DOI: 10.2298/CSIS101116008R

Solving Difficult LR Parsing Conflicts by Postponing Them

C. Rodriguez-Leon[1] and L. Garcia-Forte[1]

Departamento de EIO y Computación,
Universidad de La Laguna
casiano@ull.es, lgforte@ull.es,
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 difficult shift-reduce or 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.

Keywords: parsing, lexical analysis, syntactic analysis.

1. Introduction

Yacc-like LR parser generators [3] provide ways to solve shift-reduce conflicts 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. Quoting Merrill [5]:

> *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 [1, p. 106]. 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 [1], via the `%glr-parser directive` and Elkhound [4] provide implementations of the Generalized LR (GLR) algorithm [11]. 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 [10, p. 3]. The Bison manual [1] 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 [10] 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 some 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 `eyapp` [7], a yacc-like LALR parser generator for Perl [13, 6].

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 [1] to illustrate the advantages of the GLR engine: the declaration of enumerated and subrange types in Pascal [12]. 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 [2]. The last section summarizes the advantages and disadvantages of our proposal.

2. The *Postponed Conflict Resolution* Strategy

The *Postponed Conflict Resolution* (PPCR) is a 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 can be divided in two activities: conflict identification and mapping (steps 1a to 1d) and writing the solver (step 2a).

1. **Conflict Identification and Mapping**
   (a) 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 $A \to \alpha_{\uparrow}$ and $B \to \beta_{\uparrow}$ when the lookahead token is `@`.
   (b) Give a name to the productions: 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 `eyapp` productions can have *names* and *labels*. In `eyapp` names and labels can be explicitly given using the directive `%name`, using the following syntax:

$$\texttt{\%name}\ \texttt{:rA}\ A \to \alpha$$
$$\texttt{\%name}\ \texttt{:rB}\ B \to \beta$$

   (c) Give a symbolic name to the conflict. In this case we choose `isAorB` as name of the conflict.

(d) 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 :rB` $B \to \beta$ `%PREC IsAorB`

2. **Writing the Conflict Handler**

(a) 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' ); }
}
```

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.
Inside a conflict handler the Perl default variable `$_` refers to the full input text and `$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 token argument `'@'` is optional. If omitted, the set of conflictive tokens will be used.

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 1d and 2a change slightly:

1d'. Again, we must give a symbolic name to $A \to \alpha$ and mark with the new `%PREC` directive the places where the conflict occurs:

$$\begin{array}{rl} \texttt{\%name :rA} & A \to \alpha \ \ \texttt{\%PREC IsAorB} \\ & B \to \beta \ \ \texttt{\%PREC IsAorB} \ \ \gamma \end{array}$$

2a'. Now the conflict handler calls the `YYSetShift` method to set the `shift` action:

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

The token argument `'@'` of `YYSetShift` is optional. If omitted, the set of conflictive tokens is used instead.

In order to clarify the use of PPCR we will address three different kind of conflicts:

- A simple case of dynamically changing the associativity to show the use of the `%conflict` directive (section 3)
- The classical subrange/enum Pascal conflict [1, p. 21] presented in section 4 depicts the use of preparsing, the `%explore` directive and some details of the `eyapp` compiler
- The parsing of a non LR(k) unambiguous grammar (section 5)

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:

[1] For the full examples used in this paper, see [8]

```
$ 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

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

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

```
%%
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.

4. Nested Parsing

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*' [1, p. 21]) where it is used as a paradigmatic example of when to switch to the GLR engine [1]. 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) [12] 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`, making room for the generation of declarations more difficult to parse as:

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

Here is our modification of the vastly simplified subgrammar of Pascal type declarations found in [1].

```
%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
    |  ID
    |  NUM
;
```

4.1. Identifying the problem

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 4 (id_list)

    '*' reduce using rule 12 (expr)
    '+' reduce using rule 12 (expr)
    '-' 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
    | id_list ',' ID
;
expr : '(' expr ')'
    | %name ID:RANGE
      ID                        %PREC rangeORenum
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | expr ',' expr
    | NUM
;
%%
```

4.2. Pre-parsing the Incoming Input

To find which production applies we will pre-parse the input at the point where a range or a enumerated type is expected. To achieve it, we introduce an auxiliary syntactic variable `Lookahead` marking the point where the nested parsing starts:

```
type_decl : 'type' ID '=' type ';' ;

type :
      %name ENUM
      Lookahead '(' id_list ')'
    | %name RANGE
      Lookahead expr '..' expr
;
```

The semantic action associated with `Lookahead` is to check if the incoming input matches a range:

```
Lookahead: /* empty */
  $is_range = $self->YYPreParse('range');
;
```

The call to the method `$_[0]->YYPreParse('range')` uses a `range` parser to parse the input from the current position. It returns true if the `range` parser finds a conformant substring starting at that point.

