Generalised Parsing: Some Costs

Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos

Royal Holloway, University of London
A.Johnstone@rhul.ac.uk or E.Scott@rhul.ac.uk

Abstract. We discuss generalisations of bottom up parsing, emphasising the relative costs for real programming languages. Our goal is to provide a roadmap of the available approaches in terms of their space and time performance for programming language applications, focusing mainly on GLR style algorithms. It is well known that the original Tomita GLR algorithm fails to terminate on hidden left recursion: here we analyse two approaches to correct GLR parsing (i) the modification due to Farshi that is incorporated into Visser's work and (ii) our own right-nullable GLR (RNGLR) algorithm, showing that Farshi's approach can be expensive. We also present results from our new Binary RNGLR algorithm which is asymptotically the fastest parser in this family and show that the recently reported reduction incorporated parsers can require automata that are too large to be practical on current machines.

1 Introduction

Generalisations of deterministic parsers are increasingly being applied to programming language processors: several implementations of Eiffel use Earley parsers; the most widely used recursive descent parser generators (PCCTS [1] and its successor ANTLR) use an approximate LL(k) algorithm extended with lookahead predicates; and Bison has recently acquired a GLR mode [2]. Many less widely used parser generators exist that provide various methods of backtracking, lookahead and breadth first search.

These developments might have been surprising to language designers from forty years ago: a major result from the work on deterministic top down and bottom up parsing algorithms was that near-deterministic languages are sufficiently expressive to give the human programmer a comfortable notation that, even on the very small machines of the day, allowed thousands of tokens to be parsed per second. Since then, it has been conventional to use LALR or LL techniques to implement our synthetic languages, supplemented with simple disambiguation strategies to cope with, for instance, the IF-THEN-ELSE ambiguity. The appearance of more powerful parser generators is in part a reflection of the enormous gains in processor performance and (perhaps more importantly) memory availability since the development of LALR and LL parser generators in the early 1970's when both memory sizes and instruction rates were typically a small multiple of 10^4. Commodity computers now sport physical memory capacities and instruction rates in the 10^9 range.

E. Duesterwald (Ed.): CC 2004, LNCS 2985, pp. 89–103, 2004.

Of course, when the shape of the language is not under a designer's control, we cannot assume the existence of a deterministic grammar. Natural language parsing (NLP) has to contend with nondeterministic languages that are usually ambiguous as well: a parser must thus incorporate mechanisms for finding particular derivations from a potentially infinite set. Generalised parsing has therefore always been central to NLP applications and developments of Earley style chart parsers and Tomita style generalised LR (GLR) parsers have continued in the NLP field whilst computer science based research in parsing has declined. Other applications include biological sequence analysis [3] (where context free searches can deliver fewer false positives than those specified with regular expressions where nested matching components occur) and, closer to home, algebraic specification of language processors for reverse engineering [4]. It is sometimes hard to predict how NLP style parsers will cope with programming languages. The distinguishing characteristics of NLP systems from a parsing point of view are that the grammars are large (six or seven thousand productions is not unusual) and the input sentences are usually only at most a few tens of symbols long. In contrast, the ANSI-C grammar (after expansion of optional phrases) has only 71 nonterminals and 230 productions, but source files containing thousands of tokens are normal.

This paper is about generalisations of bottom up parsing; their size and performance relative to other kinds of general parsers. We shall identify a variety of practical limitations and discuss the practical performance in terms of (i) the *size* of a parsing program and its associated tables; (ii) the run time performance; and (iii) the run time space requirements. We compare performance with Earley's parser. We have discussed elsewhere the limitations of arbitrary lookahead top down parsers [5]. Here our goal is to provide a roadmap of the available GLR approaches in terms of their space and time performance for small-grammar/long-input problems.

It is well known that the original Tomita GLR algorithm fails to terminate on hidden left recursion. We analyse two approaches to correct GLR parsing: the modification due to Farshi [6] that is incorporated into Rekers' [7] and Visser's [8] work; and our right-nullable GLR (RNGLR) algorithm [9,10]. We also present results from our new Binary RNGLR algorithm [11] which is asymptotically the fastest parser in this family. In addition, we present results on the size of parsers generated using recently reported reduction incorporated automata [12,13] which indicate that the commonly encountered grammar constructs can yield parsers that are too large to be practical on current machines.

The paper is organised as follows. For completeness, we begin by noting some of the limitations of backtracking parsers. In section 3 we distinguish between recognisers and parsers, which must return a (potentially infinite) set of derivations, and consider the size of the recognisers. Sections 4 and 5 summarise the historical development of some recognition matrix-based algorithms and GLR algorithms, respectively. Experimental results for a variety of techniques are given in section 6. Section 7 describes an algorithm whose underlying automaton contains 'reduction' edges and gives experimental results which show that

such automata are probably impractically large. We demonstrate some simple transformations that can help ameliorate this problem in some cases.

