Solving Difficult LR Parsing Conflicts by Postponing Them

L. Garcia-Forte and C. Rodriguez-Leon

Departamento de EIO y Computación,
Universidad de La Laguna, Tenerife, Spain
`casiano@ull.es`,
WWW home page: `http://nereida.deioc.ull.es`

Abstract. Though yacc-like LR parser generators provide ways to solve shift-reduce conflicts using token precedences, no mechanisms are provided for the resolution of reduce-reduce conflicts. To solve this kind of conflicts the language designer has to modify the grammar. All the solutions for dealing with these difficult conflicts branch at each alternative, leading to the exploration of the whole search tree. These strategies differ in the way the tree is explored: GLR, Backtracking LR, Backtracking LR with priorities, etc. This paper explores an entirely different path: to extend the yacc conflict resolution sublanguage with new constructs allowing the programmers to explicit the way the conflict must be solved. These extensions supply ways to resolve any kind of conflicts, including those that can not be solved using static precedences. The method makes also feasible the parsing of grammars whose ambiguity must be solved in terms of the semantic context. Besides, it brings to LR-parsing a common LL-parsing feature: the advantage of providing full control over the specific trees the user wants to build.

1 Introduction

Yacc-like LR parser generators [1] provide ways to solve shift-reduce mechanisms based on token precedence. No mechanisms are provided for the resolution of reduce-reduce conflicts or difficult shift-reduce conflicts. To solve such kind of conflicts the language designer has to modify the grammar. Quoting Merrill [2]:

> *Yacc lacks support for resolving ambiguities in the language for which it is attempting to generate a parser. It does a simple-minded approach to resolving shift/reduce and reduce/reduce conflicts, but this is not of sufficient power to solve the really thorny problems encountered in a genuinely ambiguous language*

Some context-dependency ambiguities can be solved through the use of lexical tie-ins: a flag which is set by the semantic actions, whose purpose is to alter the way tokens are parsed. But it is not always possible or easy to resort to this kind of tricks to fix some context dependent ambiguity. A more general solution is to extend LR parsers with the capacity to branch at any multivalued entry of the

LR action table. For example, Bison [7], via the `%glr-parser directive` and Elkhound [5] provide implementations of the Generalized LR (GLR) algorithm [4]. In the GLR algorithm, when a conflicting transition is encountered, the parsing stack is forked into as many parallel parsing stacks as conflicting actions. The next input token is read and used to determine the next transitions for each of the top states. If some top state does not transit for the input token it means that path is invalid and that branch can be discarded. Though GLR has been successfully applied to the parsing of ambiguous languages, the handling of languages that are both context-dependent and ambiguous is more difficult. The Bison manual [7] points out the following caveats when using GLR:

> *...there are at least two potential problems to beware. First, always analyze the conflicts reported by Bison to make sure that GLR splitting is only done where it is intended. A GLR parser splitting inadvertently may cause problems less obvious than an LALR parser statically choosing the wrong alternative in a conflict. Second, consider interactions with the lexer with great care. Since a split parser consumes tokens without performing any actions during the split, the lexer cannot obtain information via parser actions. Some cases of lexer interactions can be eliminated by using GLR to shift the complications from the lexer to the parser. You must check the remaining cases for correctness.*

The strategy presented here extends yacc conflict resolution mechanisms with new ones, supplying ways to resolve conflicts that can not be solved using static precedences. The algorithm for the generation of the LR tables remains unchanged, but the programmer can modify the parsing tables during run time.

The technique involves labelling the points in conflict in the grammar specification and providing additional code to resolve the conflict when it arises. Crucially, this does not requires rewriting or transforming the grammar, trying to resolve the conflict in advance, backtracking or branching into concurrent speculative parsers. Instead, the resolution is postponed until the conflict actually arises during parsing, whereupon user code inspects the state of the underlying parse engine to decide the appropriate solution. There are two main benefits: Since the full power of the native universal hosting language is at disposal, any grammar ambiguity can be tackled. We can also expect - since the conflict handler is written by the programmer - a more efficient solution which reduces the required amount of backtracking or branching.

