Information and Software Technology 1995 **37** (10) 547–555

Implementing Prolog Definite Clause Grammars with SLR(1) parsers on the Relational Algebra Accelerator

Greg Michaelson

Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton EH14 4AS, Scotland, UK
e-mail:greg@cee.hw.ac.uk

Definite Clause Grammars (DCGs) are a Prolog extension which are widely used to specify and implement front ends to Prolog based systems. Efficient restricted DCG implementations may be based on LR(K) parsing techniques. The use of the Relational Algebra Accelerator (RAA), a bit serial/word parallel logic co-processor, for fast access to LR parse tables brings substantial performance improvements over software DCG implementations. The generation of SLR(1) parsers from DCGs and their implementation using the RAA are discussed, and performance figures are presented.

Keywords: Prolog, DCGs, SLR(1) parsing, Relational Algebra Accelerator

Declarative languages like Prolog are moving steadily from research and teaching to industrial and commercial use for system prototyping and development. Such languages enable a high degree of correspondence between program and data structure through case structured rule or function definitions and pattern matching. This enables the relatively quick construction of succinct programs.

Many Prolog implementations support the Definite Clause Grammar (DCG) notation for defining context free grammars. These are particularly useful for experimenting with text based user interfaces as system actions may be associated directly with different syntactic constructs.

As always, alas, there is a price to pay. Prolog is based on backtracking as a fundamental program control mechanism. This is used to search data bases of facts and rules so that a partial match may be undone in the light of subsequent processing. The steps leading to that match may then be retraced to try to find other better matches. Backtracking is computationally expensive as a record must be kept of all partial match points so that they may potentially be undone.

Prolog DCGs are a very general form of grammar rule and depend on backtracking for their implementation, to search all possible paths through a grammar when matching a sentence. This can lead to appreciable delays in responses to DCG based commands and to noticeable variations in response times for apparently similar commands. However, most text based interfaces do not require this generality and the corresponding commands can, in principle, be identified on a single pass through the grammar without backtracking.

By restricting the form of DCGs, much faster implementations may be provided through non-backtracking automata which can be generated automatically in Prolog from DCGs. While such automata offer performance improvements over backtracking based implementations, they still depend on the underlying Prolog system. A second improvement may be found by utilizing special hardware to assist the searching of Prolog data bases to minimize backtracking.

The Relational Algebra Accelerator (RAA) is a co-processor which can perform all the set theoretic and relational operations. It can also be used in computing decision tables, and to manipulate quadtree and octree spatial data representations. Its main applications are in support of main memory and deductive databases. In particular, UNSW Prolog has been implemented using the RAA to select suitable candidates for matches from very large fact and rule databases.

This paper discusses the use of the RAA to support Prolog SLR(1) parsing automata generated from DCGs. The next sections consider DCGs and the RAA in more detail. Subsequent sections present the generation of automata from DCGs and consider the resultant performance when implemented using the RAA.

Definite Clause Grammars and their implementation

Definite Clause Grammars are a means of defining and

utilizing extended context free grammars within Prolog, developed by Warren and Pereira[1]. A DCG rule has the form:

```
⟨non-terminal⟩ --⟩ ⟨body⟩
```

where the body is a conjunction of terminals and non-terminals. Such a rule may be read as:

```
to recognize the ⟨non-terminal⟩, recognize the ⟨body⟩.
```

DCG non-terminals are Prolog terms and terminals are lists of Prolog terms. At simplest, non-terminals and terminals are Prolog atoms. For example, the BNF for binary numbers:

```
⟨digit⟩ ::= 0 | 1
⟨binary⟩ ::= ⟨digit⟩ | ⟨digit⟩ ⟨binary⟩
```

might be written as the DCG:

```
digit --⟩ [0].
digit --⟩ [1].
binary --⟩ digit, binary.
binary --⟩ digit.
```

Note that in DCGs, unlike BNF, rules for the same non-terminals are not amalgamated into a single composite right-hand side with options.

