

1 NAME

Parse::Eyapp::debuggingtut - Solving ambiguities and fixing lexical, syntactic and semantic errors

2 INTRODUCTION

The sources of error when programming with `eyapp` are many and various. Some of them are minor, as having a nonterminal without production rules or a terminal that is never produced by the lexical analyzer. These kind of errors can be caught with the help of the `%strict` directive.

In the following sections we will discuss three main kind of errors that correspond to three development stages:

- Conflict errors:

Conflicts with the grammar: the grammar is ambiguous or is not clear - perhaps due to the fact that `eyapp` uses only a lookahead symbol - which sort of tree must be built for some inputs

- Tree building errors:

There are no conflicts but the parser does not build the syntax tree as expected. May be it rejects correct sentences or accepts incorrect ones. Or may be it accepts correct ones but the syntax tree has not the shape we want (i.e. we have a precedence problem).

- Semantic errors:

We have solved the conflicts and trees are satisfactory but we have errors inside the semantic actions.

Each time you discover an error write a test that covers that error. Section TREE EQUALITY deals with the problem of checking if the generated abstract syntax tree has the correct shape and attributes.

As Andreas Zeller points out in his article "Beautiful Debugging" finding the causes of a failing program must follow the scientific method:

1. Observe the failure (there are conflicts or ambiguity, there are precedence problems, there are semantic errors, the output is wrong)
2. Guess a hypothesis for the failure (if necessary use `eyapp -v` option, `yydebug`, the Perl debugger, etc. to build the hypothesis). If you use continuous testing it is likely related with the recently written code.
3. Based on your hypothesis make predictions
4. Using appropriate input tests and the available tools (`eyapp -v` option, `yydebug`, the Perl debugger, etc.) see if your predictions hold. Reject your hypothesis if they don't hold.
5. Repeat the last two steps until your hypothesis is confirmed. The hypothesis then becomes a theory.
6. Convert the knowledge and informal tests developed during this process in a formal test that covers the failure

3 THE %strict DIRECTIVE

By default, identifiers appearing in the rule section will be classified as terminal if they don't appear in the left hand side of any production rules.

The directive `%strict` forces the declaration of all tokens. The following `eyapp` program issues a warning:

```
pl@nereida:~/LEyapp/examples/eyapplanguageref$ cat -n buggyapp2.eyp
 1 %strict
 2 %%
 3 expr: NUM;
 4 %%
pl@nereida:~/LEyapp/examples/eyapplanguageref$ eyapp buggyapp2.eyp
Warning! Non declared token NUM at line 3 of buggyapp2.eyp
```

To keep silent the compiler declare all tokens using one of the token declaration directives (`%token`, `%left`, etc.)

```

pl@nereida:~/LEyapp/examples/eyapplanguageref$ cat -n bugyapp3.eyp
  1 %strict
  2 %token NUM
  3 %%
  4 expr: NUM;
  5 %%
pl@nereida:~/LEyapp/examples/eyapplanguageref$ eyapp bugyapp3.eyp
pl@nereida:~/LEyapp/examples/eyapplanguageref$ ls -ltr | tail -1
-rw-r--r-- 1 pl users 2395 2008-10-02 09:41 bugyapp3.pm

```

It is a good practice to use `%strict` at the beginning of your grammar.

4 CONFLICTS AND AMBIGUITIES

Understanding Priorities

Token and production priorities are used to solve conflicts. Recall the main points of yacc-like parsers related to priorities:

- The directives

```

%left
%right
%nonassoc

```

can be used in the head section to declare the priority of a token

- The later the declaration line the higher the priority
- The precedence of a production rule (right hand side) is the precedence of the last token in the right hand side
- In a shift-reduce conflict the default action is to shift. This action can be changed if the production and the token have explicit priorities
- If the precedence of the production rule is higher the shift-reduce conflict is solved in favor of the reduction
- If the precedence of the token is higher the shift-reduce conflict is solved in favor of the shift
- If the precedence of the token is the same than the precedence of the rule, and is left the shift-reduce conflict is solved in favor of the reduction
- If the precedence of the token is the same than the precedence of the rule, and is right the shift-reduce conflict is solved in favor of the shift
- If the precedence of the token is the same than the precedence of the rule, and is nonassoc the presence of a shift-reduce conflict means an error. This is used to describe operators, like the operator `.LT.` in FORTRAN, that may not associate with themselves. That is, because

```
A .LT. B .LT. C
```

is invalid in FORTRAN, `.LT.` would be described with the keyword `%nonassoc` in `eyapp`.

