Extending LR Conflict Resolution Mechanisms

C. Rodriguez-Leon and L. Garcia-Forte

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

Abstract. Yacc-like LR parser generators provide ways to statically solve shift-reduce mechanisms based on token precedence. No mechanisms are provided for the resolution of difficult reduce-reduce and shift-reduce conflicts. To solve such kind of conflicts the language designer has to modify the grammar. The usual approach to cope with these limitations has been to extend LR parsers with the capacity to branch at any multivalued table entry. The strategy presented in this paper takes an entirely different path: it extends yacc conflict resolution sub-language with new dynamic conflict resolution constructs and nested parsing. The LR parsing table is modified at parsing time in terms of the information provided by the new conflict resolution directives. These extensions supply ways to solve any kind of LR conflicts and can be applied to any kind of LR engines, including Backtracking and Generalized LR engines.

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 difficult reduce-reduce or shift-reduce conflicts. To solve such kind of conflicts the language designer has to modify the grammar.

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. 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 [2], via the `%glr-parser directive` and Elkhound [3] 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 is 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 [5]. The user must always analyze the conflicts 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. Furthermore, the interactions with the lexer have to be considered with great care. Since a split parser consumes tokens without performing any actions during the split, the lexer cannot obtain information via parser actions.

The strategy presented in Sections 2 and 3 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 is essentially the same but the parsing tables can be modified at run time.

The technique involves labelling the points in conflict in the grammar specification and providing additional information 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 some code inspects the state of the underlying parse engine to decide the appropriate solution. Any grammar ambiguity can be fixed with a minimal amount of branching.

This technique can be combined to complement both GLR and backtracking LR algorithms to give the programmer a finer control of the branching process. It puts the user - as it naturally 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. When using the high level formulation, PPCR does not requires more work and more knowledge of LR parsing than the usual in yacc programming.

LR conflict removal is a laborious task. The number of conflicts in a programming language can reach tens and even hundreds: The original grammars of Algol-60 and Scheme result in 61 and 78 conflicts respectively with an average density of one conflict for each two productions. By adding Postponed Conflict Resolution to the classical precedence and associativity settings we can fix all the conflicts in such grammars without modifying the grammars. Removing conflicts while preserving the grammar is preferable to rewriting the grammar in several situations: When using a conflict removal tool like the one described in [6], since the language designer will be still familiar with the resulting grammar, when the original grammar better reflects the author ideas about the syntax and semantic of the language, when the original grammar is easier to read and to understand (size matters) and when (see section 4) such unambiguous grammar is hard or impossible to find. In fact, the general question of whether a grammar is not ambiguous is undecidable. No algorithm can exist to determine the ambiguity of a grammar because the undecidable Post correspondence problem can be encoded as an ambiguity problem.

The techniques described in this paper have been implemented in `eyapp` [7], a yacc-like LALR parser generator for Perl [8]. All the grammar examples used in this work can be found in [17].

This paper is divided in seven sections. The next section introduces the Low Level Postponed Conflict Resolution (LLPPCR) strategy. Section 3 uses a grammar that depicts the well known C++ ambiguity between expression statements and declarations [9] to introduce the technique and motivate the syntax and semantics of the High Level PPCR strategy (HLPPCR). Section 4 illustrates how HLPPCR can be used to parse inherently ambiguous languages keeping full control of the derivation process. Section 5 *What is the Power of PPCR?* discusses the capabilities and limits of LLPPCR and HLPPCR. Section 6 explains the algorithms behind and some implementation details. The last section summarizes the advantages and disadvantages of our proposals.

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 `yacc/eyapp` compiler [7] announces the presence of a reduce-reduce conflict. The steps followed to solve a reduce-reduce conflict using the PPCR strategy are:

1. Identify and understand 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` using option `-v` or the graphical description obtained using option `-w` or with the help of expert systems [6, 13]. Suppose we identify that the participants are 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 production label, like `:rA`. The former call sets the parsing action for the state associated with the conflict `IsAorB` to reduce by the production `:rA` for the tokens involved in the conflict.