In many implementations, DCGs are translated directly into Prolog clauses which are then used to parse lists of terminal symbols. Each DCG is converted to a parsing clause with explicit parameters, which consumes an initial terminal symbol list and returns a final terminal symbol list. For example, the above DCG translates to:

```
digit(T1,T2) :- connects(T1,0,T2).
digit(T1,T2) :- connects(T1,1,T2).
binary(T1,T3) :- digit(T1,T2), binary(T2,T3).
binary(T1,T2) :- digit(T1,T2).
connects([S|L],S,L).
```

The connects construct may be removed by moving terminal symbols in the body into the clause head to take advantage of unification. Thus, the digits clauses in the above DCG may be rewritten as:

```
digit([0|T],T).
digit([1|T],T).
```

DCG terminal and non-terminal symbols may also be arbitrary terms, including nested structures. These may be used to pass information between the rule non-terminal and the rule body, typically for attribute transmission or to build trees. For example, the above DCG might be extended to build a tree representation of binary numbers:

```
digit(0) --⟩ [0].
digit(1) --⟩ [1].
binary(numb(D,B)) --⟩ digit(D), binary(B).
binary(D) --⟩ digit(D).
```

After translation, defined parameters precede the generated parameters in the clausal form. For example, the previous DCG translates to:

```
digit(0,[0|T],T).
digit(1,[1|T],T).
binary(numb(D,B),T1,T3)) :- digit(D,T1,T2),
                            binary(B,T2,T3).
binary(B,T1,T2) :- digit(D,T1,T2).
```

For example:

```
?- binary(T,[1,0,1],R).
T = numb(1,numb(0,1))
R = [ ]
```

DCGs may also contain semantic actions. These are arbitrary Prolog terms within { }. For example, to calculate the value of a binary number:

```
digit(0) --⟩ [0].
digit(1) --⟩ [1].
binary(N1,N3) --⟩ digit(D),
                  {N2 is 2*N1+D},
                  binary(N2,N3).
binary(N1,N2) --⟩ digit(D),
                  {N2 is 2*N1+D}.
```

For translation, the semantic actions are placed at the corresponding positions in the clause body. For example, the previous DCG translates to:

```
digit(0,[0|T],T).
digit(1,[1|T],T).
binary(N1,N3,T1,T3) :- digit(D,T1,T2),
                       N2 is 2*N1+D,
                       binary(N2,N3,T2,T3).
binary(N1,N2,T1,T2) :- digit(D,T1,T2),
                       N2 is 2*N1+D.
```

For example:

```
?- binary(0,B,[1,0,1],R).
B = 5
R = [ ]
```

DCGs are more powerful than unrestricted context free grammars but this power is unnecessary for many applications. Unrestricted context free grammars are computationally expensive to implement as the equivalent parsing automata are non-deterministic. DCG implementations based on translation to Prolog clauses utilize the full underlying backtracking mechanism which will be sub-optimal for many grammars.

For example, consider the following contrived printer command:

```
print --⟩ [print], ['-txt'], file.
print --⟩ [print], file.
```

where a ⟨file⟩ may not start with a -. The intention is that the ⟨file⟩ is implicitly PostScript unless -txt is present in which case it is raw text. Now, consider parsing:

```
print document
```

where document is some file. Parsing involves:

```
match: print - succeeds
match: -txt - fails
backtrack
match: print - succeeds
match: ⟨file⟩ - succeeds
```

Reversing the rules makes no difference:

```
print --⟩ [print], ['-txt'], file.
print --⟩ [print], file.
```

Consider matching:

```
print -txt document
```

Now:

```
match: print - succeeds
match: ⟨file⟩ - fails
backtrack
match: print - succeeds
match: -txt - succeeds
match: ⟨file⟩ - succeeds
```

However, if the parser were able to look ahead after making a match to decide what to do next, then there would be no need to backtrack. For the last example, the two options for the rule differ after the common first print has been matched:

```
match: print - succeeds
lookahead: -txt - choose 2nd option
match: -txt - succeeds
match: ⟨file⟩ - succeeds
```

There are extremely useful sub-classes of context free grammars, for example the LL(K) and LR(K)[2] grammars for which more optimal parsing automata may be constructed. Many compiler generators produce LR parsers, for example YACC[3]. LR parsers have also been used as the basis of more efficient DCG implementations. For example, AID[4] is an SLR(1) parser generator for DCGs. Morley[5] has used constraints in SLR(1) parser generation from DCGs to restrict non-determinism.

However, parser performance is still ultimately bound by the performance of the underlying Prolog system. Here we discuss an SLR(1) parser generator for DCGs which utilizes the Relational Algebra Accelerator (RAA) co-processor to obtain substantial performance improvements.

The Relational Algebra Accelerator

The Relational Algebra Accelerator[6] is a bit serial/word parallel co-processor. It may be thought of as a CPU with 256 registers each of which is 4K bits long. The RAA enables logical operations to be carried out between registers with a performance of 100 million bit operations per second. In typical applications, a large data set is encoded and held in the memory. The data set is then accessed and manipulated through bit operations performed on all records in parallel.

