Solving LR Conflicts via Nested Parsing and Dynamic Modification of The Action Table

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 specifying which production must be chosen in terms of the results of some nested parsing. The LR parsing table is then modified at parsing time using the information provided by the nested parser. Through this simple modification LALR parsers are able to parse ambiguous gramars and recognize inherently ambiguous languages.

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 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.

The Postponed Conflict Resolution strategy (PPCR) does not requires more work or more knowledge of LR parsing than the usual in yacc programming. LR conflict removal is, however, 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 2.4) such unambiguous grammar is hard or impossible to find.

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 [9].

This paper is divided in four sections. The next section introduces the use of the Postponed Conflict Resolution strategy and illustrates it with several examples. Section 3 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, 10]. Suppose we identify that the participants are two LR(0)-items $A \to \alpha_{\uparrow}$ and $B \to \beta_{\uparrow}$ 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 $A \to \alpha$ has label `:rA` and production $B \to \beta$ 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` $A \to \alpha$
   `%name` `:rB` $B \to \beta$

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

5. Introduce a `%conflict` directive inside the *head* section of the translation scheme to specify the way the conflict is solved:

```
...
%conflict conflictName nestedParser? actionName: actionName
%% /* body */
...
```

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`. In our example is:

```
...
%conflict AorB Is_A? :rA : :rB
%% /* body */
...
```

where `Is_A` is an auxilary syntactic variable. It represents the context-dependent dynamic knowledge that allows us to take the right decision. It is translated into a call to a nested parser for `Is_A`. The language defined by `Is_A` is such that any incoming input belongs to the language if, and only if, $A \to \alpha_\uparrow$ is a suitable the handler for the anti-derivation.

6. The point inside the body of the grammar in which the nested parser for `Is_A` is launched is marked using the `%conflictname?` syntax:

   `C:` λ_1 **`%AorB?`** λ_2

   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.

The construct is translated as follows: The parser for `Is_A` is called each time the exploration point `%AorB?` is reached, saving the result inside an attribute of the parser. Each time the conflict state having items $A \to \alpha_\uparrow$ and $B \to \beta_\uparrow$ is reached, this attribute is checked and the corresponding parsing action is taken.

2.1 Example: a Non LR(k) Grammar

The following example shows a PPCR solution for a simple non LR(k) grammar:

```
/* Original grammar */
%%
A: B C 'd' | E C 'f' ;
B:
 'x' 'y'
;
E:
 'x' 'y'
;
C: /* empty */ | C 'c' ;
```

```
%conflict BorE Is_B? B : E
%%
A: B C 'd' | E C 'f' ;
B: %name B
   'x' 'y' %PREC BorE
;
E: %name E
   'x' 'y' %PREC BorE
;
C: /* empty */ | C 'c' ;
```

where `Is_B` is given by the following grammar:

```
%%
Is_B:  C 'd'                ;
C:  /* empty */ |  C 'c' ;
%%
```

When no exploration point `%BorE?`- as in this example - is specified, the exploration point is the conflict point (i.e. the places where the `%PREC` directive appears).

2.2 Example: Insufficient Lookahead

The following grammar illustrates a typical LALR conflict due to a bad grammar design. The conflict between productions `ds` → `D ';' ds` and `ds` → `D` can be easily solved by changing the second production to `ds` → `ds ';' D`. If, for some reason, we insist in keeping the grammar, we can apply the PPCR methodology:

```
%token D S

%%
p: ds ';' ss  | ss ;
ds: D ';' ds
  | D
;
ss: S ';' ss  | S  ;
%%
```

```
%token D S
%conflict LD Is_L? L : shift
%%
p:  ds ';' ss  | ss ;
ds: D %PREC LD ';' ds
  | %name L D %PREC LD
;
ss: S ';' ss   | S  ;
%%
```

Where `Is_L` is given by the grammar:

```
%%
Is_L: ';' S ;
%%
```

2.3 Example: Extended Pascal

The third example illustrates the technique through a problem that arises in the declaration of enumerated and subrange types in the programming language Pascal. 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) 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. Here is a PPCR solution of a vastly simplified subgrammar of Pascal type declarations found in [2].

```
%conflict rORe range? ID:RANGE : ID:ENUM
%token ID  = /({A-Za-z}\w*)/
%token NUM = /(\d+)/
%left   ','
%left   '-' '+'
%left   '*' '/'
%%
type_decl: 'type' ID '=' type ';'                ;
type:      %rORe? '(' id_list ')' | %rORe? range ;
range:     expr '..' expr                        ;
id_list:
      %name ID:ENUM
      ID                      %PREC rORe
    | id_list ',' ID
;
expr: '(' expr ')'   |  expr '+' expr
    | expr '-' expr  |  expr '*' expr
    | expr '/' expr  |  expr ',' expr
    | %name ID:RANGE
      ID                      %PREC rORe
    | NUM
;
%%
```

2.4 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 below 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'
;
%%
```

3 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 [11]:

```
method PPCRparse(LRParser $p)          # 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);
```

```
25          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→ α}` is a reference to the semantic code associated with production `A→` α. Such code is called in line 24, whenever a reduction by `A→` 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 the correct action.

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. Furthermore, a yacc compiler can generate optimal code by using the classical LR algorithm when no conflict handlers were defined.

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

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 $[A' \to A_{\uparrow}, a]$ 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 $S \stackrel{*}{\Longrightarrow} \alpha A b \beta$. The end of input is also included in $FOLLOW(A)$ if, and only if, exists a derivation $S \stackrel{*}{\Longrightarrow} \alpha A$.

For each explorer definition the current implementation of 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 (see [9] for examples) or implicit as in the example in section 2.4 . 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

- makes easy the nested parsing and the recollection of the information needed to take the decision
- minimizes the number of times it is executed and the time taken by the decision/exploration code,
- guarantees that the decision taken will be valid when the next visit to the conflict state occurs and
- 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.

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 [9]. Contributions are welcome.

4 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. 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. Rodríguez-León, C. García, L., A Repository of LALR Conflictive Grammars google-code: `http://code.google.com/p/grammar-repository/`, 2011.
10. 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
11. Aho A.V., Lam, M., Sethi, R., Ullman, J., *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.