2 Backtracking and Lookahead

Many authors have described parsing algorithms in which deterministic approaches are enhanced with backtracking and lookahead. Such parsers generally have exponential performance bounds in their naïve form. Algorithms augmented with a well-formed substring table (actually, the CYK recognition matrix described below) can avoid re-computing substring matches, in which case $O(n^3)$ time bounds can be achieved although we know of no production backtracking parser generator that uses this strategy.

BtYACC (Backtracking YACC) [14] is a bottom up backtracking parser: Berkeley YACC was modified to perform exhaustive backtracking when it comes across a conflict in the table by first shifting and then backing up and making all possible reductions until it finds a successful parse or until it has tried every possibility. Using this technique it is relatively simple to produce quadratic and exponential parse-time behaviours. The general principle is to force the parser to shift past many potential reduction steps whilst pursuing fruitless paths within the grammar, thus maximising the wasted backtracking steps. Consider the (ambiguous) grammar

$$\begin{array}{lcl} S & ::= & Ab \mid A'c \\ A' & ::= & A'a \mid a \\ A & ::= & aA \mid Aa \mid a \end{array}$$

Strings in the b-sublanguage are parsed in linear time, but exponential time is required to parse strings in the c-sublanguage. Using string lengths n from 10 to 20 tokens on a 120MHz Pentium, we found that the run times measured as above are a good fit to $6.75 \times 10^{-6}\, 2^n$ which translates to run times of over 7 seconds for strings of only 20 characters (and a little over 240 years for strings of 50 tokens).

For completeness we note that several public domain top down backtracking parser generators utilise a longest match strategy, returning a single match from each parser function. This approach requires a grammar to be ordered so that the longest match is encountered first by the parser. Sadly, there exist languages that are incompatible with ordered longest match in that the rules must be ordered one way for some strings and another way for others. Of course, even correct recursive descent backtracking parsers cannot cope with left recursion since the recursion will never bottom out.

3 Parsers and Recogniser Size

In the rest of this paper we focus on algorithms which attempt to explore all possible derivations in a breadth-first manner. In this section we look briefly at the additional issues surrounding derivation output and then consider the size of the recognisers themselves.

Deterministic LL and LR parsers can easily provide a derivation as a side effect of the parsing process, but in general we must distinguish between *recognisers* which simply return a boolean result and *parsers* which must return at least one derivation of a string. Recognition is inherently cheaper than parsing and some care is required when interpreting reports of the asymptotic behaviour of algorithms. Algorithms such as Earley, CYK and Valiant return recognition matrices from which a single derivation may be extracted in $O(n^2)$ time, while Tomita style algorithms (Farshi, RNGLR, BRNGLR) and Reduction Incorporated (RI) parsers return a Shared Packed Parse Forest (SPPF) which encodes all derivations.

The parser versions of Earley's algorithm return only one derivation. As noted by Tomita [15], Earley's own attempt to return all derivations contains an error. As reported by Johnson [16], any parser which returns a Tomita style SPPF will be inherently of unbounded polynomial order. The RI parser also returns a Tomita style SPPF and hence has the same order as the RNGLR parser, despite the fact that the underlying RI recogniser is cubic. The BRNGLR algorithm returns a binary form of SPPF which is of at most cubic size, and is the only general parser discussed here which returns a representation of all the derivations and which is $O(n^3)$. For the rest of this paper we consider only the results for the underlying recognisers of each algorithm.

Algorithms are usually compared in terms of their space and time requirements at run time. A parser looking at short strings may be fast and need little run time space, but if its table contains many millions of states it may still be impractical, so we must also consider the size of the parser. Table 1 summarises asymptotic costs of the algorithms mentioned above. All of the parsers require $O(n^2)$ space for the structures that they construct at parse time.

Table 1. Asymptotic costs of some general recognisers

Algorithm	Recogniser size	Recogniser time
CYK 2-form	$O(\|\Gamma\|)$	$O(n^3)$
Valiant 2-form	$O(\|\Gamma\|)$	$O(n^{2.376})$
Earley	$O(\|\Gamma\|)$	$O(n^3)$
Farshi	$O(2^{\|\Gamma\|+log\|T\|+log\|\Gamma\|})$	$O(n^{k+1})$
RNGLR	$O(2^{\|\Gamma\|+log\|T\|+log\|\Gamma\|})$	$O(n^{k+1})$
BRNGLR	$O(2^{\|\Gamma\|+log\|T\|+log\|\Gamma\|})$	$O(n^3)$
RI	$O(2^{\|\Gamma\|})$	$O(n^3)$

$|\Gamma|$ means the size of the grammar, that is the sum of the lengths of all productions. k is the length of the longest production. $|T|$ is the number of terminals in the grammar. n is the length of the input string. For the last four algorithms, the parser size is the size of the underlying table.