The RAA has been used as a pseudo-associative memory for fast clause retrieval from Prolog clause databases[7]. Superimposed coding is used to encode database clauses and queries. Coded clauses are held in the RAA memory and a query code is used to construct a sequence of logical operations which are performed on the clause codes. The result is a bit sequence which identifies those database clauses which may satisfy the query. Thus, the effect is of pre-unification filtering: with very large databases this reduces unification substantially.

For example, consider the Prolog facts:

```
fruit(apple,granny_smith).
fruit(apple,golden_delicious).
computer(apple,macintosh).
```

With some suitable superimposed coding these might be represented by the following 8 bit codes:

```
fruit(apple,granny_smith).      = 11111100
fruit(apple,golden_delicious).  = 11110101
computer(apple,macintosh).      = 11010011
```

These might be loaded into 8 RAA registers as:

```
          register
  7 6 5 4 3 2 1 0
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | fruit(apple,granny_smith).
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | fruit(apple,golden_delicious).
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | computer(apple,macintosh).
```

The query:

```
fruit(apple,X).
```

might then be coded as:

```
fruit(apple,X). = 10110100
```

Thus, we wish to find all facts with:

```
register 7 = 1
register 5 = 1
register 4 = 1
register 2 = 1
```

which is equivalent to:

```
register 7 AND register 5 AND register 4 AND register 2
```

Carrying out these operations on the RAA registers gives:

```
| 1 | fruit(apple,granny_smith).
| 1 | fruit(apple,golden_delicious).
| 0 | computer(apple,macintosh).
```

so the facts:

```
fruit(apple,granny_smith).
fruit(apple,golden_delicious).
```

are good candidates for solutions to the query.

Note that superimposed coding, as with any coding technique based on a reduction in discriminatory information, may result in a number of distinct facts having the same code. This will in turn result in false drops from query processing. None the less, with large Prolog databases, this approach results in substantial savings over ordinary unification. The RAA has been integrated[8] with UNSW Prolog[9] using superimposed coding. On a SUN 3/160 this runs at 100 kLIPS compared with 2 kLIPS without the RAA.

The RAA is well suited to LR parser implementations. LR parsers are based on state machines where the next action is determined by the current parser state and current input symbols. Such parsers operate in linear time. State machines are highly amenable to table driven implementations and the tables are usually sparse. Thus, parsers for restricted DCGs may be implemented on the RAA as state transition tables, represented as Prolog clauses, driven by a simple Prolog program.

LR parsing

LR parsing[2] is based on Left to right parsing with a

Rightmost derivation. It is an efficient, non-backtracking method which can be used to recognize a wide range of useful grammars including almost all context free programming language constructs.

An LR parser is a pushdown automata. It works from left to right along a sequence of input symbols changing state as it recognizes grammatical constructs corresponding to grammar rules. Its components are the input, which is the sequence of symbols to be parsed, the stack, which holds consumed symbols from the input and corresponding state information, and the parsing table, which indicates the parsing action for each possible state and input symbol.

Parsing tables have entries of the form:

```
table(⟨state⟩,⟨symbol⟩) = ⟨action⟩
```

For a given input symbol, ⟨symbol⟩, in a given state, ⟨state⟩, the parsing table is consulted, the appropriate action, ⟨action⟩, is carried out and a new state is entered ready to process the new current input symbol. Parsing actions depend on how the current input symbol affects the recognition of the current grammatical construct.

A shift action, shift(⟨state⟩), takes place if the parser is in the middle of recognizing some construct and the current input symbol, though valid, does not complete that construct. Then, the current state and symbol are pushed onto the stack, the next input symbol becomes the current symbol and a new state, ⟨state⟩, is entered to deal with it.

A goto action, goto(⟨state⟩), takes place if the parser is in the middle of recognizing some construct and a subconstruct has been recognized which, though valid, does not complete that construct. Then, a new state, ⟨state⟩, is entered to deal with the current input symbol.

A reduce action, reduce(⟨rhs⟩), takes place if the current input symbol indicates that the parser has successfully recognized a construct, say corresponding to the rule:

```
l --⟩ r1, r2, . . . rN
```

The stack will already contain the symbols for the right-hand side of the construct, r1, r2, ... rN, and the corresponding states from preceding shift actions. These symbols and states are removed from the stack uncovering the state preceding the start of the recognition of the construct, say ⟨old state⟩. The non-terminal for the construct, l, is pushed onto the stack to indicate that the construct has been recognized and a new state:

```
table(⟨old state⟩,l) = goto(⟨new state⟩)
```

is entered to deal with the current input symbol.