This technique can be combined to complement both GLR and backtracking LR algorithms [6] to give the programmer a finer control of the branching process. It puts the user - as it occurs in top down parsing - in control of the parsing strategy when the grammar is ambiguous, making it easier to deal with efficiency and context dependency issues. One disadvantage is that it requires a comprehensive knowledge of LR parsing. It is conceived to be used when none of the available techniques - static precedences, grammar modification, backtracking LR or Generalized LR - produces satisfactory solutions. We have implemented these techniques in `Parse::Eyapp` [9], a yacc-like LALR parser generator for

Perl [10, 11]. The Perl language is, quoting Paul Hudak's article [12] a "*domain specific language for text manipulation*".

This paper is divided in six sections. The next section introduces the Postponed Conflict Resolution (PPCR) strategy. The following three sections illustrate the way the technique is used. The first presents an ambiguous grammar where the disambiguating rule is made in terms of the previous context. The next shows the technique on a difficult grammar that has been previously used in the literature [7] to illustrate the advantages of the GLR engine: the declaration of enumerated and subrange types in Pascal [13]. The last example deals with a grammar that can not be parsed by any LL(k) nor LR(k), whatever the value of `k`, nor for packrat parsing algorithms [14]. The last section summarizes the advantages and disadvantages of our proposal.

2 The *Postponed Conflict Resolution* Strategy

The *Postponed Conflict Resolution* is a strategy (PPCR strategy) to apply whenever there is a shift-reduce or reduce-reduce conflict which is unsolvable using static precedences. It delays the decision, whether to shift or reduce and by which production to reduce, to parsing time. Let us assume the `eyapp` compiler announces the presence of a reduce-reduce conflict. The steps followed to solve a reduce-reduce conflict using the PPCR strategy are:

1. Identify the conflict: What LR(0)-items/productions and tokens are involved?.
   Tools must support that stage, as for example via the `.output` file generated by `eyapp`. Suppose we identify that the participants are the two LR(0)-items $A \rightarrow \alpha_\uparrow$ and $B \rightarrow \beta_\uparrow$ when the lookahead token is `@`.
2. The software must allow the use of symbolic labels to refer by name to the productions involved in the conflict. Let us assume that production $A \rightarrow \alpha$ has label `:rA` and production $B \rightarrow \beta$ has label `:rB`. A difference with `yacc` is that in `Parse::Eyapp` productions can have *names* and *labels*. In `Eyapp` names and labels can be explicitly given using the directive `%name`, using a syntax similar to this one:

   `%name :rA` $A \rightarrow \alpha$
   `%name :rB` $B \rightarrow \beta$

3. Give a symbolic name to the conflict. In this case we choose `isAorB` as name of the conflict.
4. Inside the *body* section of the grammar, mark the points of conflict using the new reserved word `%PREC` followed by the conflict name:

   `%name :rA` $A \rightarrow \alpha$ `%PREC IsAorB`
   `%name :rA` $B \rightarrow \beta$ `%PREC IsAorB`

5. Introduce a `%conflict` directive inside the *head* section of the translation scheme to specify the way the conflict will be solved. The directive is followed by some code - known as the *conflict handler* - whose mission is to modify the parsing tables. This code will be executed each time the associated conflict state is reached. This is the usual layout of the conflict handler:

```
%conflict  IsAorB {
  if (is_A) { $self->YYSetReduce('@', ':rA' ); }
       else { $self->YYSetReduce('@', ':rB' ); }
}
```

   Inside a conflict code handler the Perl default variable `$_` refers to the input and `$self` refers to the parser object.
   Variables in Perl - like `$self` - have prefixes like `$` (scalars), `@` (lists), `%` (hashes or dictionaries), `&` (subroutines), etc. specifying the type of the variable. These prefixes are called *sigils*. The sigil `$` indicates a *scalar* variable, i.e. a variable that stores a single value: a number, a string or a reference. In this case `$self` is a reference to the parser object. The arrow syntax `$object->method()` is used to call a method: it is the equivalent of the dot operator `object.method()` used in most OOP languages. Thus the call

```
$self->YYSetReduce('@', ':rA' )
```

   is a call to the `YYSetReduce` method of the object `$self`.
   The method `YYSetReduce` provided by `Parse::Eyapp` receives a token, like `'@'`, and a production label, like `:rA`. The call

```
$self->YYSetReduce('@', ':rA' )
```

   sets the parsing action for the state associated with the conflict `IsAorB` to reduce by the production `:rA` when the current lookahead is `@`.
   The call to `is_A` represents the context-dependent dynamic knowledge that allows us to take the right decision. It is usually a call to a nested parser for A but it can also be any other contextual information we have to determine which one is the right production.

