexer would be.

The experimental observations described here mostly concentrate on asymptotic performance of the algorithms in terms of the size of the run time structures and the number of stack operations. We have, at this stage, not characterised the constants of proportionality in our algorithms. For both algorithms, the costs of stack operations and structure searching is similar, but there are alternative implementations of the run time set manager for both the RIGLR and RNGLR algorithms that merit careful study; and these alternatives allow significant time/space tradeoffs of their own. We shall report in a later paper on some tightly engineered versions of the algorithms. All of the experiments here executed in at most a few seconds on our prototype implementations. There are additional opportunities for time/space tradeoff in the RIGLR automata which we illustrate in the next section.

6. Some factors influencing automaton size

In this section we present some heuristic insights into the structure of terminalisation sets as reflected in the size of the resulting automata. It is possible for terminalisations of similar parse-time performance to vary significantly in size.

6.1. Long chains in ANSI C

In [5] we noted that for each non-terminalised instance of a non-terminal $B$ in the grammar generates a sub-automaton in the RCA of the size of RIA($B$). Thus for a GDG dependency chain $B_0 \rightarrow B_1 \rightarrow \cdots \rightarrow B_n$ the number
Table 4

The effect of chain breaking

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>RCA nodes</th>
<th>Call stack Nodes×Edges</th>
<th>Pops</th>
</tr>
</thead>
<tbody>
<tr>
<td>conditional_expression</td>
<td>1,499,973</td>
<td>2.615×3,083</td>
<td>3,336</td>
</tr>
<tr>
<td>logical_or_expression</td>
<td>1,504,750</td>
<td>2.631×4,121</td>
<td>3,384</td>
</tr>
<tr>
<td>logical_and_expression</td>
<td>770,018</td>
<td>2.638×3,131</td>
<td>3,405</td>
</tr>
<tr>
<td>inclusive_or_expression</td>
<td>406,954</td>
<td>2.655×3,152</td>
<td>3,152</td>
</tr>
<tr>
<td>exclusive_or_expression</td>
<td>234,026</td>
<td>2.656×3,154</td>
<td>3,410</td>
</tr>
<tr>
<td>and_expression</td>
<td>164,770</td>
<td>2.659×3,157</td>
<td>3,438</td>
</tr>
<tr>
<td>equality_expression</td>
<td>164,558</td>
<td>2.683×3,185</td>
<td>3,521</td>
</tr>
<tr>
<td>relational_expression</td>
<td>286,785</td>
<td>2.751×3,268</td>
<td>3,568</td>
</tr>
<tr>
<td>shift_expression</td>
<td>1,100,476</td>
<td>2.790×3,315</td>
<td>3,587</td>
</tr>
<tr>
<td>additive_expression</td>
<td>3,392,827</td>
<td>2.806×3,334</td>
<td>3,639</td>
</tr>
</tbody>
</table>

of copies of $\text{RIA}(B_n)$ contributed to the RCA is $c_1 \times c_2 \times \cdots \times c_n$ where $c_i$ is the number of instances of $B_{i+1}$ in the rules for $B_i$. Thus by constructing a grammar with $n$ non-terminals each of which has two instances of the next non-terminal 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 $O(2^{n+2})$.

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 the (manually generated) terminalised grammar for C used for the experiments reported in [5], there is a chain of length 16 in the expression part of the grammar. Table 4 shows the effect of 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 instances 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.

6.2. 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 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 non-terminal 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.
Table 5
Some profile-style terminalisations

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Keyword</th>
<th>RCA nodes × edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy_operand</td>
<td>BY</td>
<td>5,030,351 × 5,349,721</td>
</tr>
<tr>
<td>Unstring_statement_simple</td>
<td>UNSTRING</td>
<td>4,509,150 × 5,216,413</td>
</tr>
<tr>
<td>When_clause</td>
<td>WHEN</td>
<td>3,885,498 × 4,125,327</td>
</tr>
<tr>
<td>When_clauses</td>
<td>EVALUATE</td>
<td>4,901,871 × 5,179,718</td>
</tr>
<tr>
<td>When_phrase</td>
<td>WHEN</td>
<td>4,764,322 × 5,025,832</td>
</tr>
</tbody>
</table>

6.3. In support of profiling for automaton size optimisation

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 run time stack activity for input which does not contain this keyword. Table 5 shows the effect of terminalising five non-terminals 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 our profiler could also be used 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 non-terminal are likely to be effective for C and Pascal and these could also be identified via profiling.

7. 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.

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 [22]). 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 (see, for instance, chapters 9 and 10 of [23]). Tools such as GenLang [24] 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 re-engineering; and 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


A. Johnstone, E. Scott, rdp—an iterator based recursive descent parser generator with tree promotion operators, SIGPLAN notices 33 (9).