An accept action, accept, takes place when the sentence construct has been recognized and the parse has succeeded. An error action, error, takes place if the current input symbol is inappropriate in the context of the current state of recognition of the current construct. Error recovery may then be invoked or the parse may be abandoned.

Simple LR parsers with 1 place lookahead, SLR(1), are a subset of the more general LR(1) parsers which are very easy to implement.

For example, the binary number DCG:

```
digit --⟩ [0].
digit --⟩ [1].
binary --⟩ digit.
binary --⟩ digit, binary.
```

has the SLR(1) parse table:

```
table(0,[0]) = shift(3)
table(0,[1]) = shift(4)
table(0,digit) = goto(2)
table(0,binary) = goto(1)

table(1,[$]) = accept

table(2,[0]) = shift(3)
table(2,[1]) = shift(4)
table(2,[$]) = reduce(binary --⟩ digit)
table(2,digit) = goto(2)
table(2,binary) = goto(5)

table(3,[0]) = reduce(digit --⟩ [0])
table(3,[1]) = reduce(digit --⟩ [0])
table(3,[$]) = reduce(digit --⟩ [0])

table(4,[0]) = reduce(digit --⟩ [1])
table(4,[1]) = reduce(digit --⟩ [1])
table(4[$]) = reduce(digit --⟩ [1])

table(5,[$]) = reduce(binary --⟩ digit, binary)
```

For example, to parse [1,0] starting in state 0:

```
input = [1,0] stack = [0]
table(0,[1]) = shift(4)

input = [0] stack = [0,[1],4]
table(4,[0]) = reduce(digit --⟩ [1])

input = [0] stack = [0,digit]
table(0,digit) = goto(2)

input = [0] stack = [0,digit,2]
table(2,[0]) = shift(3)

input = [ ] stack = [0,digit,2,[0],3]
table(3,[$]) = reduce(digit --⟩ [0])

input = [ ] stack = [0,digit,2,digit]
table(2,digit) = goto(2)

input = [ ] stack = [0,digit,2,digit,2]
table(2,[$]) = reduce(binary --⟩ digit)

input = [ ] stack = [0,digit,2,binary]
table(2,binary) = goto(5)

input = [ ] stack = [0,digit,2,binary,5]
table(5,[$]) = reduce(binary --⟩ digit, binary)

input = [ ] stack = [0,binary]
table(0,binary) = goto(1)

input = [ ] stack = [0,binary,1]
table(1,[$]) = accept
```

Implementing SLR(1) parsers on the RAA

Parser generation

An SLR(1) parser generator, based on the algorithms from Aho, Sethi and Ullman[2], was written in UNSW Prolog. Prolog proved to be an excellent implementation language for prototyping: the initial parser generator was completed very quickly. However, the use of Prolog with the above algorithms resulted in highly recursive clauses operating on lists which execute much more slowly than the equivalent iterative algorithms operating on arrays in a compiled language. They are also slow because of UNSW's sub-optimal clause indexing mechanism. In addition, they

quickly consume all available stack space on relatively small DCGs as UNSW Prolog has no tail recursion optimization.

Subsequently, the clauses were converted to MU-Prolog[10] and then to SICStus Prolog[11]. The latter has an efficient clause indexing mechanism and tail recursion optimization, and enabled the construction of parse tables from larger DCGs.

Adapting parse tables for the RAA

To begin with, tables similar in format to those above were produced based on simple shift/reduce/goto entries. Table entries were of the form:

```
t(⟨state⟩,⟨symbol⟩,⟨action⟩)
```

where:

```
⟨action⟩ ::= shift(⟨state⟩) |
             goto(⟨state⟩) |
             reduce(⟨number⟩,⟨symbol⟩)
```

where ⟨number⟩ is the number of right-hand symbols to be reduced. The end of input symbol was rstop instead of $.

For the binary number grammar, the table entries for state 0 are:

```
t(0,[0],shift(3)).
t(0,[1],shift(4)).
t(0,digit,goto(2)).
t(0,binary,goto(1)).
```

The whole table is shown in Appendix A.

Parsing was driven by the explicit parse clause shown in Appendix B. This first attempt enabled table generation to be checked but was very slow, partly because of explicit stack popping for reduce actions. It was also hard to adapt to parameterized DCGs.