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 of a subgrammar component of the main grammar. Therefore, for the PPCR method to work we need also to modify LR parsing to give some support for nested parsing (see section 6 for the details). Ideally, we can use an extension providing LR composition like the ones described in [10–12]. The call to `is_A` 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 $A \to \alpha_\uparrow$ and $B \to \beta_\uparrow \gamma$ for some token `'@'`. Only steps 4 and 5 change slightly:

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

`%name :rA` $A \to \alpha$ `%PREC IsAorB`
$B \to \beta$ `%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 Low Level and High Level PPCR

3.1 An Example: The C++ ambiguity

This section illustrates the technique through a problem that arises in C++. The C++ syntax [9] does not disambiguate between expression statements (`stmt`) and declaration statements (`decl`). The ambiguity arises when an expression statement has a function-style cast as its left-most subexpression. Since C [14] does not support function-style casts, this ambiguity does not occur in C programs. For example, the phrase `int (x) = y+z;` parses as either a `decl` or a `stmt`. The disambiguation rule used in C++ is that *if the statement can be interpreted both as a declaration and as an expression, the statement is interpreted as a declaration statement.* The following examples disambiguate into *expression* statements when the potential *declarator* is followed by an operator different from equal or semicolon (`type_spec` stands for a type specifier):

expr	dec

`type_spec(i)++;`
`type_spec(i,3)<<d;`
`type_spec(i)->l=24;`
	`type_spec(*i)(int);`
	`type_spec(j)[5];`
	`type_spec(m) = { 1, 2 };`

Regarding to this problem, Bjarne Stroustrup [15] remarks:

> *Simple lexical lookahead can help a parser disambiguate most cases. Consider analyzing a statement consisting of a sequence of tokens as follows:*
>
> `type_spec (dec_or_exp) tail`
>
> *Here* `dec_or_exp` *must be a declarator, an expression, or both for the statement to be legal. This implies that* `tail` *must be a semicolon, something that can follow a parenthesized declarator or something that can follow a parenthesized expression, that is, an initializer,* `const`, `volatile`, `(`, `[`, *or a postfix or infix operator. The general cases cannot be resolved without backtracking, nested grammars or similar advanced parsing strategies. In particular, the lookahead needed to disambiguate this case is not limited.*

3.2 Simplifying the Problem

The following Eyapp grammar depicts an oversimplified version of the C++ ambiguity

```
%right '='
%left '+'
%%
prog: /* empty */         | prog stmt ;
stmt: expr ';'            | decl      ;
expr: ID                  | NUM
    | INT '(' expr ')'    # typecast
    | expr '+' expr       | expr '=' expr
;
decl: INT declarator ';'
    | INT declarator '=' expr ';'
;
declarator: ID            | '(' declarator ')' ;
%%
```

3.3 Identifying the problem

This grammar is ambiguous since an input like: `int (x) = 4;` can be interpreted as a `decl` or an `expr`. The `eyapp` compiler warn us of the presence of reduce/reduce conflict:

```
$ eyapp -W SimplifiedCplusplusAmbiguity.eyp
  1 reduce/reduce conflict
```

When we look at the `.output` file or to the generated automaton graph we see the reasons for the conflict:

```
Warnings:
---------
1 reduce/reduce conflict
Conflicts:
----------
..................................
State 18 contains 1 reduce/reduce conflict
..................................
State 18:

    expr -> ID .    (Rule 5)
    declarator -> ID .  (Rule 12)

    ')' [reduce using rule 12 (declarator)]
    ')' reduce using rule 5 (expr)
    '+' reduce using rule 5 (expr)
    '=' reduce using rule 5 (expr)
```

This information tell us that when parsing `'int (x↑) = 4;'`, once the parser has seen the `ID x` and is in the presence of the closing parenthesis `')'`, it is incapable to decide whether to reduce by rule 12 (`declarator -> ID`) or rule 5 (`expr -> ID`) .

3.4 Low Level PPCR Syntax

The next figure shows a low level solution to the problem using PPCR.

```
%{
my $ISDEC;
%}
%token NUM = /(\d+)/
%token INT = /(int)\b/
%token ID  = /([a-zA-Z_][a-zA-Z_0-9]*)/
%right '='
%left '+'
%conflict decORexp {
  if ($ISDEC)
    { $self->YYSetReduce('ID:DEC' ); }
  else
    { $self->YYSetReduce('ID:EXP' ); }
}
%explorer decORexp { $ISDEC = $self->YYPreParse('decl'); }
%expect-rr 1  # expect 1 reduce-reduce conflict
%%
prog: /* empty */
    | prog %decORexp? stmt
;
stmt: expr ';'
    | decl
;
expr:
      %name ID:EXP
      ID                    %PREC decORexp
    | NUM
    | INT '(' expr ')' /* typecast */
    | expr '+' expr
    | expr '=' expr
;
decl: INT declarator ';'
    | INT declarator '=' expr ';'
;
declarator:
      %name ID:DEC
      ID                    %PREC decORexp
    | '(' declarator ')'
;
%%
```

Additionally to the classic static precedence directives (lines 7-8) we can see two PPCR directives: `%conflict` (lines 10-14) and `%explorer` (line 15).