Users of YACC might be surprised at the exponential bound for the size of the tables shown above. Knuth observed that LR(1) tables could be exponen-

tially large in the number of productions and did not present his work as being practical. It turns out that even LR(0) tables are worst-case exponential, but in practice pathological grammars are not encountered: YACC users expect small tables. Could we use LR(1) tables in practice and extend tools like YACC to full LR(1) parsing power? An (uncompressed) LR(1) table for ANSI-C produced by our GTB tool contains 1796 states and 278,380 cells, which is larger than the 382 state LALR table with 59,210 cells, but only by a factor of 4.7. We should also note that comb table compression will find more opportunities for space saving on the LR(1) table, so this ratio would improve for compressed tables. This is encouraging, but languages in which nonterminals appear in many different right hand side contexts (such as COBOL) would show less favourable ratios.

4 Recognition Matrices

Initial approaches to general parsing were based on the construction of a recognition matrix M: an $n \times n$ upper triangular array each element of which contains a set of grammar slots (dotted rules, or LR(0) items). An item is in $M_{i,j}$ if it matches a substring of the input which begins at token i and extends to token j. See [17] for a very approachable unified discussion of these algorithms.

In the *CYK* algorithm only slots with the dot at the end of a rule are permitted, that is, a CYK parser constructs a table of complete rule matches for all substrings of the grammar. For a grammar that is in Chomsky Normal Form, the table requires $O(n^2)$ space and may be constructed in $O(n^3)$ time. A derivation (but not the complete set of derivations) may be extracted from the table in $O(n^2)$ time. It is not hard to generalise the algorithm so that it accepts arbitrary grammars, but the run time is $O(n^{k+1})$ where k is the length of the longest production.

Valiant established that the same CYK recognition matrix can be constructed using a closure operation equivalent to the multiplication of boolean matrices. This can be performed in time $O(n^{2.376})$ but the constants of proportionality are very high, making the algorithm uncompetitive for the length of inputs normally encountered in programming language applications.

The Earley algorithm also effectively constructs a recognition matrix, but this time the entries are arbitrary slots, not just complete productions. As originally described, Earley constructs n sets of ordered pairs (slot, length of substring). We give experimental results for Earley parsing in section 6.

5 Generalised LR Parsers

The fundamental problem in all generalised parsers is to simulate nondeterminism on a deterministic machine by having a single parser program examine multiple execution threads corresponding to the potentially many derivations. Backtracking parsers do this by pursuing each avenue in a depth first fashion, looking backwards and forwards in the input string as necessary. An alternative strategy is to perform a breadth first search, maintaining a list of all possible

(partial) derivations up to the current input pointer. Knuth-style LR parsers (which may be using LR(k), SLR(k) or LALR(k) tables) are an attractive starting point for breadth first search because the entire parser configuration is encoded onto a stack of state numbers. Maintaining breadth first multiple threads of control therefore requires us only to maintain multiple stacks, with the parsing algorithm modified to ensure that all active stacks are scanned for reduction actions before advancing the input pointer and performing shift actions. If we use a pointer based representation for our stacks, then stack bifurcation leads to a tree structure. Tomita noted that the set of values that can appear at the top of a stack is just the finite set of parser states, and that the context free nature of stack activations meant that branches of the tree can be merged when they have the same leaf value. He called the resulting object a Graph Structured Stack (GSS). There is now a family of algorithms (the GLR parsers) based on the GSS representation of LR parse stacks.

The general principle of a GLR parser is as follows. In an LR parser, and by extension a GLR parser, a reduction by the rule $A ::= \alpha$ corresponds to retracing edges in the LR deterministic automaton that are labelled with the elements of α, and then traversing an out edge labelled A to the 'go to' state g. If $|\alpha|$ is the length of α, then a reduction in a deterministic parser corresponds to simply popping $|\alpha|$ states from the (single) stack, and then pushing g. In a GLR parser, edges in the GSS must be retraced: one interpretation of the GSS is that it is simply the trace of all the LR DFA traversals made during a parse. For each path of length $|\alpha|$ found, an edge is added to the destination from state g on the current stack frontier.

