LR Parsing of CFG's with Restrictions

B. J. McKenzie

University of Canterbury, Christchurch, New Zealand

Postal Address:

Dr. B.J.McKenzie,
Department of Computer Science,
University of Canterbury,
Private Bag, Christchurch,
New Zealand.

Phone:	+64 3 642-349
Fax:	+64 3 642-999
Telex:	UNICANT NZ4144

Electronic mail address (internet):

bruce@cosc.canterbury.ac.nz

LR Parsing of CFG's with Restrictions

B. J. McKenzie

University of Canterbury, Christchurch, New Zealand

SUMMARY

A method for extending the LR parsing method to enable it to deal with context free grammars containing imbedded restrictions is presented. Such restrictions are usually dealt with in LR based parsers by executing semantic code outside the context of the LR method. By including such restrictions within the LR method itself, potential shift-reduce and reduce-reduce conflicts can be resolved and provide greater control over the language accepted. The proposed method can be easily incorporated into existing LR based parser generating systems.

KEY WORDS: LR Parsing, semantic restrictions, compilers

INTRODUCTION

Often when a language is specified by a context free grammar (CFG) additional restrictions are placed on the grammar rules to further constrain the strings accepted by the grammar. These are often referred to as semantic restrictions because they are outside the purely syntactic restrictions specified by the grammar. The following concrete examples are representative of the form of such restrictions. For each example a (very) simplified grammar is exhibited which retains the kernal of the problem; these grammars will be used as running examples to illustrate the new technique.

Author's Address: Department of Computer Science, University of Canterbury, Private Bag, Christchurch, New Zealand.

a) Rules where earlier parts of the input language determine which of a number of alternative interpretations of later constructs should be made. Thus the initial part of the input might specify whether coordinates appearing later should be interpreted as two-dimensional pairs of integers or three-dimensional triples.

Grammar A:

input	→	format data
format	→	**integer**
data	→	data point
data	→	∈
point	→	**integer integer**
point	→	**integer integer integer**

Restrictions: The value of the format integer is either 2 or 3 and specifies the number of integers used for a point.

b) Checks such as declaration before use or agreement in the number of parameters in a procedure declaration with the number of expressions provided in a procedure invocation [1].

Grammar B:

input	→	proc_decl proc_use
proc_decl	→	**procedure identifier (identifier** id_list **)**
id_list	→	**, identifier** id_list
id_list	→	∈
proc_use	→	**identifier (expression** exp_list **)**
exp_list	→	**, expression** exp_list
exp_list	→	∈

Restrictions: The number of **identifier**s in parenthesis in the proc_decl equals the number of **expression**s in parenthesis in the proc_use

c) Use of a highly ambiguous CFG for expressions but using the priority and associativity of the operators to resolve the ambiguity inherent in the grammar.

Grammar C:

input	→	E
E	→	E **+** E
E	→	E * E
E	→	E ^ E
E	→	**integer**

Restrictions: The priority and associativity of the operators is as follows:

operator	meaning	priority	associativity
+	addition	1	left
*	multiplication	2	left
^	exponentiation	3	right

d) Use of CFG rules in machine descriptions in code generators using the Graham-Glanville [2] method.

Grammar D:

input	→	reg		
reg	→	**+** reg **const**	"incl reg"	[const==1]
reg	→	**+** reg **const**	"add const,reg"	[const≠1]
reg	→	**const**	"clr reg"	[const==0]
reg	→	**const**	"movl const, reg"	[const≠0]

Restrictions: The valid values of the const in the productions are specified by the expressions in brackets. These are known as semantic qualifications in the context of the Graham-Glanville method.

Existing LR-based parser generator systems [5,6] are normally unable to deal with grammars such as A but can deal with examples of the other forms by a variety of methods.

Grammars such as B can be handled by building the LR parsing tables ignoring the restrictions. Code can then be attached to each production and be executed just before the parser makes a reduction with this production. This code can then test the restriction and issue an error or warning message about the mis-match in the declaration and use of the production.