The procedure is similar for shift-reduce conflicts. Let us assume we have identified a shift-reduce conflict between LR-(0) items $A \rightarrow \alpha_\uparrow$ and $B \rightarrow \beta \uparrow \gamma$ for some token `'@'`. Only steps 4 and 5 change slightly:

4'. Again, we must give a symbolic name to $A \rightarrow \alpha$ and mark with the new `%PREC` directive the places where the conflict occurs:

   `%name :rA` $A \rightarrow \alpha$ `%PREC IsAorB`
   $B \rightarrow \beta$ `%PREC IsAorB` γ

5'. Now the conflict handler calls the `YYSetShift` method to set the `shift` action:

```
%conflict  IsAorB {
      if (is_A) { $self->YYSetReduce('@', ':rA' ); }
      else { $self->YYSetShift('@'); }
   }
```

3 A Simple Example

The following example[1] accepts lists of two kind of commands: *arithmetic expressions* like `4-2-1` or one of two *associativity commands*: `left` or `right`. When a `right` command is issued, the semantic of the `'-'` operator is changed to be right associative. When a `left` command is issued the semantic for `'-'` returns to its classic left associative interpretation. Here follows an example of input. Between shell-like comments appears the expected output:

```
$ cat input_for_dynamicgrammar.txt
2-1-1 # left:  0 = (2-1)-1
RIGHT
2-1-1 # right: 2 = 2-(1-1)
LEFT
3-1-1 # left:  1 = (3-1)-1
RIGHT
3-1-1 # right: 3 = 3-(1-1)
```

We use a variable `$reduce` (initially set to `1`) to decide the way in which the ambiguity `NUM-NUM-NUM` is solved. If `false` we will set the `NUM-(NUM-NUM)` interpretation. The variable `$reduce` is modified each time the input program emits a `LEFT` or `RIGHT` command.

Following the steps outlined above, and after looking at the `.output` file, we see that the items involved in the announced shift-reduce conflict are

$$expr \to expr_{\uparrow} - expr$$
$$expr \to expr - expr_{\uparrow}$$

and the lookahead token is `'-'`. We next mark the points in conflict in the grammar using the `%PREC` directive (see Figure 1)

[1] For the full examples used in this paper, see the directory `examples/debuggingtut/` in the `Parse::Eyapp` distribution [9]

```
%%
p:
      /* empty */     {}
    | p c             {}
;

c:
      $expr { print "$expr\n" }
    | RIGHT { $reduce = 0}
    | LEFT  { $reduce = 1}
;
```

```
expr:
      '(' $expr ')'  { $expr }
    | %name :M
      expr.left         %PREC lOr
        '-' expr.right  %PREC lOr
         { $left -$right }

    | NUM
;
```

Fig. 1. An Example of Context Dependent Ambiguity Resolution

The *dollar* and *dot* notation used in some right hand sides (rhs) like in `expr.left '-' expr.right` and `$expr` is used to associate variable names with the attributes of the symbols.

The conflict handler `lOr` defined in the header section is:

```
%conflict lOr {
  if ($reduce) {$self->YYSetReduce('-', ':M')}
  else         {$self->YYSetShift('-')}
}
```

If `$reduce` is `true` we set the parsing action to *reduce* by the production labelled `:M`, otherwise we choose the *shift action.*

Observe how PPCR allow us to dynamically change at will the meaning of the same statement. That is certainly harder to do using alternative techniques, either problem specific, like *lexical Tie-Ins* [7], or more general, like GLR [4].