The `range` parser recognizes the language defined by the subgrammar:

```
range: expr '..' expr ';'
```

where the definition of `expr` is as in the previous grammar.

The conflict handler in the head section decides which production must be used in terms of the value of `$is_range`:

```
%{
  my $is_range = 0;
%}
%conflict rangeORenum {
 if ($is_range)
  { $self->YYSetReduce('ID:RANGE'); }
 else
  { $self->YYSetReduce('ID:ENUM'); }
}
```

The `eyapp` compiler uses these token definitions

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

to automatically generate the lexical analyzer. The rest of the head section of the grammar set the classic static priorities for the arithmetic operators:

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

4.3. Compiling the Grammar

To produce the parser, we start compiling the auxiliary grammar:

```
$ eyapp -P range.eyp
```

The `-P` option used when compiling `range` tells `eyapp` to produce parsing tables that will accept if a prefix of the input belongs to the language generated by the `range` grammar[2]. We then proceed to compile the full grammar[3]:

```
$ eyapp -TC enumvsrange.eyp
```

The `eyapp` compiler provides a default `main` which will be used if no `main` is provided. The default `main` accepts several command line arguments:

```
$ ./enumvsrange.pm -t -i -m 1 -c 'type e = (x, y, z);'
```

[2] By default, the generated parser only accepts if the full input conforms to the grammar

[3] Option `-T` tells the compiler to insert semantic actions in order to produce the syntax tree. Option `-C` is used to generate an executable

Option `-t` tells the `main` to print the result returned by the parser: a description of the syntax tree will be printed. Options `-i` and `-m 1` control the way the syntax tree is shown. Option `-c` is followed by the input for the parser. It indicates that the input is given in the command line. The execution of the former command produces the following output:

```
typeDecl_is_type_ID_type(
  TERMINAL[e],
  ENUM(
    idList_is_idList_ID(
      idList_is_idList_ID(ID(TERMINAL[x]), TERMINAL[y]),
      TERMINAL[z])))
```

4.4. The `%explore` Directive

In the previous grammar we explicitly introduced a new syntactic variable `Lookahead` to set the point for nested parsing. The `eyapp` programmer can use the

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

directive inside the head section to declare the code in charge of the nested parsing:

```
%explorer rangeORenum {
            $is_range = $_[0]->YYPreParse('range');
}
```

Then the point where the exploration starts is marked inside the grammar body using the `%conflictname?` syntax:

```
type :
      %name ENUM
      %rangeORenum? '(' id_list ')'
    | %name RANGE
      %rangeORenum? expr '..' expr
;
```

The `eyapp` compiler will mimic the technique outlined in the previous section, creating a new syntactic variable, let us call it `Lh`, whose only empty production has as associated semantic action the code defined in the `%explorer` directive:

```
Lh: /* empty */ { $is_range = $_[0]->YYPreParse('range') }
```

The points where the `%rangeORenum?` directive appears are substituted by that variable:

```
type :
      %name ENUM
      Lh '(' id_list ')'
    | %name RANGE
      Lh expr '..' expr
;
```

forcing the execution of the exploration code at that points.

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 by packrat parsing algorithms [2].

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

Though it is straightforward to find equivalent LL(1) and LR(1) grammars (the language is even regular: `/x(xx)*/`), even GLR [11] and Backtrack LR parsers [5] 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. To make the problem more difficult and more representative, let us assume `x` is not a token but defines the language of the arithmetic expressions.