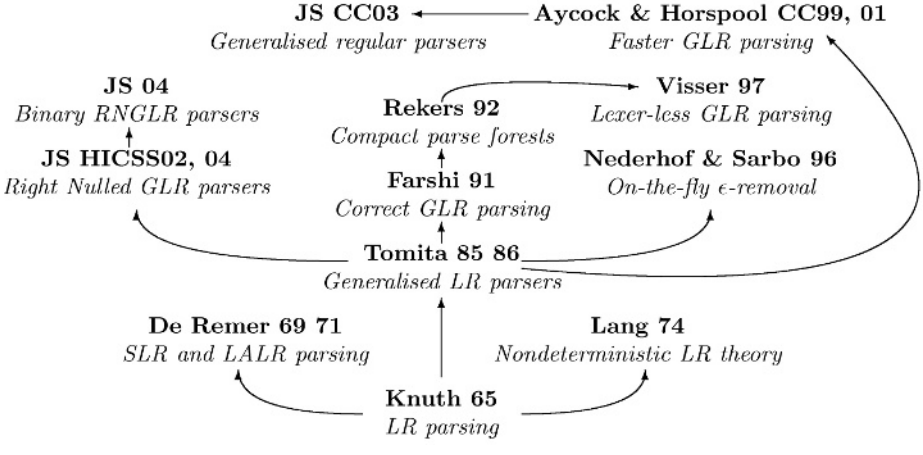

Fig. 1. The development of generalised LR parsers

Figure 1 shows the key developments in parsers which use GSS-like structures. Knuth's seminal 1965 paper contains most of the known results concerning

LR parsers. De Remer's LALR and SLR work was the basis of practical implementations such as YACC. Lang [18] considered the theoretical opportunities for simulating nondeterminism on a sequential machine. Tomita's work can be seen as a concrete implementation of Lang's work, with the GSS structure as a primary contribution. It is not clear whether Tomita knew of Lang's work at the time. Tomita [15] actually describes five algorithms. His first algorithm is correct for ϵ-free grammars, and the subsequent versions develop extra machinery to cope with reductions by ϵ-rules. Unfortunately, there is an error which leads to non-termination on grammars with hidden left recursion. Farshi's solution [6] is to apply an exhaustive search of all states active on the current frontier for new reductions. This search can be expensive, as we shall demonstrate below. Rekers' [7] and Visser's [8] theses modify Farshi's algorithm to produce denser parse forests and to run without a lexer respectively, but the search associated with ϵ-reductions is retained. Our RNGLR [9,10] algorithm presents an elegant fix to Tomita which uses a modified form of reduction. Our BRNGLR algorithm uses a 'binarisation' of the search process to yield a Tomita style parser which is cubic on any grammar. The RI and Aycock and Horspool parsers are inspired by, but not directly related to GLR parsing. We consider them in section 7.

6 Experiments on Earley and GLR Parsers

In this paper we have focused our experimental work on four grammars. We used standard ANSI C both because it is a widely used language and because it triggers some of the unpleasant behaviour in the RI algorithm. However, the grammar for ANSI C is ϵ-free so we have also used a grammar for Pascal based on the ISO-7185 grammar. This grammar is given in EBNF so the grammar we have used has been generated from the EBNF grammar using our `ebnf2bnf` tool.

To demonstrate the non-cubic behaviour of the RNGLR and Farshi algorithms we have used the grammar Γ_1 whose rule is

$$S ::= SSS \mid SS \mid b$$

This is one of a family of grammars used by Johnson to show that Tomita's original algorithm is worse than cubic in certain cases.

To further demonstrate the differences between the algorithms we use Γ_2

$$S ::= T \quad T ::= Ab \mid TTT \quad A ::= TbAAA \mid TTb \mid \epsilon$$

The grammars Γ_1 and Γ_2 are considered in detail in [11].

Parsers for Γ_1 and Γ_2 are exercised using a string of 100 and 50 b's respectively. Parsers for Pascal are run on tokenised versions of two source files, `quadratic` a quadratic root calculator which contains 279 tokens, and `tree viewer` a tree construction and visualisation application which has 4,425 tokens. Parsers for ANSI-C are run on a tokenised version of a Quine-McCluskey boolean minimiser containing 4,291 tokens.

We begin by comparing the total size of the Earley sets with the size of the GSS constructed by the RNGLR algorithm, and comparing the number of symbol comparisons required to construct the Earley sets with the number of edge visits required by the RNGLR algorithm to construct the GSS. As described in [11], an edge is only counted as visited if it is traversed as part of the application of a reduction. The creation of an edge is not counted as a visit.

Grammar	Input length	Earley set size	Earley symbol comparisions	RNGLR GSS nodes	RNGLR GSS edges	RNGLR edge visits
ANSI C	4,291	283,710	2,994,766	28,323	28,477	4,450
Pascal	279	7,648	53,068	1,263	1,266	364
Γ_1	100	25,355	1,628,586	398	14,949	12,405,821
Γ_2	50	15,902	905,536	578	19,737	14,726,271

We see that in general, the Earley sets are larger than the RNGLR GSS.

Next we compare the Farshi and the RNGLR algorithms which construct the same GSS so we report only the numbers of edge visits in each case. Farshi's algorithm specifies that each time an edge is added to an existing node in the GSS, for each node w not currently waiting to be processed and for each reduction of length q associated with w, all paths of length q from w which contain the new edge must be re-traced. In a naïve implementation, this means that all paths of length q must be searched in order to establish whether they include the required edge. This strategy clearly undermines the efficiency of Tomita's original approach which was designed to ensure that each path was only explored once, and in the limit will result in an explosion of the searching phase of the algorithm. To illustrate this point we have implemented the algorithm as written in [6] and reported the number of edge visits. In practice when searching for paths which contain the new edge we can terminate a search branch once it moves down a 'level' in the GSS, so we have also implemented an optimised version of Farshi's algorithm.