- The default precedence of a production can be changed using the `%prec TOKEN` directive. Now the rule has the precedence and associativity of the specified `TOKEN`.

The program `Precedencia.eyp` illustrates the way priorities work in `eyapp`:

```

pl@europa:~/LEyapp/examples/debuggingtut$ eyapp -c Precedencia.eyp
%token NUM
%left '@'
%right '&' dummy
%tree
%%

```

```

list:
  | list '\n'
  | list e
;
e:
  %name NUM
  NUM
  | %name AMPERSAND
  e '&' e
  | %name AT
  e '@' e %prec dummy
;

%%

```

See an execution:

```

pl@europa:~/LEyapp/examples/debuggingtut$ ./Precedencia.pm
Expressions. Press CTRL-D (Unix) or CTRL-Z (Windows) to finish:
2@3@4
2@3&4
2&3@4
2&3&4
<CTRL-D>
AT(AT(NUM(TERMINAL[2]),NUM(TERMINAL[3])),NUM(TERMINAL[4]))
AT(NUM(TERMINAL[2]),AMPERSAND(NUM(TERMINAL[3]),NUM(TERMINAL[4])))
AT(AMPERSAND(NUM(TERMINAL[2]),NUM(TERMINAL[3])),NUM(TERMINAL[4]))
AMPERSAND(NUM(TERMINAL[2]),AMPERSAND(NUM(TERMINAL[3]),NUM(TERMINAL[4])))

```

See if you are able to understand the output:

- 2@3@4: The phrase is interpreted as (2@3)@4 since the rule e '@' e has the precedence of the token dummy which is stronger than the priority of token @. The conflict is solved in favor of the reduction
- 2@3&4: The rule e '@' e has the precedence of dummy which is the same as the token &. The associativity decides. Since they were declared %right the conflict is solved in favor of the shift. The phrase is interpreted as 2@(3&4)
- 2&3@4: The rule e '&' e has more precedence than the token @. The phrase is interpreted as (2&3)@4
- 2&3&4: Both the rule and the token have the same precedence. Since they were declared %right, the conflict is solved in favor of the shift. The phrase is interpreted as 2&(3&4)

An eyapp Program with Errors

The following simplified eyapp program has some errors. The generated language is made of lists of declarations (D stands for declaration) followed by lists of sentences (S stands for statement) separated by semicolons:

```

pl@nereida:~/LEyapp/examples/debuggingtut$ cat -n Debug.eyp
 1  %{
 2  =head1 SYNOPSIS
 3
 4  This grammar has an unsolved shift-reduce conflict.
 5
 6  Be sure C<DebugTail.pm> is reachable.
 7  Compile it with
 8
 9      eyapp -b '' Debug.eyp
10
11  See the C<Debug.output> file generated.
12  Execute the generated modulino with:
13
14      ./Debug.pm -d # to activate debugging

```

```

15     ./Debug.pm -h # for help
16
17 The generated parser will not recognize any input, since its shifts forever.
18 Try input C<'D; D; S'>.
19
20 =head1 See also
21
22     http://search.cpan.org/perldoc?Parse::Eyapp::debuggingtut
23
24     Debug1.eyp Debug2.eyp DebugLookForward.eyp DebugDynamicResolution.eyp
25
26 =cut
27
28 our $VERSION = '0.01';
29 use base q{DebugTail};
30
31 %}
32
33 %token D S
34
35 %%
36 p:
37     ds ';' ss
38     | ss
39 ;
40
41 ds:
42     D ';' ds
43     | D          /* this production is never used */
44 ;
45
46 ss:
47     S ';' ss
48     | S
49 ;
50
51 %%
52
53 __PACKAGE__->main('Provide a statement like "D; D; S" and press <CR><CTRL-D>: ') unless caller;

```

Focusing in the Grammar

Sometimes the presence of actions, attribute names and support code makes more difficult the readability of the grammar. You can use the `-c` option of `eyapp`, to see only the syntactic parts:

```

$ eyapp -c examples/debuggingtut/Debug.eyp
%token D S

%%

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

```

\$

It is clear now that the language generated by this grammar is made of non empty sequences of D followed by non empty sequences of <S> separated by semicolons.

Detecting Conflicts

When compiling this grammar, eyapp produces a warning message announcing the existence of a conflict:

```
pl@nereida:~/LEyapp/examples$ eyapp Debug.eyp
1 shift/reduce conflict (see .output file)
State 4: shifts:
  to state    8 with ';' ;'
```

Studying the .output file

The existence of warnings triggers the creation of a file Debug.output containing information about the grammar and the syntax analyzer.