Grammars such a C can be dealt with by the parser generator as the parsing tables are constructed. The ambiguities in the grammar manifest themselves as shift-reduce conflicts in the parsing DFA that can not be resolved by LALR(1) lookahead. Each production is assigned the priority and associativity of the last terminal symbol in its right hand side. The shift/reduce conflicts are then resolved in favour of the shift or production with the highest priority. If the priorities are identical then the associativity is used by resolving in favour of the shift for right associative operators and the reduce for left associative operators.

Grammars such as D are dealt with using a maximal-munch approach [2-4]. Shift/reduce conflicts are resolved in favour of shifts and reduce/reduce conflicts in favour of the longest production to encourage the consumption of as many terminals as possible at each reduction. The restrictions are tested before the reductions are made and used to choose between a number of similar productions. Care is taken in such systems to ensure that the parse is never blocked by the failure of a restriction.

In this paper the restrictions will be included as an integral part of the grammar while the parsing DFA is constructed. Each of the grammar types discussed above can be handled by this method including those of type A.

HANDLING EMBEDDED RESTRICTIONS

LR based parser generator systems usually allow code to be attached to a production that is executed just before the production is reduced. In *yacc* and *bison* for example, such code consists of code written in the C language enclosed in braces {...}. At parse time a value stack is maintained in parallel with the state stack to store a value associated with each terminal (returned by the scanner) and non-terminal (calculated by actions) symbol. Each occurrence of $1, $2, ... $9 in an action retrieves the value of the 1st, 2nd, ... 9th symbol on the right hand side of the production from the value stack.

To allow code to be attached to an earlier position of the production and hence be executed after only part of the production has been parsed the use of marker non-terminals is employed. For example given a production with embedded code such as:

$$A \rightarrow X_1 \;\; X_2 \; \{ \text{ code } \} \; X_3 \;\; X_4$$

a dummy marker non-terminal, M say, with an empty right hand side is introduced and the code attached to the end of this new production [1].

$$A \rightarrow X_1 \;\; X_2 \; M \; X_3 \;\; X_4$$

$$M \rightarrow \in \; \{ \text{ code } \}$$

The addition of marker non-terminals to an LR grammar can introduce parsing conflicts [7]. For example, the grammar:

$$L \rightarrow L \; b \quad | \quad a$$

is LR(1) but the grammar:

$$L \rightarrow M \; L \; b \quad | \quad a$$

$$M \rightarrow \in$$

is not LR(1).

The proposed method for dealing with embedded restrictions in CFG uses a similar technique. Each test in the production, regardless of where it appears is replaced by a marker non-terminal. A new null production for this marker non-terminal is then added to the grammar and the test associated with it. To illustrate the approach let us add restrictions to grammar A we introduced earlier. These will use an extension of the notation used for actions in *yacc* and *bison* discussed earlier; they consist of C expressions within parenthesis. Furthermore we have included variable declarations within %{ .... %}.

Grammar A':

```
%{  int  dimen; %}
input     →    format  data
format    →    integer  { dimen=$1; }
data      →    data  point
          |    ∈
point     →    (dimen==2) integer  integer
          |    (dimen==3) integer  integer  integer
```

The value of the format integer (2 or 3) is saved in the declared variable `dimen` and used in the tests controlling the two possible productions for point. Replacing these restrictions with marker terminals:

```
point     →    T1  integer  integer
          |    T2  integer  integer  integer
T1        →    ∈ (dimen==2)
T2        →    ∈ (dimen==3)
```

we construct the LR parsing DFA for the resulting grammar using standard methods [1]. The calculation of the lookahead sets proceeds as if these additional productions were a normal part of the grammar so any existing methods such as DeRemer and Pennello's [8] for LALR(1) grammars can be used. This results in the DFA of figure 1.

The restriction code is associated with the reductions to the marker non-terminals in whatever states they appear. These are treated in a special manner when they are involved in shift/reduce or reduce/reduce conflicts; in particular such conflicts are ignored when the parse tables are constructed and the conflicts resolved during parsing as follows. When the parser enters a state containing one or more restriction reduction, the restriction code for all reductions that are valid for the current lookahead symbol is executed. If a single restriction succeeds then its corresponding reduction should be made which (being a null production) involves a simple "goto" to the new state (which is then pushed on the state stack).