Grammar	Input length	Naïve Farshi edge visits	Optimised Farshi edge visits	RNGLR edge visits
ANSI C	4,291	30,754	30,484	4,450
Pascal	279	1,350	1,313	364
Pascal	4,425	23,505	22,459	5,572
Γ_1	100	2,429,702,294	15,154,234	12,405,821
Γ_2	50	3,382,571,879	15,770,426	14,729,271

As we have remarked, the worst case performance of both the Farshi and RNGLR algorithms is unbounded polynomial. In [11] we describe the BRNGLR algorithm which we prove is worst case $O(n^3)$ and has worst case $O(n^2)$ space requirements. However, the size of the GSS constructed by BRNGLR will be larger (by a small constant factor) than that constructed by the RNGLR algorithm. So in this case we report the relative GSS sizes and edge visits. As discussed in [11] the interesting case is Γ_1 on which the BRNGLR algorithm is cubic but the RNGLR and Farshi algorithms are quadratic.

Grammar	Input length	BRNGLR/RNGLR GSS nodes	BRNGLR/RNGLR GSS edges	BRNGLR/RNGLR GSS edge visits
ANSI C	4,291	30,331/28,323	30,512/28,477	4,446/4,450
Pascal	279	1,424/1,263	1,428/1,266	364/364
Pascal	4,425	23,521/21,039	23,655/21,135	5,570/5,572
Γ_1	100	496/398	29,209/14,949	1,407,476/12,405,821
Γ_2	50	764/578	32,078/19,737	448,899/14,726,271

7 Reduction Incorporated Parsers

Aycock and Horspool [12,19] described a novel parser in which the cost of a reduction is reduced in some cases. However, their algorithm can only be used on grammars without hidden left recursion. Our RI parser [13] is very closely related to their work, but is truly general. These parsers use a GSS-style structure to represent the multiple threads of control within the underlying automaton, however RI parsers do not use LR tables and do not behave like LR(1) parsers on LR(1) grammars, so we prefer to think of them as a separate family.

The motivation behind Aycock and Horspool's algorithm was to lower the constant of proportionality in Tomita style algorithms by reducing the amount of stack activity generated. Aycock and Horspool have reported an improvement in execution time for certain ambiguous grammars but to date there has been no published data on the size of the parser itself.

LR parsers are based on a DFA which recognises precisely the *viable strings* [20] of the input grammar. In the RI approach, for a grammar Γ with start symbol S, a new grammar Γ_S is constructed by 'terminalising' all but the non-hidden left recursive instances of all nonterminals. The (finite) set of viable strings is then generated and built into a trie, to which 'reduction' transitions are added. The result is a *reduction incorporated automaton* $RIA(\Gamma_S)$ which accepts precisely the language of Γ_S. Then, for each nonterminal A in Γ which was terminalised, an automaton $RIA(\Gamma_A)$ is constructed from the grammar obtained from Γ by taking A to be the start symbol. Finally an automaton is constructed by 'stitching together' the RIA's as follows. Each transition, in any $RIA(\Gamma_A)$, from a state h to a state k, say, labelled with a terminalised B is replaced with a transition labelled k from h to the start state of $RIA(\Gamma_B)$. We call the resulting push down automaton a *recursive call automaton* for Γ, $RCA(\Gamma)$. We have shown, [21], that $RCA(\Gamma)$ accepts precisely the language of Γ.

In order to construct the tries it is necessary to remove all but non-hidden left recursion from the grammar. We have given [21] a different method which allows for the construction of a nondeterministic automaton, called $IRIA(\Gamma)$, from any given grammar. We terminalise the grammar so that the only recursion is non-hidden left and right recursion, then build $IRIA(\Gamma_A)$ for nonterminals A. Finally, we use the standard subset construction algorithm to generate $RIA(\Gamma_A)$ from $IRIA(\Gamma_A)$ and build $RCA(\Gamma)$ as above. We have also provided an RCA traversal algorithm which terminates and is correct even for grammars with hidden left recursion. This algorithm is the RI algorithm which is a worst case

cubic recogniser for all context free grammars, and which has less associated stack activity than the RNGLR algorithm.

The Size of Reduction-Incorporated Automata

The speed up of the RI algorithm over the RNGLR algorithm is obtained by effectively unrolling the grammar. In a way this is essentially a back substitution of alternates for instances of nonterminals on the right hand sides of the grammar rules. Of course, in the case of recursive nonterminals the back substitution process does not terminate, which is why such instances are terminalised before the process begins. However, even without recursion we would expect full back substitution to generate an explosion in the number and size of the grammar alternates. It is often possible to show artificial pathological examples which generate disproportionately large objects, just as it is possible to generate exponentially large LR(0) tables. However, the generation of large RCA's does not require artificial examples.