The challenge is to make the parser work *without changing* the grammar. As in the previous example we start identifying the conflict - which we name `isInTheMiddle` -, labelling as `:MIDx` the reduction item and marking the exploration point:

```
%token NUM = /(\d+)/
%token OP  = /([-+*\/])/

%%
T: %isInTheMiddle? S ;

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

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

The exploration code uses the auxiliary parser `ExpList` to compute the number of `x`s in the list. Variable `$nxr` is then used to store the mid position:

```
%explorer isInTheMiddle {
   ($nxr) = $self->YYPreParse('ExpList');
   $nxr = int ($nxr/2);
}
```

When `YYPreParse` is called in a list context like above - observe the parenthesis around `$nxr` - it returns the semantic value computed by `ExpList`. The `ExpList.eyp` grammar computes the number of `x`s:

```
%%
S: $S x  { $S + 1 } |  x  { 1 }  ;
%%
```

Where the definition of `x` is as in the previous grammar.

The conflict solver code is quite simple: it keeps the position of the current `x` inside the state/persistent variable `$nxs`. The reduction is called when the middle point is reached:

```
%conflict isInTheMiddle {
  state $nxs = 0;

  $nxs++;
  if ($nxs == $nxr+1) {
    $self->YYSetReduce(':MIDx' );
    $nxr = $nxs = 0;
  }
  else { $self->YYSetShift() }
}
```

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 other alternatives. 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 more effort than branching methods like GLR or backtracking LR. With

some effort, the PPCR methodology can be extended to be merged with GLR and backtracking LR, allowing for a mixed exploration that uses both branching (GLR) and correct pruning (PPCR). This research line seems worth to explore.

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 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 [9], 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 such unambiguous grammar is hard or impossible to find. As future work, we intend to address the building of tools assisting the process of conflict identification and conflict removal without modifying the original grammar.

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. Donnelly, C., Stallman, R.M.: Bison: the yacc-compatible parser generator. Technical report, Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139 (2010)
2. Ford, B.: Functional pearl: Packrat parsing: Simple, powerful, lazy, linear time. Massachusetts Institute of Technology. Cambridge, MA (2002)
3. Johnson, S.C.: Yacc: Yet another compiler compiler. AT&T Bell Laboratories Technical Report July 31, 1978 2, 353–387 (1979)
4. Mcpeak, S.: Elkhound: A fast, practical GLR parser generator (2004), [Online]. Available: `http://scottmcpeak.com/elkhound/`
5. Merrill, G.H.: Parsing Non-LR(k) Grammars with Yacc. Software, Practice and Experience 23(8), 829–850 (1993)
6. Randal, A., Sugalski, D., Totsch, L.: Perl 6 and Parrot Essentials. O'Reilly Media (2004)
7. Rodríguez-León, C.: Parse::Eyapp Manuals (2007), [Online]. Available at CPAN: `http://search.cpan.org/dist/Parse-Eyapp/`
8. Rodríguez-León, C., García-Forte, L.: Grammar Repository (2010), [Online]. Available at google-code: `http://code.google.com/p/grammar-repository/`
9. Teixeira Passos, L., Bigonha, M.A., Bigonha, R.: A methodology for removing LALR(k) conflicts. In: Journal of Universal Computer Science. pp. 735–752 (2007)
10. Thurston, A.D., Cordy, J.R.: A backtracking LR algorithm for parsing ambiguous context-dependent languages. In: 2006 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2006). pp. 39–53. Toronto (2006)

11. Tomita, M.: The generalized LR parser/compiler - version 8.4. In: Proceedings of International Conference on Computational Linguistics (COLING'90). pp. 59–63. Helsinki, Finland (1990)
12. van der Veen, V.: Extended pascal iso 10206:1990, [Online]. Available: http://www.standardpascal.org/iso10206.txt
13. Wall, L., Christiansen, T., Schwartz, R.: *Programming Perl. O'Reilly & Associates (1996)*

C. Rodriguez-Leon is a full professor of Computer Science at Universidad de La Laguna, Spain. He received his Diploma in Mathematics and his Doctorate (Ph. D.) in 1978 and 1987, respectively, both from the same University. He is in the Editorial Board of journals like Parallel Computing, International Journal of Computational Science and Engineering (IJCSE), etc. and has been in the Program Committee of several parallel computing conferences including EuroPVM/MPI, HeteroPar, EuroPar, HLPP, CIT, ICA3PP, CACIC, INFORUM, WGISD, etc. His research interest includes Parallel Algorithms, Evolutionary Computation, Combinatorial Optimization, Distributed Computing, Language Processing and High Level Programming.

L. Garcia-Forte is a Ph. D. student of the Department of Statistics, Operation Research and Computation at Universidad de La Laguna, Spain. He received his Diploma in Computer Engineering in 1997. His main research interests focus on Programming Languages and Parallel Systems.

Received: November 16, 2010; Accepted: April 19, 2011.