If all restrictions fail the parsing should continue as if the restriction productions were not there. If this results in a parsing error then our implementation treats this using standard error reporting and recovery techniques. [5]. A more restrictive approach is typically taken by the code generation parsers [2-4] were such a condition is referred to as semantic blocking. This situation is avoided by checking when the parser is constructed that a default production has been supplied that contains no restrictions and hence can never fail. This approach is appropriate in a code generator where such errors correspond to failure to generate code from valid intermediate code but is too restrictive for a more general parser.

If more than one restriction succeeds then this should be regarded as a new form of conflict, a test/test conflict. By choosing the restrictions carefully the user can ensure such conflicts never occurred but it would be impossible to automatically test that such a condition was satisfied without severely limiting the form that restrictions could take. One approach would be to treat such conflicts as a parser error to be reported and recovered from. Alternatively these can be resolved in some manner when the parser is constructed. In our implementation the parser uses the first restriction that succeeds and so the user has a measure of control by ordering the productions in the grammar.

Consider the parse of a string using a parser for Grammar A' based on the parsing DFA exhibited in figure 1. When the parser enters state 3 the global variable `dimen` will contain the value of the first integer seen which determines whether the points are to be interpreted as pairs or triples of integers. If the lookahead symbol is integer then the tests associated with the marker non-terminals T_1 and T_2 should be tested. If the test for T_1, `(dimen==2)`, succeeds the reduction "T_1 → ∈" and a transition to state 5 is made with a subsequent shifting of two integers before reaching state 9 and reducing **two** integers to a point. Alternatively if the test for T_2, `(dimen==3)`, succeeds the reduction "T_2 → ∈" and a transition to state 6 is made with a subsequent shifting of three integers before reaching state 11 and reducing **three** integers to a point. If both tests fail then an error would be detected as there is no other shift or reduction possible when the next symbol is integer. If the lookahead symbol had been eof then the tests and their associated reductions would have been ignored and reduction "input → format data" made.

The other three examples introduced earlier can similarly be handled by adding restrictions to the grammars as shown:

Grammar B':

```
%{  int N, M;  %}
input        →     proc_decl  proc_use
proc_decl    →     procedure  identifier  (  identifier { N=1; }  id_list   )
id_list      →     ,  identifier  { N++; }  id_list
             |     ∈
proc_use     →     identifier  (  expression   { M=1; } exp_list   )
exp_list     →     (M < N)  ,  expression   { M++; } exp_list
             |     (M == N)  ∈
```

Here the number of identifiers in the declaration are counted with the global variable N. As the expressions in the exp_list are encountered they are counted using the global variable M and the tests (M<N) and (M==N) ensure that exactly the same number of expressions are expected before the final ")".

Grammar C':

```
input    :    E ;
E        :    E  +  E ( pri(+)>=pri(yychar)) ;
E        :    E  *  E ( pri(*)>=pri(yychar)) ;
E        :    E  ^  E ( pri(^)>  pri(yychar)) ;
E        :    integer  ;
```

This example uses the value of `yychar` the lookahead symbol, and the function `pri(sym)` which returns 0 if `sym` is not an operator or the priority of `sym` otherwise. By comparing the priority of the lookahead symbol with that of the operator in the production it is possible to explicitly control whether shifting or reduction should be applied to obtain the correct interpretation of the priority and associativity (>= for left and > for right) of the operators.

For example if the input *integer * integer* has already been parsed then the parser will be in state 7 of the DFA of figure 2. If the lookahead symbol is '+', '*', '^' or eof then the test T_2: `(pri(*)>=pri(yychar))` will be executed. If it succeeds then the reduction "$T_2 \rightarrow \in$" and a transition to state 10 would result with a subsequent reduction "E→E∗E" made. Given the priorities of the operators this would only happen if the next symbol was '+' or '*'. For the lookahead symbols of '^' or eof the test would fail and the restriction productions would be ignored. This would result in a shift to state 3 on '^' or the reduction "E→E∗E" on eof. These are exactly the actions that are required for the correct interpretation of these operators.

Grammar D':

```
input    :    reg
reg      :    + reg  const  ($3==1)     { out("incl   reg"); }
         |    + reg  const  ($3!=1)     { out("add   const,reg"); }
         |    const  ($1==0)            { out("clr    reg"); }
         |    const  ($1!=0)            { out("movl   const, reg"); }
```

The use of the restrictions in this case is straightforward.