Let us see the contents of the Debug.output file:

```
pl@nereida:~/LEyapp/examples$ cat -n Debug.output
 1 Warnings:
 2 -----
 3 1 shift/reduce conflict (see .output file)
 4 State 4: shifts:
 5   to state    8 with ';' ;'
 6
 7 Conflicts:
 8 -----
 9 State 4 contains 1 shift/reduce conflict
10
11 Rules:
12 -----
13 0:   $start -> p $end
14 1:   p -> ds ';' ss
15 2:   p -> ss
16 3:   ds -> D ';' ds
17 4:   ds -> D
18 5:   ss -> S ';' ss
19 6:   ss -> S
20
21 States:
22 -----
23 State 0:
24
25     $start -> . p $end      (Rule 0)
26
27     D      shift, and go to state 4
28     S      shift, and go to state 1
29
30     p      go to state 2
31     ss     go to state 3
32     ds     go to state 5
33
.. .....
55 State 4:
56
57     ds -> D . ';' ds      (Rule 3)
58     ds -> D .           (Rule 4)
59
60     ';'    shift, and go to state 8
61
```

```

62      ';'      [reduce using rule 4 (ds)]
63
.. .....
84 State 8:
85
86      ds -> D ';' . ds      (Rule 3)
87
88      D      shift, and go to state 4
89
90      ds      go to state 11
91
.. .....
112 State 12:
113
114      p -> ds ';' ss .      (Rule 1)
115
116      $default      reduce using rule 1 (p)
117
118
119 Summary:
120 -----
121 Number of rules      : 7
122 Number of terminals : 4
123 Number of non-terminals : 4
124 Number of states   : 13

```

The parser generated by `Parse::Eyapp` is based on a *deterministic finite automaton*. Each state of the automaton *remembers* what production rules are candidates to apply and what have been seen from the right hand side of the production rule. The problem, according to the warning, occurs in state 4. State 4 contains:

```

55 State 4:
56
57      ds -> D . ';' ds      (Rule 3)
58      ds -> D .      (Rule 4)
59
60      ';'      shift, and go to state 8
61
62      ';'      [reduce using rule 4 (ds)]
63

```

An state is a set of production rules with a marker (the dot in rules 3 and 4) somewhere in its right hand side. If the parser is in state 4 is because the production rules `ds -> D ';' ds` and `ds -> D` are potential candidates to build the syntax tree. That they will win or not depends on what will happen next when more input is processed.

The dot that appears on the right hand side means *position* in our guessing. The fact that `ds -> D . ';' ds` is in state 4 means that if the parser is in state 4 we have already seen `D` and we expect to see a semicolon followed by `ds` (or something derivable from `ds`). If such thing happens this production will be the right one (will be the *handle* in the jargon). The comment

```

60      ';'      shift, and go to state 8

```

means that if the next token is a semicolon the next state will be state 8:

```

84 State 8:
85
86      ds -> D ';' . ds      (Rule 3)
87
88      D      shift, and go to state 4
89
90      ds      go to state 11

```

As we see state 8 has the item `ds -> D ';' . ds` which means that we have already seen a D and a semicolon.

The fact that `ds -> D .` is in state 4 means that we have already seen D and since the dot is at the end of the rule, this production can be the right one, even if a semicolon is just waiting in the input. An example that it will be correct to "reduce" by the rule `ds -> D .` in the presence of a semicolon is given by the input `D ; S`. A rightmost derivation for such input is:

```
p => ds ; ss => ds ; S => D ; S
```

that is processed by the LALR(1) algorithm following this sequence of actions:

rule	read	input
		D ; S \$
	D	; S \$
ds->d	ds	; S \$
	ds ;	S \$
	ds ; S	\$
ss->s	ds ; ss	\$
p->ds;ss	p	

Since it is correct to reduce in some cases by the production `ds -> D .` and others in which is correct to shift the semicolon, `eyapp` complains about a shift/reduce conflict with `';' .` State 4 has two rules that compete to be the right one:

```
pl@nereida:~/LEyapp/examples$ eyapp Debug.eyp
1 shift/reduce conflict (see .output file)
```

We can guess that the right item (the rules with the dot, i.e. the states of the automaton are called LALR(0) items in the yacc jargon) is `ds -> D . ';' ds` and *shift to state 8* consuming the semicolon, expecting to see something derivable from `ds` later or guess that `ds -> D .` is the right LR(0) item and *reduce* for such rule. This is the meaning of the comments in state 4:

```
60      ';'      shift, and go to state 8
61
62      ';'      [reduce using rule 4 (ds)]
```

To illustrate the problem let us consider the phrases `D;S` and `D;D;S`.