Next, tables were generated in a recursive form with stock popping for reduce actions effected through explicit unification. Table entries took the form:

```
t([⟨state⟩|⟨rest of stack⟩],
  [⟨symbol⟩|⟨rest of symbols⟩],
  ⟨result⟩,⟨left over symbols⟩) :-
t(⟨params for next action⟩)
```

For example, the state 0 table entries for the binary numbers DCG are:

```
t([0|SY],[[0]|[T],TR,R) :- t([3,[0],0|SY],T,TR,R).
t([0|SY],[[1]|[T],TR,R) :- t([4,[1],0|SY],T,TR,R).
goto(0,digit,2).
goto(0,binary,1).
```

The full table is in Appendix C. This gave a substantial improvement in speed and enabled the introduction of parameterized DCGs. However, when parse tables were loaded into the RAA, the performance was rather disappointing.

First of all, the RAA/UNSW Prolog interface involves the explicit reconstruction and initiation of a clause body after clause selection. This was overcome by putting the clause body into the clause head. The potential infinite recursive regress was avoided by abstraction at the recursion point and the use of unification to pick up the abstraction. The recursive call is then invoked explicitly:

```
rt(A,B,C,D) :- t(A,B,C,D,E), E. % pick up RHS at E &
                                % invoke E - E will
                                % call rt
exp(E,T,R) :- rt([0],E,T,R).
```

For example, the state 0 table entries for the binary number grammar are:

```
t([0|SY],[[0]|T],TR,R,rt([3,[0],0|SY],T,TR,R)).
t([0|SY],[[1]|T],TR,R,rt([4,[1],0|SY],T,TR,R)).
goto(0,digit,2).
goto(0,binary,1).
```

The full table is in Appendix D.

The second source of poor performance was because the general purpose superimposed coding used normally to load Prolog clauses into the RAA was not appropriate to this application. There was insufficient variation in the codes for table entries and so there were several false drops on each RAA access. Charles Chung observed that entries are distinguished by state and symbol and he introduced a field encoding to represent entries. This set 1 bit in 64 bit fields for both states and symbols giving a unique match on each RAA access. These changes brought significant gains in performance, which are presented below.

Problems with table generation from DCGs

Table generation involves, in part, the use of symbol sequences from DCG productions, for example to identify non-terminal symbols and production right-hand sides. This led to difficulties with the movement of Prolog variables from parameterized rules into inappropriate scopes. These may arise from user mistakes where symbols introduced for lexical analysis and symbols in the grammar contain the same variable names. This may result in inappropriate sharing, for example, between a symbol on the stack and a symbol in the input sequence.

However, there may also be computer generated clashes if the symbol starting a production is also the symbol following a production. Consider for example the rule for a line of BASIC:

```
line --> [number(N)], command.
lines --> line.
lines --> line, lines.
```

Now, note from the second lines production that a line may be followed by lines and hence by another line. Thus [number (N)] which starts a line is also a follower for line so there is a reduce action to recognize:

```
[number(N)], command
```

having encountered the symbol [number (N)]. Thus there is a table entry with [number (N)] both on the stack and in the input:

```
t([...,[number(N)],_,line,NS|NSY],
  [[number(N)]|_],...) :- ...
```

Alas, the Ns should be distinct. This was solved by renaming variables in symbols before table entry construction to ensure distinct names.

Another source of difficulty was the treatment of semantic actions in DCGs. These present no problems when they

occur at the end of productions. However, it is not clear how to handle actions within productions. The construction of tables separates the entries for recognition of sub-items from the entry for the recognition of the item. Thus, it is hard to decide whether to carry out the semantic action for a sub-item on the shift action for that sub-item or on the reduce action for the enclosing item. If the semantic action is carried out on the shift action, when the sub-item is recognized, then it cannot communicate information to the rest of the item by implicit variable sharing. If the semantic actions for sub-items are carried out together on the reduce action, at end of the item, then they cannot affect sub-item recognition.

In parsing it is usual to restrict semantic actions to the end of productions so the same restriction to DCGs has been adopted. Thus, it is necessary to introduce explicit sub-productions for intermediate actions, and intermediate actions will not interact other than via the Prolog database through the explicit use of assert/retract.

Testing and timing

LR parser tests

The parser generator was first tested with a DCG for restricted [*sic*] English, from Clocksin and Mellish[12] (after Pereira and Warren)[1]. This had 15 productions and the corresponding parser had 55 actions.