The auxiliary variable `$ISDEC` (declared at line 2 and used inside the conflict handler at line 10) will be set to `true` if, and only if, the conflictive incoming input conforms (line 15) to the language defined by `decl` (line 32).

The conflict handler `decORexp` simply sets the LR action to reduce by `expr` $\rightarrow$ ID or `declarator` $\rightarrow$ `ID` according to the value of `$ISDEC` (lines 10-13).

The `%explorer` directive used in line 15 has the syntax:

```
%explorer conflictName { CODE }
```

the purpose of `CODE` is to perform some nested parsing, to decide which action is correct.

The call to the method `YYPreParse` in line 15 returns true if there is a prefix of the unexpended input that belongs to the language defined by the `decl` subgrammar (line 32).

The point where the exploration starts - defining the time when the code of the `%explorer` directive is called - is marked inside the grammar body using the `%conflictname?` syntax:

```
18  prog:
19        /* empty */
20      | prog %decORexp? stmt
21  ;
```

The `eyapp` compiler creates a syntactic variable whose only empty production has as associated semantic action the code defined in the `%explorer` directive. The points where the `%decORexp?` directive appears are substituted by that variable. Thus, the former lines are conceptually equivalent to:

```
prog:
    /* empty */
  | prog decORexp.explorer stmt
;

decORexp.explorer : /* empty */
       { $ISDEC = $self->YYPreParse('decl'); }
```

The `%expect-rr 1` directive at line 16 keeps `eyapp` silent, avoiding the warnings regarding the reduce-reduce conflict.

3.5 High Level PPCR Syntax

The methodology followed in the previous section is so general that we have extended `eyapp` with a higher level syntax which specifies this particular way of solving difficult conflicts:

```
%conflict conflictName nestedParser? actionName: actionName
```

Where `conflictName` is the name given to the conflict, `nestedParser` refers to the sub-grammar used for pre-parsing the incoming input and `actionName` is the name of a production involved in the conflict or the reserved word `shift`.

The construct is translated as follows: The parser for `nestedParser` is called each time the exploration point is reached, saving the result inside an attribute of the parser. Each time the conflict state is reached, this attribute is checked and the corresponding parsing action is taken.

The next figure shows the complete header of the solution to the C++ example using the new syntax. The body is exactly the same that appears in the previous listing in section 3.4. In this version any explicit code has disappeared.

```
%token NUM = /(\d+)/
%token INT = /(int)/
%token ID  = /([a-zA-Z_][a-zA-Z_0-9]*)/

%right '='
%left '+'

%conflict decORexp decl? ID:DEC : ID:EXP

%expect-rr 1  # expect 1 reduce-reduce conflict

%%
prog: /* empty */ | prog %decORexp? stmt ;
stmt: expr ';'    | decl ;
expr:
      %name ID:EXP
      ID                      %PREC decORexp
    | NUM          | INT '(' expr ')'
    | expr '+' expr | expr '=' expr
;
decl: INT declarator ';' | INT declarator '=' expr ';'
;
declarator:
      %name ID:DEC
      ID                      %PREC decORexp
    | '(' declarator ')'
;
%%
```

4 Parsing Inherently Ambiguous Languages

A context-free language is inherently ambiguous if all context-free grammars generating that language are ambiguous. While some context-free languages have both ambiguous and unambiguous grammars, there are context-free languages for which no unambiguous context-free grammar can exist. An example of an inherently ambiguous language is the set

$$\{a^n b^m c^r \text{ such that } n = m \text{ or } m = r \text{ and } n, m, r \geq 0\}$$

which is context-free, since it is generated by the following grammar:

```
1  %%
2  s: aeqb | beqc ;
3  aeqb: ab cs ;
4  ab: /* empty */ | 'a' ab 'b' ;
5  cs: /* empty */ | cs 'c' ;
6  beqc: as bc ;
7  bc: /* empty */ | 'b' bc 'c' ;
8  as: /* empty */ | as 'a' ;
9  %%
```

The symbol `aeqb` correspond to guess that there are the same number of `a`s than `b`s. In the same way, `beqc` starts the subgrammar for those phrases where the number of `b`s is equal to the number of `c`s. The usual approach to eliminate the ambiguity by changing the grammar to an unambiguous one does not work. Such grammar does not exist and will never be found. Observe, however, that the language is not an exotic one. In fact, some usefull languages contain this kind of constructs. When parsing such languages the programmer has no other way than to resort to semantic actions to check that the constraints are satisfied. But this approach implies that we can't cleanly separate syntactic analysis and semantic actions for such kind of languages.