4 Nested Parsing of Unexpended Input and Context

This section illustrates the technique through a problem that arises in the declaration of enumerated and subrange types in the programming language Pascal. The problem is taken from the Bison manual, (see section '*Using GLR on Unambiguous Grammars*') where it is used as a paradigmatic example of when to switch to the GLR engine [7]. Here are some cases:

```
type subrange = lo .. hi;
type enum = (a, b, c);
```

The original language standard allows only numeric literals and constant identifiers for the subrange bounds (`lo` and `hi`), but Extended Pascal (ISO/IEC

10206) [13] and many other Pascal implementations allow arbitrary expressions there. This gives rise to declarations like the following:

`type subrange = (a) .. b;`	`type enum = (a);`

The corresponding declarations look identical until the '`..`' token. With normal LALR(1) one-token lookahead it is not possible to decide between the two forms when the identifier '`a`' is parsed. It is, however, desirable for a parser to decide this, since in the latter case '`a`' must become a new identifier to represent the enumeration value, while in the former case '`a`' must be evaluated with its current meaning, which may be a constant or even a function call. The Bison manual considers and discards several potential solutions to the problem to conclude that the best approach is to declare the parser to use the GLR algorithm. To aggravate the conflict we have added the C *comma* operator inside `expr`[2], making room for the generation of declarations like:

`type subrange = (a, b, c) .. (d, e);`	`type enum = (a, b, c);`

which makes the parsing even more difficult.

Here is our modification of the vastly simplified subgrammar of Pascal type declarations found in [7].

```
%token ID  = /([A-Za-z]\w*)/
%token NUM = /(\d+)/

%left   ','
%left   '-' '+'
%left   '*' '/'

%%

type_decl : 'TYPE' ID '=' type ';'
;

type :
      '(' id_list ')'
    | expr '..' expr
;
```

```
id_list :
      ID
    | id_list ',' ID
;

expr :
      '(' expr ')'
    |  expr '+' expr
    |  expr '-' expr
    |  expr '*' expr
    |  expr '/' expr
    |  expr ',' expr /* new */
    |  ID
    |  NUM
;
```

[2] Perhaps the language designer wants to extend Pascal with lexicographic ranges

When used as a normal LALR(1) grammar, eyapp correctly complains about two reduce/reduce conflicts:

```
$ eyapp -v pascalenumeratedvsrange.eyp
2 reduce/reduce conflicts
```

The generated `.output` file tell us that both conflicts occur in state 11. It also give us the contents of state 11:

```
 State 11:
     id_list -> ID . (Rule 4)
     expr -> ID .    (Rule 12)

     ')' [reduce using rule 12 (expr)]
     ')' reduce using rule 4 (id_list)
     '*' reduce using rule 12 (expr)
     '+' reduce using rule 12 (expr)
     ',' [reduce using rule 12 (expr)]
     ',' reduce using rule 4 (id_list)
     '-' reduce using rule 12 (expr)
     '/' reduce using rule 12 (expr)
```

From the inspection of state 11 we can conclude that the two reduce-reduce conflicts occur between productions `id_list -> ID` and `expr -> ID` in the presence of tokens ')' and ','. To solve the conflict we label the two involved productions and set the `%PREC` directives:

```
id_list :
      ...
      %name ID:ENUM
      ID                      %PREC rangeORenum
      ...
expr : '(' expr ')'
      ...
    | %name ID:RANGE
      ID                      %PREC rangeORenum
      ...
```

When the conflict point is reached the conflict handler below calls the method `YYLookBothWays(`a, b`)`.

```
%conflict rangeORenum {
  my $s = $self->YYLookBothWays('TYPE', ';');
  if ($s =~ /^TYPE ID = \( ID ( , ID )* \) ;$/x)
       { $self->YYSetReduce([',', ')'], 'ID:ENUM' ); }
  else { $self->YYSetReduce([',', ')'], 'ID:RANGE' ); }
}
```

The substring from the *right sentential form*[3] between `'TYPE'` and `';'` is stored in `$s`. If the string of tokens `$s` in Σ^* conforms to the syntax of an enumerated type

```
/^TYPE ID = \( ID ( , ID )* \) ;$/
```

we set the parsing action to reduce by the production $id\_list \rightarrow ID$ for the two conflictive tokens `','` and `')'`. Otherwise the input represents a range type.