More substantial tests were then made with a DCG for an SQL subset, based on a YACC grammar in Heerjee and Sadeghi[13]. This had 27 productions and the corresponding parser had 162 actions. The SQL parse table had 40 states with 18 terminal symbols. Times in seconds for test sentences of symbols, with and without the RAA, were:

symbols	actions	RAA time		no RAA time	
10	25	.26	diff	.50	diff
15	38	.39	.13	.72	.22
20	51	.51	.12	.94	.22
25	64	.65	.14	1.17	.23
30	77	.77	.12	1.40	.23
35	90	.90	.13	1.61	.21
40	103	1.04	.14	1.84	.23
45	116	1.13	.09	2.07	.23
50	129	1.27	.14	2.29	.22
55	142	1.40	.13	2.51	.22

Note that the difference in times (diff) as the number of symbols in successive test sentences increases is almost constant, corresponding to the anticipated linear correlation between sentence length and parse time.

The average time per action was:

```
RAA     .01
no RAA .018
```

Thus, parsing with the RAA was around 1.8 times as fast.

Assuming that the same amount of time was spent processing each successfully matched clause with and without the RAA, and that the RAA time for clause matching is insignificant, then the difference in times as a proportion of the time without the RAA:

```
(no RAA - RAA) / no RAA
```

gives the proportion of time spent in unsuccessful clause matching without the RAA. Here it was around 45% of the time.

A DCG for a BASIC subset was also tested. This had 43 productions and the corresponding parser had 449 actions. The BASIC parse table had 81 states with 32 symbols. Times in seconds with and without the RAA were:

symbols	actions	RAA	no RAA
58	153	1.67	7.37
95	258	2.87	12.34
146	401	4.41	19.26
179	481	5.26	23.16

The average time per action was:

```
RAA     .011
no RAA .048
```

Thus, parsing with the RAA was around 4.4 times as fast.

Using the above formula, without the RAA around 77% of the time was spent in unsuccessful clause matching. The SQL and BASIC tests suggest that worthwhile time savings result from the use of the RAA. They also suggest that as the size of the DCG and hence the size of the parse table increase so do the gains from using the RAA. It would be interesting to make the same comparisons with substantially larger grammars but the parser generator implementation did not permit this.

LR and direct DCG parsing

The BASIC DCG was also tested as a direct translation to UNSW Prolog using SICStus Prolog as a translator. This generated Prolog with explicit `connects` clauses as described in the first section of the paper. On the same test set as above, the times in seconds were:

symbols	DCG	RAA	no RAA
58	1.07	1.67	7.37
95	6.78	2.87	12.34
146	7.80	4.41	19.26
179	49.44	5.26	23.16

These times illustrate the apparently erratic behaviour of top down parsing. The first time is better than LR bottom up with the RAA. The second and third times are better than LR without the RAA and worse than LR with the RAA. The last time is worse than LR both with and without the RAA.

As discussed above, DCGs are normally translated directly into Prolog clauses with one clause for each DCG production. Parsing then involves the normal Prolog search mechanism with full backtracking. This might be expected to be faster than the LR technique as the latter is based on an explicit interpretation mechanism sitting on top of the underlying Prolog mechanism.

Furthermore, the DCG consists of a relatively small number of clauses with a relatively large number of distinct principle functors and a relatively small number of options for each functor. The LR approach is based on a table which has more clauses than those for the equivalent DCG. The table has one common principle functor and a relatively large number of options. Thus, the clause

indexing mechanism will affect their relative speeds.

UNSW Prolog uses principal functor clause indexing and then searches for appropriate options within clauses. Thus, the short time taken to locate the unique table functor for the LR approach must be offset against the far longer time spent searching for the correct option corresponding to the current state and symbol.

More recent Prologs, like NU, Quintus and SICStus, use first argument indexing. With the LR approach this would identify all the clauses for the appropriate state within the table but still necessitate searching for the clause for the current symbol. Thus, the timings for direct DCG parsing and LR parsing without the RAA might be more similar than with principal functor indexing. The use of the RAA should still be superior, particularly when there are large numbers of terminal symbols relative to the number of productions, as it removes all option searching.

However, the direct DCG approach can also be far slower, as the last test example above shows. LR parsing is always in linear time, proportional to the number of symbols in the sentence, and it involves no backtracking. Direct DCG parsing without backtracking may be faster than LR parsing but that depends on the sentence being parsed. The order in which DCG productions are attempted is fixed by the underlying Prolog search mechanism and so backtracking will often occur where an optimal method like LR will avoid it. Full DCGs are, of course, more powerful than LR grammars and so expressive strength must be offset against efficiency.

Future work

The present system is experimental: it does not analyse grammars to ensure that they are SLR(1) and parse tables are not factored to reduce entries.