The ambiguity above is pointed out by the compiler as a shift-reduce conflict in the presence of token `'a'`:

– We can `reduce` by production `as: /* empty */` (line 8), meaning that we will consider the sequence of `a` as a free sequence, and we commit to option `beqc` or
– We can `shift` from item `ab: .'a' ab 'b'` to `ab: 'a'. ab 'b'`, in whose case, we choose to follow via `aeqc`

Using PPCR the yacc programmer keeps control of the specific AST to build, either via `aeqb` or via `beqc`. The high-level PPCR solution below gives preference to `aeqb`. By changing the conflict resolution directive at line 1 by:

```
%conflict AorC bc? BC : shift
```

we give preference to `beqc` trees instead.

```
%conflict AorC ab? shift: BC
%%
st:  %AorC? s ;
s: aeqb | beqc ;
aeqb: ab cs ;
ab: /* empty */
  | 'a'         %PREC AorC ab 'b'
;
cs: /* empty */ | cs 'c' ;
beqc: as bc ;
bc: /* empty */ | 'b' bc 'c' ;
as: %name BC
    /* empty */ %PREC AorC
  | as 'a'
;
%%
```

5 What is the Power of PPCR?

In terms of grammar parsing abilities, we presume that any Context Free Grammar can be parsed using a Low Level PPCR LALR parser to produce the AST the user requires. This presumption is based in the fact that we can resort to the full Turing Complete target language to find the conditions that satisfactorily solve the conflict. The answer to this question when using only the High Level PPCR syntax - the syntax introduced in section 3.5 - is not so clear. Here the expression *High Level PPCR* refers to PPCR solutions in which no explicit code in the hosting language appears, either in the main grammar or in the auxiliar subgrammar used to decide the correct action.

We believe, but have no formal proof, that the following unambiguous grammar is an example of a Low Level PPCR grammar that is not a High Level PPCR grammar.

```
%%
T: S          ;
S: x S x | x  ;
x: NUM   |  x OP NUM ;
%%
```

Though it is straighforward to find equivalent LL(1) and LR(1) grammars[1] the grammar can not be parsed by any LL(k) nor LR(k), nor by packrat parsing algorithms. [18]. An LR parser has to shift on `NUM` until the middle `x` is reached.

[1] The language is even regular in `x`. Is given by: `/x(xx)*/`

At that time it has to reduce by $S \rightarrow x$. This is the `eyapp` description of the conflict:

```
State 6:                              # If we are in this state
 S -> x . S x       (Rule 2)      #  we just have seen an x:
 S -> x .           (Rule 3)      #  Observe the dot after x
 x -> x . OP NUM    (Rule 5)      #  in the 3 kernel items
 NUM shift, and go to state 4     # NUM will be reduced to x
 OP shift, and go to state 8      # Still in the same x
 $end reduce using rule 3 (S)     # Only one x
 NUM [reduce using rule 3 (S)]    # Only if it is the middle x
 S go to state 7                  # Follow in rule 2
 x go to state 6

State 4:
        x -> NUM .          (Rule 4)
        $default         reduce using rule 4 (x)
```

The challenge here is to make the parser work *without changing* the grammar.

Let us see that this grammar is Low Level PPCR. We indentify the conflict - which we name `isInTheMiddle` -, labelling as `:MIDx` the reduction item and marking the exploration point:

```
%token NUM = /(\d+)/
%token OP  = /([-+*\/])/
%{
my $nxs = 0;
%}
%conflict isInTheMiddle {
  $nxs++;
  if ($nxs == $self->YYVal('ExpList')) {
    $self->YYSetReduce(':MIDx' );
    $nxs = 0;
  }
  else { $self->YYSetShift() }
}
%explorer isInTheMiddle ExpList

%%
T: %isInTheMiddle? S ;
S:   x  %PREC isInTheMiddle S x
  |  %name :MIDx
     x  %PREC isInTheMiddle
;
x: NUM | x OP NUM ;
%%
```

The explorer directive at line 14 is translated by the compiler into a call to `$self->YYPreParse('ExpList')`. The auxiliary grammar `ExpList` contains *semantic actions*, to compute the middle `x`:

```
%%
ExpList: S    { 1+int($_[1]/2) } ;
S:       S x  { $_[1] + 1 } |  x   { 1 } ;
x:       NUM                |  x OP NUM ;
%%
```

The semantic value returned by the `ExpList` grammar is later accessed inside the conflict handler code above via the method call `$self->YYVal('ExpList')`.