For both phrases, after consuming the D the parser will go to state 4 and the current token will be the semicolon.

For the first phrase `D;S` the correct decision is to use rule 4 `ds -> D` (to *reduce* in the jargon). For the second phrase `D;D;S` the correct decision is to follow rule 3 `ds -> D . ';' ds`.

The parser generated by `eyapp` would be able to know which rule is correct for each case if it were allowed to look at the token after the semicolon: if it is a `S` is rule 4, if it is a `D` is rule 3. But the parsers generated by `Eyapp` do not lookahead more than the next token (this is what the "1" means when we say that `Parse::Eyapp` parsers are LALR(1)) and therefore is not in condition to decide which production rule applies.

Unfortunately this is the sort of conflict that can't be solved by assigning priorities to the productions and tokens as it was done for the calculator example in `Parse::Eyapp::eyappintro`. If we run the analyzer it will refuse to accept correct entries like `D;D;S`:

```
pl@europa:~/LEyapp/examples/debuggingtut$ eyapp -b '' -o debug.pl Debug.eyp
1 shift/reduce conflict (see .output file)
```

```
State 4: shifts:
```

```
  to state 8 with ';' .
```

```
pl@europa:~/LEyapp/examples/debuggingtut$ ./debug.pl
```

```
D;D;S
```

```
-----
In state 0:
```

```
Stack: [0]
```

```
Need token. Got >D<
```

```
Shift and go to state 4.
-----
```

```

In state 4:
Stack: [0,4]
Need token. Got >;<
Shift and go to state 8.
-----
In state 8:
Stack: [0,4,8]
Need token. Got >D<
Shift and go to state 4.
-----
In state 4:
Stack: [0,4,8,4]
Need token. Got >;<
Shift and go to state 8.
-----
In state 8:
Stack: [0,4,8,4,8]
Need token. Got >S<
Syntax error near input: 'S' line num 1

```

The default parsing action is to shift the token ; giving priority to the production

```
ds -> D . ';' ds
```

over the production

```
ds -> D .
```

Since no `ds` production starts with `S`, the presence of `S` is (erroneously) interpreted as an error.

The Importance of the FOLLOW Set

You may wonder why the productions

```

ss:
    S ';' ss
    | S
;

```

do not also produce a shift-reduce conflict with the semicolon. This is because the reduction by `ss -> S` always corresponds to the last `S` in a derivation:

```
ss => S ; ss => S ; S ; ss => S ; S ; S
```

and thus, the reduction by `ss -> S` only occurs in the presence of the `end of input` token and never with the semicolon. The FOLLOW set of a syntactic variable is the set of tokens that may appear next to such variable in some derivation. While the semicolon ; is in the FOLLOW of `dd`, it isn't in the FOLLOW of `ss`.

Solving Shift-Reduce Conflicts by Factorizing

To solve the former conflict the `Eyapp` programmer has to reformulate the grammar modifying priorities and reorganizing the rules. Rewriting the recursive rule for `ds` to be left recursive solves the conflict:

```

pl@nereida:~/LEyapp/examples/debuggingtut$ sed -ne '/^ds:\/,\/^;/p' Debug1.eypp | cat -n
1 ds:
2     %name D2
3     ds ';' D
4     | %name D1
5     D
6     ;

```

Now, for any phrase matching the pattern `D ; ...` the action to build the tree is to reduce by `ds -> D`. The rightmost reverse derivation for `D;D;S` is:

Derivation		Tree
D;D;S <= ds;D;S <= ds;S <= ds;ss <= p		p(ds(ds(D),',',D),',',ss(S))

while the rightmost reverse derivation for D;S is:

Derivation		Tree
D;S <= ds;S <= ds;ss <= p		p(ds(D),',',ss(S))

When we recompile the modified grammar no warnings appear:

```
pl@nereida:~/LEyapp/examples$ eyapp Debug1.eyp
pl@nereida:~/LEyapp/examples$
```