A fundamental limitation to the investigation of this approach to parsing is the small size of the DCG which can be handled by the current Prolog system. This might be overcome by building a C version which uses iteration on arrays to build LR parse tables. This would enable the testing of LR parser implementations on the RAA with much larger DCGs. It would also be interesting to investigate the use of LR parsing with a RAA interface to an optimized Prolog like Quintus, SICStus or NU.

In the longer term, it might be appropriate to investigate at the implementation of parse tables from the YACC LR parser generator on the RAA. This would enable its evaluation with a large body of existing software developed for UNIX use. For Prolog work, DCGs might then be translated to YACC grammars.

Acknowledgements

This work was carried out at the CSIRO Division of Information Technology in Sydney, Australia. I would like to thank Charles Chung for his help with the implementation of LR parse tables on the RAA, in particular for his development of a field encoding for table entries. I would also like to thank Bob Colomb for his advice and support throughout this project.

References

1 Pereira, F C N and Warren, D H D 'Definite Clause Grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks' *Artificial Intelligence* Vol 13 (1980) pp 231–278
2 Aho, A V, Sethi, R and Ullman, J D *Compilers: principles, techniques and tools* Addison-Wesley (1977)
3 Johnson, S 'YACC: yet another compiler compiler' *CSTR* 32, Bell Laboratories (1975)
4 Nilsson, U 'AID: an alternative implementation of DCGs' *New Generation Computing* Vol 4 No 4 (1986) pp 383–399
5 Morley, D 'Efficient parsing using constraints' in *Proc. 12th Australian Computer Science Conference* Vol 11 Australian Computer Science Communications (1989) pp 37–47
6 Colomb, R M 'Table searching using a content addressable memory' *Australian Computer J* Vol 20 No 3 (August 1988) pp 105–112
7 Colomb, R M 'Enhanced unification in Prolog through clause indexing' *Journal of Logic Programming* Vol 10 No 1 (1991) pp 23–44
8 Chung, C 'Integration of Relational Algebra Accelerator with UNSW Prolog' *TR-FB-88-04* CSIRO Division of IT (February 1988)
9 Sammut, C *UNSW Prolog Reference Manual* University of NSW
10 Naish, L *MU-Prolog 3.1db Reference Manual* Melbourne University (May 1984)
11 Carlsson, M and Widen, J 'SICStus Prolog User's Manual' *SICS R88007B* Swedish Institute of Computer Science (October 1988)
12 Clocksin, W F and Mellish, C S *Programming in Prolog* Springer-Verlag (1987)
13 Herjee, K B and Sadeghi, R 'Rapid implementation of SQL: a case study using YACC & LEX' *Inf. and Soft. Technol.* Vol 30 No 4 (May 1988) pp 228–236

Appendix A
Direct encoding of parse table

```
t(0,[0],shift(3)).
t(0,[1],shift(4)).
t(0,digit,goto(2)).
t(0,binary,goto(1)).

t(1,[rstop],oh).

t(2,[0],shift(3)).
t(2,[1],shift(4)).
t(2,[rstop],reduce(1,binary)).
t(2,digit,goto(2)).
t(2,binary,goto(5)).

t(3,[0],reduce(1,digit)).
t(3,[1],reduce(1,digit)).
t(3,[rstop],reduce(1,digit)).

t(4,[0],reduce(1,digit)).
t(4,[1],reduce(1,digit)).
t(4,[rstop],reduce(1,digit)).

t(5,[rstop],reduce(2,binary)).
```

Appendix B
Top level parser

```
exp(E,T,R) :- sr([0],E,T,R).

% sr(<stack>,<lexemes>, <result>,<extras>).

sr([S | SY],[H | T],TR,R) :-
  t(S,H,reduce(N,P)),                    % reduce action?
  pop(N,[S | SY],[NS | NSY]),            % pop RHS
  t(NS,P,NNS),                           % find goto action
  sr([NNS,P,NS | NSY],[H | T],TR,R).     % push LHS & next
                                         % state & try
                                         % again

pop(0,S,S).
pop(N,[S,SY | T],R) :- N1 is N-1, pop(N1,T,R).

sr([S | SY],[H | T],TR,R) :-
  t(S,H,shift(V)),                       % shift action?
  sr([V,H,S | SY],T,TR,R).               % push symbol H &
                                         % enter state V

sr([S |_],[H | T],ok,T) :- t(S,H,h).     % success? check
                                         % for halt state

sr(_,E,fail,E).                          % fail so return
                                         % symbols
                                         % at failure point
```