We terminalised and then built the RIAs and RCAs for the Pascal, ANSI C, and the Γ_1 and Γ_2 grammars described above. In each case the grammars were terminalised so that all but non-hidden left recursion was removed before constructing the RIAs. Figure 2 shows the number of terminalised nonterminals, the number of instances of these terminalised nonterminals in the grammar, the number of symbol labelled transitions in $RCA(\Gamma)$ and the number of reduction labelled transitions in $RCA(\Gamma)$. (It is a property of $RCA(\Gamma)$ that the number of states is equal to the number of symbol labelled transitions + the number of terminalised nonterminal instances + 1.)

Table 2. The size of some reduction incorporated automata

Grammar	Terminalised nonterminals	Instances of terminalised nonterminals	Symbol transitions	Reduction transitions
Pascal	8	11	13,522	11,403
ANSI C	12	42	6,907,535	5,230,022
Γ_1	1	3	5	4
Γ_2	2	6	34	19

The basic problem is that each terminalised instance of a nonterminal B in the grammar generates a sub-automaton in $RIA(\Gamma_S)$ of the size of $RIA(\Gamma_B)$. We say that a nonterminal B *depends* on A if there is a rule $B ::= \alpha A \beta$ and that a *dependency chain* is a sequence of distinct nonterminals $B_0, B_1, \ldots, B_n$ such that B_i depends on B_{i+1}, $0 \leq i \leq n-1$. Then the number of copies of $RIA(\Gamma_{B_n})$ contributed to $RIA(\Gamma_S)$ by this chain 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 $O(2^{n+2})$. If we construct a grammar with n nonterminals each of which has k instances of the next nonterminal in its rules we have a grammar of size $O(n)$ which has an RIA of size at least $O(2k^{n+1})$. We have constructed the RIA's for the family of grammars, $Family_1$,

$$B_0 ::= B_1 \mid a_1 B_1 \quad B_1 ::= B_2 \mid a_2 B_2 \quad \ldots \quad B_{n-1} ::= B_n \mid a_n B_n \quad B_n ::= a$$

with $n = 2, 5, 10, 16$. (Our implementation was unable to cope with $n = 20$.) Note that there is no recursion in the grammars and $RCA(\Gamma)$=$RIA(\Gamma_{B_0})$. The sizes of the RIAs are given in the following table.

n	2	5	10	16
symbol transitions	15	127	4,095	262,144
reduction transitions	11	95	3,071	196,607

The potential explosion in automaton size is not of itself surprising: we know it is possible to generate an LR(0) automaton which is exponential in the size of the grammar. The point is that the example here is not artificial. In the ANSI C grammar the expression subtree contains a non-recursive dependency chain of depth 16, and five nonterminals in the chain contain at least three instances of the next nonterminal on the right hand sides of their rules. It is this that causes the RCA for ANSI C to be impractically large: the total number of edges in the automaton is 12,137,557.

In the above case we could right factor the grammar, replacing each rule $B_{i-1} ::= B_i \mid a_i B_i$ with two rules $B_{i-1} ::= C_i B_i$, $C_i ::= \epsilon \mid a_i$. This generates the following results

n	2	5	10	16	20
symbol transitions	9	18	33	51	63
reduction transitions	8	17	32	50	62

However, this approach can not be applied to grammars which can not be right factored in this way, for example the family $Family_2$

$$B_0 ::= B_1 \mid a_1 B_1 b_1 \quad B_1 ::= B_2 \mid a_2 B_2 b_2 \quad \ldots \quad B_{n-1} ::= B_n \mid a_n B_n b_n \quad B_n ::= a$$

Further, the point of a general parsing technique is that it can be used without modification to the grammar, allowing the grammar to be written in a way which is natural for the required semantics.

So far we have only discussed the size of the final automaton $RCA(\Gamma)$. We also need to worry about the space required to construct the parser. The construction method that we have described [13] produces intermediate automata, $IRIA(\Gamma_A)$, to which the subset construction is then applied to generate the trie-style automata $RIA(\Gamma)$. The $IRIA(\Gamma_A)$ do not have the merging of paths on common left hand ends which is a feature of the $RIA(\Gamma_A)$ thus the explosion

described above will also occur in $IRIA(\Gamma_{B_0})$ (but not in $RIA(\Gamma_{B_0})$) for the grammar family $Family_3$

$$B_0 ::= B_1 \mid B_1 a_1 \quad B_1 ::= B_2 \mid B_2 a_2 \quad \ldots \quad B_{n-1} ::= B_n \mid B_n a_n \quad B_n ::= a$$