The conflict solver code is quite simple: it keeps the position of the current `x` inside the variable `$nxs`. Each time a new `x` is seen, `$nxs` is incremented. The reduction is called when the middle point is reached.

The proof that the conflicts in this grammar can not be solved using High Level PPCR remains open for us.

6 Implementation and Performance

For PPCR to work we need some slight modifications in the LR parsing algorithm. The following Perl pseudocode describes the PPCR parsing algorithm. Our notation follows the conventions used in [16]:

```
method PPCRparse(LRParser $p)      # $p is the LR parser object
 my @stack;                             # The LR parsing stack
 my $s0 = $p->startstate;
 push(@stack, $s0);        # Push the start state in the stack
 my $b = $p->yylex();                         # Get next token
 forever do {
   my $s = top();     # Get the state in the top of the stack
   my $a = $b;
   my $act;
   if ($p->isConflictive($s, $a)) $p->conflictHandler($s, $a);
   $act = $p->action($s->state, $a);
   switch ($act) {
     case "shift t" :
       my $t;
       $t->{state} = t;
       $t->{attr}  = $a->{attr};
       push($t);               # Store the state in the stack
       $b = $p->yylex();                      # Get next token
       break;
     case "reduce A → α" :
       my $r;
                    # Get the attributes of the |α| top states
       my @ats = getAttributes(@stack[-|α| .. -1]);
       $r->{attr} = $p->Sem{A->α}->(@ats);
       pop(|α|);                         # Pop length(α) states
26         my $t = top();
27         $r->{state} = $p->goto($t, A);
28         push($r);
29         break;
30       case "accept" : return ($s->attr);
31       default : $p->yyerror("syntax error");
32     }
```

As usual `|`α`|` denotes the length of string α. The call `top()` returns the state in the top of the stack, while the call `pop(k)` extracts the top `k` states from the stack. The notations `$a->{attr}` and `$t->{attr}` make reference to the attribute associated with the token `$a` and to the attribute associated with the state `$s`. The call `getAttributes(@stack[-|`α`| .. -1])` returns the list of attributes of the `|`α`|` top states in the stack. A negative index in Perl starts from the end of the array, thus `$stack[-1]` is the state in the top of the stack. The hash entry `$Sem{A`$\rightarrow$ α`}` is a reference to the semantic code associated with production `A`$\rightarrow$ α. Such code is called in line 24, whenever a reduction by `A`$\rightarrow$ α is detected.

The calls to `action` (line 11) and `goto` (line 27) respectively return the corresponding entries of the `ACTION` and `GOTO` parsing tables.

The only change is in line 10. If the couple state-token is conflictive, i.e. if the LR action table was multivalued, the conflict handler for such entry is called to decide which one is the correct action.

The information about the correspondence between conflictive pairs (*state*, *token*) and conflict handlers used by the method `conflictHandler` is generated by the compiler during the construction of the LR tables using the information gathered from the `%PREC` directives.

This give us a measure of the overhead introduced by PPCR per iteration: the cost of checking the condition (line 10). In case of being a conflictive state we also have to add the cost of executing the handler, which usually takes constant time: it checks the attribute containing the result of the last explorer execution and changes accordingly the corresponding entry of the action table. Observe that a yacc compiler can generate optimal code avoiding the PPCR extra cost by using the classical LR algorithm when no conflict handlers were defined.

An additional modification is needed when generating the LR tables for the sub-grammar starting in A. Nested parsers for a nonterminal A must accept prefixes of the incoming input not necessarily terminated by the end of input token. The action table is modified to have an *accept* action for each entry (s, a) where the state s has an LR item $[A' \rightarrow A_{\uparrow}, a]$ where $a \in FOLLOW(A)$. That is, it accepts when something derivable from A has been seen and the next token is a legal one. Here A' denotes the super-start symbol for the sub-grammar starting in A. The set $FOLLOW(A)$ is the set of tokens b in the super-grammar such that exists a derivation $S \overset{*}{\Longrightarrow} \alpha A b \beta$. The end of input is also included in $FOLLOW(A)$ if, and only if, exists a derivation $S \overset{*}{\Longrightarrow} \alpha A$.