Appendix C
Parse table with reduce actions through unification

```
t([0 | SY],[[0] | T],TR,R) :- t([3,[0],0 | SY],T,TR,R).
t([0 | SY],[[1] | T],TR,R) :- t([4,[1],0 | SY],T,TR,R).
goto(0,digit,2).
goto(0,binary,1).

t([1,binary | _],[[rstop]],binary,T).

t([2 | SY],[[0] | T],TR,R) :- t([3,[0],2 | SY],T,TR,R).
t([2 | SY],[[1] | T],TR,R) :- t([4,[1],2 | SY],T,TR,R).
t([2,digit,NS | NSY],[[rstop]],TR,R) :-
  goto(NS,binary,NNS),
  t([NNS,binary,NS | NSY],[[rstop]],TR,R).
goto(2,digit,2).
goto(2,binary,5).

t([3,[0],NS | NSY],[[0] | T],TR,R) :-
  goto(NS,digit,NNS),
  t([NNS,digit,NS | NSY],[[0] | T],TR,R).
t([3,[0],NS | NSY],[[1] | T],TR,R) :-
  goto(NS,digit,NNS),
  t([NNS,digit,NS | NSY],[[1] | T],TR,R).
t([3,[0],NS | NSY],[[rstop]],TR,R) :-
  goto(NS,digit,NNS),
  t([NNS,digit,NS | NSY],[[rstop]],TR,R).

t([4,[1],NS | NSY],[[0] | T],TR,R) :-
  goto(NS,digit,NNS),
  t([NNS,digit,NS | NSY],[[0] | T],TR,R).
t([4,[1],NS | NSY],[[1] | T],TR,R) :-
  goto(NS,digit,NNS),
  t([NNS,digit,NS | NSY],[[1] | T],TR,R).
t([4,[1],NS | NSY],[[rstop]],TR,R) :-
  goto(NS,digit,NNS),
  t([NNS,digit,NS | NSY],[[rstop]],TR,R).

t([5,binary,_,digit,,NS | NSY],[[rstop]],TR,R) :-
  goto(NS,binary,NNS),
  t([NNS,binary,NS | NSY],[[rstop]],TR,R).

t(_,E,fail,E).

exp(E,T,R) :- t([0],E,T,R).
```

Appendix D
Parse table with explicit recursion

```
t([0 | SY],[[0] | T],TR,R,rt([3,[0],0 | SY],T,TR,R)).
t([0 | SY],[[1] | T],TR,R),rt([4,[1],0 | SY],T,TR,R)).
goto(0,digit,2).
goto(0,binary,1).

t([1,binary | _],[[rstop]],binary,T,true).

t([2 | SY],[[0] | T],TR,R),rt([3,[0],2 | SY],T,TR,R)).
t([2 | SY],[[1] | T],TR,R,rt([4,[1],2 | SY],T,TR,R)).
```

```
t([2,digit,NS | NSY],[[rstop]],TR,R,
   (goto(NS,binary,NNS),
   rt([NNS,binary,NS | NSY],[[rstop]],TR,R)).
goto(2,digit,2).
goto(2,binary,5).

t([3,[0],NS | NSY],[[0] | T],TR,R,
   (goto(NS,digit,NNS),
   rt([NNS,digit,NS | NSY],[[0] | T],TR,R)).
t([3,[0],NS | NSY],[[1] | T],TR,R,
   (goto(NS,digit,NNS),
   rt([NNS,digit,NS | NSY],[[1] | T],TR,R)).
t([3,[0],NS | NSY],[[rstop]],TR,R,
   (goto(NS,digit,NNS),
   rt([NNS,digit,NS | NSY],[[rstop]],TR,R)).

t([4,[1],NS | NSY],[[0] | T],TR,R,
   (goto(NS,digit,NNS),
   rt([NNS,digit,NS | NSY],[[0] | T],TR,R)).
t([4,[1],NS | NSY],[[1] | T],TR,R,
   (goto(NS,digit,NNS),
   rt([NNS,digit,NS | NSY],[[1] | T],TR,R)).
t([4,[1],NS | NSY],[[rstop]],TR,R,
   (goto(NS,digit,NNS),
   rt([NNS,digit,NS | NSY],[[rstop]],TR,R)).

t([5,binary,_,digit,NS | NSY],[[rstop]],TR,R,
   (goto(NS,binary,NNS),
   rt([NNS,binary,NS | NSY],[[rstop]],TR,R)).

t(_,E,fail,E,true).
```