Perhaps more importantly, the same effect is triggered by left recursive rules. For a rule in Γ of the form

$$A ::= A\beta \mid \alpha$$

$IRIA(\Gamma_A)$ contains a sub-automaton equivalent to an automaton $IRIA(\Gamma_{A'})$ generated by the rule $A' ::= \alpha$. The recursive call to A implies that $IRIA(\Gamma_A)$ effectively contains two copies of $IRIA(\Gamma_{A'})$. The effect becomes explosive when there is more than one left recursive rule

$$A ::= A\beta \mid A\gamma \mid A\delta \mid \alpha$$

In this case $A' ::= A\gamma \mid A\delta \mid \alpha$ and so $IRIA(\Gamma_{A'})$ contains two copies of $IRIA(\Gamma_{A''})$ and of $IRIA(\Gamma_{A'''})$ where $A'' ::= A\delta \mid \alpha$ and $A''' ::= A\gamma \mid \alpha$.

Again this is not an artificial example. In the ANSI C grammar one nonterminal, `postfix_expression`, in the expression subtree has seven alternates, six of which begin with a recursive call to itself.

All these sub-automata will be merged by the subset algorithm and hence do not cause an explosion in the size of the RIA, but using the method described in [13] they must be constructed. Thus we have modified our IRIA construction method slightly so that, as for the trie construction method, some aspects of the subset construction are carried out 'on-the-fly'.

IRIA Representation

The IRIA are not held in memory in their full form. There are *headers* representing each instance of each nonterminal in the terminalised grammar Γ_S, and these headers effectively contain links into the grammar Γ_S in such a way that the final structure can be traversed as though it were $IRIA(\Gamma_S)$. This means that for practical purposes the size of $IRIA(\Gamma_S)$ is the number of headers plus the size of Γ. The construction method described in [13] requires a new header for each direct left recursive call of a nonterminal to itself, but in fact the original header can be re-used in this case because the same automaton will be constructed either way by the subset construction. Thus we have modified the IRIA construction algorithm to re-use the original header in the case where a nonterminal instance is the result of direct left recursion.

Using this we find that the IRIAs have a size which is comparable with the corresponding RIAs. For example, for the Pascal and ANSI C grammars we have

grammar	total IRIA headers	RIA symbol transitions	RIA reduction transitions
Pascal	7,759	13,522	11,403
ANSI C	9,594,266	6,907,535	5,230,022

For both Pascal and ANSI C the numbers of headers required for each of the sub IRIA is between half and two thirds of the number of symbol transitions in the corresponding RIA. The increase in the total number of headers required for ANSI C is because the `conditional_expression` IRIA, whose size dominates the total size, contains some non-left recursive rules which could be left factored, i.e. rules of the form $A ::= B \mid BC$.

We have further modified our construction procedure so that it effectively works on grammars written in a form of EBNF. The input grammar supplied by the user is still in BNF form, and the output derivation trees are still with respect to this grammar. But internally the IRIA construction procedure uses the grammar in a factored form: $A ::= B(\epsilon \mid C)$. This has the effect of ensuring that the size of the IRIAs is comparable with the size of the final RIA. We could extend this approach further and use, internally, right factored grammars but the problem then is dealing correctly with the reduction transitions. We return briefly to this below, when we discuss the issues surrounding derivation tree generation.

Controlling the Size of the RCA

We conclude this discussion by observing that ultimately the way to control the explosion in the size of $RCA(\Gamma)$ is to introduce more nonterminal terminalisations. We must terminalise the grammar in such a way that Γ_S has no self embedding for the process to be correct, but the process is still correct if further terminalisations are applied. By terminalising the middle nonterminal in a dependency chain we will effectively replace the single automaton whose size is $O(2^{n+2})$ with two automata $RIA(\Gamma_{B_0})$ and $RIA(\Gamma_{B_{n/2}})$ each of size $O(2^{(n+1)/2})$. If we take this approach with the family $Family_1$ of grammars above and terminalise by simply replacing the rule for $B_{n/2}$ with

$$B_{(n/2)-1} ::= \ (B_{n/2})^{\perp} \mid a_{n/2}(B_{n/2})^{\perp}$$

and leaving the other rules the same (we assume for simplicity that n is even) we get the following results

n	2	6	10	16	20
symbol transitions	12	54	222	1,790	7,678
reduction transitions	8	38	158	1,278	5,118

Here we have reported the total numbers of transitions in the two RIA automata created in each case.

Taking this approach with the ANSI C grammar, terminalising non-recursive instances of a nonterminal in a rule in the middle of the expression sub-grammar, we get the following results

Grammar	Terminalised nonterminals	Instances of terminalised nonterminals	Symbol transitions	Reduction transitions
ANSI C	13	47	21,895	16,573

Of course, this approach will also work for $Family_2$ and can significantly reduce the parser size for any grammar with long dependency chains. However, it is at the cost of introducing more recursive calls when the parser is run and hence the (practical but not asymptotic) run time performance and space requirements of the parser will be increased. Thus there is an engineering tradeoff to be made between the size of the parser and its performance.