As was mentioned in section 3, for each explorer definition the `eyapp` compiler creates an auxiliary syntactic variable whose only empty production has as associated semantic action the code defined by the `%explorer` directive. Such code can be explicit as in the low level example in section 3.4 or implicit as in the examples in sections 3.5 and 5. The points where the `%conflict?` directive appears inside the body are substituted by that variable. The execution of the exploration code is an additional source of overhead. The programmer must choose the exploration point in a way that

– The set of exploration points must hold the following covering property: any visit to the conflictive state must be preceded by a visit to an exploration state
– When the exploration state is reached, the incoming input belongs to the exploration language if, and only if, the selected production is a suitable handler for the LR anti-derivation at the time of the next visit to the conflict state
– Does not introduce new conflicts. This restriction can be lifted if, instead of implementing explorers through a syntactic variable, the PPCR algorithm is modified to execute the exploring code when the state is an explorer-state
– Minimizes the time taken by the decision/exploration code,

The last version of `eyapp` can be obtained from [7]. There is also a repository of difficult and conflictive yacc grammars with their corresponding PPCR solutions in [17]. Contributions are welcome.

7 Conclusions

The strategy presented in this paper extends the classic yacc precedence mechanisms with new dynamic conflict resolution constructs. These new instruments provide ways to resolve conflicts that can not be solved using static precedences. providing a finer control over the conflict resolution process than other alternatives like GLR and backtracking LR. The cost of implementing these extensions in existing yacc-like parsers is minimal.The lost of efficiency is also negligible.

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. Thanks to Jose Manuel Rodríguez de la Rosa for his valuable comments. It has also been supported by the Canary Government project number PI2007/015.

References

1. Johnson, S. C. 1979. Yacc: Yet another compiler compiler. *AT&T Bell Laboratories Technical Report July 31, 1978 2*, 353–387.

2. Donnelly, C. and Stallman, R. M. 1995. Bison: the yacc-compatible parser generator. Technical report, Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, Tel: (617) 876-3296.
3. Mcpeak, S. 2004. Elkhound: A fast, practical GLR parser generator. Available at: `http://scottmcpeak.com/elkhound/`.
4. Tomita, M. 1990. The generalized LR parser/compiler m Proceedings of International Conference on Computational Linguistics (COLING'90). Helsinki, Finland, 59–63.
5. Thurston, A. D. and Cordy, J. R. 2006. A backtracking LR algorithm for parsing ambiguous context-dependent languages. In 2006 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2006). Toronto, 39–53.
6. Teixeira Passos, L. Bigonha, M. A.S., Bigonha, R.S., A Methodology for Removing LALR(k) Conflicts Journal of Universal Computer Science 13 737–752 2007.
7. Rodríguez-León C., `Parse::Eyapp` Manuals. 2011.
   CPAN: `http://search.cpan.org/dist/Parse-Eyapp/`
   google-code: `http://code.google.com/p/parse-eyapp/`
8. Wall L., Christiansen T., S. R. 1996. *Programming Perl.* O'Reilly & Associates.
9. Stroustrup, B. 1986. *The C++ Programming Language.* Addison Wesley.
10. Wu, X., Bryant, B.R., Gray, J., Mernik, M., Component-based LR parsing. m Computer Languages, Systems & Structures 36(1), 16-33 (2010)
11. Schwerdfeger A, Van Wyk E., Verifiable Parse Table Composition for Deterministic Parsing Software Language Engineering, Lecture Notes in Computer Science Volume: 5969 184-203 2010
12. Bravenboer, M., Visser, E., Parse Table Composition. Software Language Engineering LNCS, 74–94 (2009)
13. Basten, H.J.S., Vinju, J.J., Faster Ambiguity Detection by Grammar Filtering. Tenth Workshop on Language Descriptions, Tools, and Applications (LDTA 2010), Paphos, Cyprus, March 2010
14. Ritchie, K. . 1988. The C Programming Language. Prentice Hall.
15. Ellis, M. A. and Stroustrup, B. 1990. *The Annotated C++ Reference Manual.* Addison-Wesley.
16. Aho A.V., Lam, M., Sethi, R., Ullman, J., *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, August 2006.
17. Rodríguez-León, C., García, L., A Repository of LALR Conflictive Grammars google-code: `http://code.google.com/p/grammar-repository/`, 2011.
18. Ford, B. 2002. Functional Pearl: Packrat Parsing: Simple, Powerful, Lazy, Linear Time.
    `http://pdos.csail.mit.edu/ baford/packrat/icfp02/packrat-icfp02.pdf`.