In most cases, as occurs in this example, the nested parsing step required to decide which action must be taken can be accomplished through a simple regular pattern. Nested parsing is extraordinarily eased by the fact that Perl 5.10 standard regular patterns permit the description of context free languages. Even more, modules like `Regexp::Grammar` [15] bring Perl 6 [11] regular patterns to Perl 5, extending Perl 5 regular patterns beyond the capabilities of Packrat parsing [14].

The call `YYLookBothWays`(a, b) returns the string that is the concatenation of the transition tokens in the *stack* after token a followed by the tokens in the *unexpended input* before token b. It does not alter the current parsing position.

To be more precise, suppose that at the time of the call the pair

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m, a_i a_{i+1} \cdots a_n \$) \tag{1}$$

is the *configuration* of the LR parser. Here $X_j \in \Sigma \cup V$ is a token or a syntactic variable, $a_k \in \Sigma$ are tokens and s_i are the states of the LR automata. Remember that the equation $\delta(s_k, X_{k+1}) = s_{k+1}$ for each k is hold by any configuration, δ being the transition function. This configuration corresponds to the right sentential form

$$X_1 X_2 \cdots X_m, a_i a_{i+1} \cdots a_n \$ \tag{2}$$

which must be in the rightmost derivation from the grammar start symbol being built. The call `YYLookBothWays(`a, b`)` returns

$$X_j \cdots X_m a_i a_{i+1} \cdots a_s \$ \tag{3}$$

where j is the shallowest index in the stack such that $X_j = a$ and s is the nearest index in the unexpended input such that $a_s = b$.

[3] A string $\alpha \in (\Sigma \cup V)^*$ is said a *right sentential form* for a grammar $G = (\Sigma, V, P, S)$ if, and only if, exists a rightmost derivation from the start symbol S to α

5 Conflicts Requiring Unlimited Look-ahead

The following unambiguous grammar can not be parsed by any LL(k) nor LR(k), whatever the value of `k`, nor for packrat parsing algorithms [14].

```
%%
S: x S x | x  ;
%%
```

Though it is straighforward to find equivalent LL(1) and LR(1) grammars (the language is even regular: `/x(xx)*/`), both GLR [4] and Backtrack LR parsers [2] for this grammar will suffer of a potentially exponential complexity in the input size. The unlimited number of look-aheads required to decide if the current `x` is in the middle of the sentence, leads to an increase in the number of branches to explore. The challenge is to make the parser work *without changing* the grammar. Figure 2 shows a solution using PPCR:

```
%conflict isInTheMiddle {
  my $nxs = $self->YYSymbolStack(0,-1, 'x'); # number of visited 'x's
  my $nxr = (unexpendedInput() =~ tr/x//);   # number of remaining 'x's

  if ($nxs == $nxr+1) { $self->YYSetReduce('x', ':MIDx' ) }
  else { $self->YYSetShift('x') }
}

%%
S:
     x  %PREC isInTheMiddle S x
  |  %name :MIDx
     x  %PREC isInTheMiddle ;
```

Fig. 2. Parsing a Non LL(k) nor LR(k) nor packrat grammar

A call `$self->YYSymbolStack(`$a, b, [filter]$`)` returns the list of symbols associated with the parser stack states between a and b. A negative value of a or b refers to the position from the end of the list. Thus, a call like `$self->YYSymbolStack(0,-1)` returns the whole list of symbols in the parsing stack. The optional $filter$ argument can be a string, a closure[4] or a regular pattern.

[4] A closure is a first-class function with free variables that are bound in the lexical environment

When it is a pattern the sublist for which the pattern matches is selected. If it is a closure, it returns the sublist of symbols for which the evaluation of the code is true. If it is a string, as in `$self->YYSymbolStack(0,-1, 'x')`, the sublist of symbols equal to the string is selected. Since the assignment `$nxs = $self->YYSymbolStack(0,-1, 'x')` is evaluated in a scalar context, the length of the resulting sublist is stored in the variable `$nxs`. The copy of the unexpended input - returned by the call to `unexpendedInput()` - is then scanned for `'x'`s and its number is stored in `$nxr`. When `$nxs` equals `$nxr +1` it is time to reduce by $S \rightarrow x$. It may seem that this solution cannot be generalized when `'x'` is an arbitrary grammar. Remember however that Perl 5.10 regular patterns can parse any context free grammar [15].