As we have remarked above, the parser version of the RI algorithm is worst case unbounded polynomial. We expect to be able to modify our SPPF construction so that it produces binary SPPFs in a form similar to those produced by the BRNGLR algorithm by internally treating a grammar rule of the form $A ::= XYZW$ as though it were an EBNF rule of the form $A ::= ((XY)Z)W$. This approach will automatically produce smaller parsers (RCAs) than those currently produced by the Aycock and Horspool approach in the case where the grammar can be right factored. This will also permit the grammar right-factoring mentioned above, but this is still work in progress.

8 Conclusions

We conclude that Earley and GLR style parsers are practical on current machines for programming language applications. Of these, the BRNGLR algorithm is asymptotically fastest but requires slightly more space than the polynomial RNGLR algorithm. RNGLR performs fewer GSS traversals than Farshi's (and by extension Visser's) algorithms. The original Tomita algorithm fails to terminate on hidden left recursion, and in any case constructs larger GSS's as a result of sub-frontier creation. Reduction Incorporated algorithms at present strain the limits of practicality even for standard programming languages. The underlying automaton for the RI parser for ANSI-C contains over 12×10^6 edges. However RI algorithms may become generally useful in the future and do have some present applications. An engineering tradeoff is possible: they can be made practical by introducing additional sub-automaton calls effectively trading execution time for parser size.

References

1. Parr, T.J.: Language translation using PCCTS and C++. Automata Publishing Company (1996)
2. Eggert, P.: `http://compilers.iecc.com/comparch/article/03-01-042`. (2003)
3. Durbin, R., Eddy, S., Krogh, A., Mitchison, G.: Biological Sequence Analysis. Cambridge University Press (1998)
4. van den Brand, M., Heering, J., Klint, P., Olivier, P.: Compiling language definitions: the ASF+SDF compiler. ACM Transactions on Programming Languages and Systems **24** (2002) 334–368
5. Johnstone, A., Scott, E.: Generalised recursive descent parsing and follow determinism. In Koskimies, K., ed.: Proc. 7th Intnl. Conf. Compiler Construction (CC'98), Lecture notes in Computer Science 1383, Berlin, Springer (1998) 16–30

6. Nozohoor-Farshi, R.: GLR parsing for ϵ-grammars. In Tomita, M., ed.: Generalized LR parsing. Kluwer Academic Publishers, Netherlands (1991) 60–75
7. Rekers, J.G.: Parser generation for interactive environments. PhD thesis, Universty of Amsterdam (1992)
8. Visser, E.: Syntax definition for language prototyping. PhD thesis, Universty of Amsterdam (1997)
9. Scott, E., Johnstone, A.: Reducing non-determinism in reduction modified GLR parsers. To appear in: Acta Informatica (2004)
10. Johnstone, A., Scott, E.: Generalised reduction modified LR parsing for domain specific language prototyping. In: Proc. 35th Annual Hawaii International Conference On System Sciences (HICSS02). IEEE Computer Society, IEEE, New Jersey (2002)
11. Scott, E., Johnstone, A., Economopoulos, G.: BRN-table based GLR parsers. Technical Report TR-03-06, Royal Holloway, University of London, Computer Science Department (2003)
12. Aycock, J., Horspool, N.: Faster generalised LR parsing. In: Compiler Construction: 8th International Conference, CC'99. Volume 1575 of Lecture Notes in computer science., Springer-Verlag (1999) 32 – 46
13. Johnstone, A., Scott, E.: Generalised regular parsers. In Hedin, G., ed.: Compiler Construction, 12th Intnl. Conf, CC2003. Volume 2622 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2003) 232–246
14. Dodd, C., Maslov, V.: `http://www.siber.com/btyacc`. (2002)
15. Tomita, M.: Efficient parsing for natural language. Kluwer Academic Publishers, Boston (1986)
16. Johnson, M.: The computational complexity of GLR parsing. In Tomita, M., ed.: Generalized LR parsing. Kluwer Academic Publishers, Netherlands (1991) 35–42
17. Graham, S.L., Harrison, M.A.: Parsing of general context-free languages. Advances in Computing **14** (1976) 77–185
18. Lang, B.: Deterministic techniques for efficient non-deterministic parsers. In: Automata, Languages and Programming: 2nd Colloquium. Lecture Notes in computer science, Springer-Verlag (1974) 255 – 269
19. Aycock, J., Horspool, R.N., Janousek, J., Melichar, B.: Even faster generalised LR parsing. Acta Informatica **37** (2001) 633–651
20. Aho, A.V., Sethi, R., Ullman., J.D.: Compilers: principles techniques and tools. Addison-Wesley (1986)
21. Scott, E., Johnstone, A.: Table based parsers with reduced stack activity. Technical Report TR-02-08, Royal Holloway, University of London, Computer Science Department (2002)