IMPLEMENTATION

The alteration of existing LR based parser generator systems to accommodate restrictions can be easily achieved. A version of *bison* [6] incorporating restrictions was developed by making the following changes. The restriction code can be replaced by a null production marker non-terminal while the grammar is read as is done with embedded action code. Such non-terminals and productions need not be distinguished from the user defined non-terminals and productions during the construction of the parsing DFA or the calculation of the lookahead-sets. They result in extra items in the states formed during construction of the parsing DFA that may result in states being split relative to those obtained without the restrictions. For example in figure 1 the items of states 7 and 8 would have been combined into a single state without the restrictions. However the restriction productions must be distinguished from user productions during the detection and resolution of shift/reduce and reduce/reduce conflicts. Conflicts involving such conflict productions should not be reported as conflicts as these will be resolved at parse-time. Furthermore the user should be warned about potential test/test conflicts that are detected.

The storage method used for the parsing tables needs to be altered to include the execution and testing of restriction code. A popular method for storing the DFA transition table (used by both *bison* and *yacc*) is row displacement encoding [1,5] where the current state and lookahead symbol are used to access a table with shifts to state 's' encoded as a positive '+s' and reduction by production 'r' encoded as a negative '-r'. When restrictions are added each such entry may need to encode a list of restrictions that should be tested first. In our implementation these were encoded as positive values greater than the largest possible shift and this value was used to index into a table containing the tests to be evaluated followed by the value (positive or negative) that would have been in the table in the absence of the tests and should be used if all the tests fail. Figure 3 outlines the main features of the parsing routine in psuedo-code. The suggested implementation results in only a marginal speed penalty compared to a parser without tests; the only cost is a check for the existence of a test state on each move of the parser. There is also a small increase in the memory requirements for the tables as a result of the extra dummy non-terminals and empty productions. Space is also required to store the lists of tests to be executed for test states. In the worst case were there are *T* distinct restrictions in the grammar and *N* states in the parsing DFA containing restriction reductions, then *2*N + T*N* small integers need be stored. In the best case were each restriction appears in only one state this would be reduced to *2*N + T*. In grammar C' for example we find (figure 2) that T=3 and N=3 and the best case of 9 applies while for grammar D' we have T=4 and N=2, again with a best case of 8.

Conclusions

A method has been presented for incorporating tests into a CFG to control semantic restrictions during parsing using LR techniques. We have presented a number of example uses of the method and outlined how existing LR based parser generator systems can be altered to include such a facility. Our approach shares a number of similarities with the method of Ganapathi [3,4] were code generation is controlled using affix grammars and predicates. The current proposal however is not limited to code generation applications but can be used in general LALR(1) and LR(1) parser generator systems. Furthermore the form of the restrictions (predicates) allowed by our approach is more general than those considered by Ganapathi.

Acknowledments

I thank Tim Bell for useful comments on this paper.

REFERENCES

1. AHO, A.V., SETHI, R. and ULLMAN, J.D.. *Compilers: Practical techniques and tools.* Addison Wesley Reading Mass. 1986

2. GLANVILLE, R.S., GRAHAM, S.L., *A new method for compiler code generation*, Fifth ACM symposium on Principles of Programming Languages, 231-240, 1978

3. GANAPATHI, M., *Retargetable Code Generation and Optimization using Attribute Grammars*, Ph.D dissertation, Computer Science Department, University of Wisconsin-Madison, 1980

4. GANAPATHI, M., FISCHER, C.N., *Affix Grammar Driven Code Generation*, ACM Transactions on Programming Languages and Systems, **7**, 560-599, 1985

5. JOHNSTON, S. C., *Yacc - yet another compiler compiler* Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J. 1975

6. STALLMAN, R. M., *Bison* Free Software Foundation Inc. Report, 1986

7. PURDON, P. and BROWN, C.A. *Semantic routines and LR(k) parsers* Acta Informatica **14**, 299-315, 1980

8. DEREMER, F. and PENNELLO, T. *Efficient Computation of LALR(1) Look-Ahead Sets* ACM TOPLAS **4**, 615-649, 1982