We can now compile the former program `nopackratSolved.eyp` to generate a script `nopackratSolved.pl` containing the parser. When executed with input `xxx` it outputs a description of the abstract syntax tree:

```
$ eyapp -TC -o nopackratSolved.pl nopackratSolved.eyp
$ ./nopackratSolved.pl -t -i -c 'xxx'
S(TERMINAL[x],S(TERMINAL[x]),TERMINAL[x])
```

Option `-T` instructs `eyapp` to automatically insert semantic actions to produce a data structure representing the abstract syntax tree. Option `-C` tells the compiler to produce an executable (by default it produces a class containing the parser). Eyapp provides default lexical analyzer, error handler and main subroutines for the generated program. The default `main` subroutine admits several command line options, like: `-t` (print the AST), `-i` (print the semantic values of the tokens) and `-c arg` (take the input from `arg`).

6 Conclusions

The strategy presented in this paper extends the classic yacc precedence mechanisms with new dynamic conflict resolution mechanisms. These new mechanisms provide ways to resolve conflicts that can not be solved using static precedences. They also provides finer control over the conflict resolution process than existing alternatives, like GLR and backtracking LR. There are no limitations to PPCR parsing, since the conflict handler is implemented in a universal language and it then can resort to any kind of nested parsing algorithm. The conflict resolution mechanisms presented here can be introduced in any LR parsing tools, since they are independent of the implementation language and the language used for the expression of the semantic actions. One disadvantage of PPCR is that it requires some knowledge of LR parsing. Though the solution may be more efficient, it certainly involves more programmer work than branching methods like GLR or backtracking LR.

Acknowledgments

This work has been supported by the EC (FEDER) and the Spanish Ministry of Science and Innovation inside the 'Plan Nacional de I+D+i' with the contract number TIN2008-06491-C04-02. It has also been supported by the Canary Government project number PI2007/015.

References

1. Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Reinhart, and Winston, 1979. AT&T Bell Laboratories Technical Report July 31, 1978.
2. Gary H. Merrill. Parsing Non-LK(k) Grammars with Yacc. Software, Practice and Experience 23(8): 829-850 (1993).
3. Kernighan & Ritchie. *The C Programming Language.* Prentice Hall.
4. Tomita, M. (1990). The Generalized LR Parser/Compiler - Version 8.4. In *Proceedings of International Conference on Computational Linguistics (COLING'90)*, pages 59–63, Helsinki, Finland.
5. Mcpeak, Scott. September 2004. Elkhound: A Fast, Practical GLR Parser Generator. `http://scottmcpeak.com/elkhound/`
6. Adrian D. Thurston and James R. Cordy. A Backtracking LR Algorithm for Parsing Ambiguous Context-Dependent Languages. 2006 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2006), pp. 39-53, Toronto, October 2006.
7. Charles Donnelly and Richard M. Stallman. Bison: the yacc-compatible parser generator. Technical report, Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, Tel: (617) 876-3296, 1988.
8. Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachussetts, 1990.
9. Rodríguez-León Casiano. `Parse::Eyapp` Manuals. 2007.
   CPAN: `http://search.cpan.org/dist/Parse-Eyapp/`
   google-code: `http://code.google.com/p/parse-eyapp/`
10. Wall, L., Christiansen, T., Schwartz, R. (1996). *Programming Perl.* O'Reilly & Associates.
11. Allison Randal, Dan Sugalski, Leopold Totsch. *Perl 6 and Parrot Essentials.* O'Reilly Media. June 2004.
12. Hudak, P. Modular Domain Specific Languages and Tools. ICSR '98: Proceedings of the 5th International Conference on Software Reuse. IEEE Computer Society. Pages 134-142, June 1998.
13. ISO. Extended Pascal ISO 10206:1990.
    `http://www.standardpascal.org/iso10206.txt`.
14. Bryan Ford. Functional Pearl: Packrat Parsing: Simple, Powerful, Lazy, Linear Time.
    `http://pdos.csail.mit.edu/ baford/packrat/icfp02/packrat-icfp02.pdf` (2002).
15. Damian Conway. Regexp::Grammars. Add grammatical parsing features to Perl 5.10 regexes. `http://search.cpan.org/dist/Regexp-Grammars/`.