Solving Shift-Reduce Conflicts By Looking Ahead

The problem here is that Eyapp/Yapp/Yacc etc. produce LALR(1) parsers. They only look the next token. We can decide how to solve the conflict by rewriting the lexical analyzer to peer forward what token comes after the semicolon: it now returns SEMICOLONS if it is an S and SEMICOLOND if it is an D. Here is a solution based in this idea:

```
pl@nereida:~/LEyapp/examples/debuggingtut$ cat -n DebugLookForward.eyp
 1 /*VIM: set ts=2 */
 2 %{
 3 =head1 SYNOPSIS
 4
 5 See
 6
 7     http://search.cpan.org/perldoc?Parse::Eyapp::debuggingtut
 8     file DebugLookForward.eyp
 9
10 This grammar fixes the conflicts an bugs in Debug.eyp and Debug1.eyp
11
12 Be sure C<DebugTail.pm> is reachable
13 compile it with
14
15     eyapp -b '' DebugLookForward.eyp
16
17 execute the generated modulino with:
18
19     ./DebugLookForward.pm -t
20
21 =head1 See also
22
23     Debug.eyp Debug1.eyp Debug2.eyp
24
25 =cut
26
27 our $VERSION = '0.01';
28 use base qw{DebugTail};
29
30 %}
31
32 %token D S
33 %syntactic token SEMICOLONS SEMICOLOND
34
35 %tree
36
37 %%
38 p:
39     %name P
```

```

40     ds SEMICOLONS ss
41   | %name SS
42     ss
43   ;
44
45 ds:
46     %name D2
47     D SEMICOLOND ds
48   | %name D1
49     D
50   ;
51
52 ss:
53     %name S2
54     S SEMICOLONS ss
55   | %name S1
56     S
57   ;
58
59 %%
60
61 __PACKAGE__->lexer(
62   sub {
63     my $self = shift;
64
65     for (${$self->input()}) {
66       s{^\s+}{ } and $self->tokenline($1 =~ tr{\n}{ });
67       return ('',undef) unless $_;
68
69       return ($1,$1) if s/^[sSDd]//;
70       return ('SEMICOLOND', 'SEMICOLOND') if s/^\s*D/D/;
71       return ('SEMICOLONS', 'SEMICOLONS') if s/^\s*S/S/;
72       die "Syntax error at line num ${$self->tokenline()}: ${substr($_,0,10)}\n";
73     }
74     return ('',undef);
75   }
76 );
77
78 __PACKAGE__->main unless caller();

```

5 ERRORS DURING TREE CONSTRUCTION

Though Debug1.pm seems to work:

```

pl@nereida:~/LEyapp/examples/debuggingtut$ ./Debug1.pm -t
Try first "D;S" and then "D; D; S" (press <CR><CTRL-D> to finish): D;D;S
P(D2(D1(TERMINAL[D]),TERMINAL[D]),S1(TERMINAL[S]))

```

There are occasions where we observe an abnormal behavior:

```

pl@nereida:~/LEyapp/examples/debuggingtut$ ./Debug1.pm -t
Try first "D;S" and then "D; D; S" (press <CR><CTRL-D> to finish):
D

;

D

;
S
Syntax error near end of input line num 3. Expecting (;)

```

We can activate the option `yydebug => 0xF` in the call to the parser method `YYParse`. The integer parameter `yydebug` of `new` and `YYParse` controls the level of debugging. Different levels of verbosity can be obtained by setting the bits of this argument. It works as follows:

```

/=====\
| Bit Value | Outputs |
|-----+-----|
| 0x01      | Token reading (useful for Lexer debugging) |
|-----+-----|
| 0x02      | States information |
|-----+-----|
| 0x04      | Driver actions (shifts, reduces, accept...) |
|-----+-----|
| 0x08      | Parse Stack dump |
|-----+-----|
| 0x10      | Error Recovery tracing |
\=====/

```

Let us see what happens when the input is `D;S`. We have introduced some white spaces and carriage returns between the terminals:

```

pl@nereida:~/LEyapp/examples/debuggingtut$ ./Debug1.pm -d
Try first "D;S" and then "D; D; S" (press <CR><CTRL-D> to finish):
D

;

D

;
S

```

```

-----
In state 0:
Stack: [0]
Need token. Got >D<
Shift and go to state 4.
-----
In state 4:
Stack: [0,4]
Don't need token.
Reduce using rule 4 (ds --> D): Back to state 0, then go to state 5.
-----
In state 5:
Stack: [0,5]
Need token. Got ><
Syntax error near end of input line num 3. Expecting (;)

```

What's going on? After reading the carriage return

```
Need token. Got >D<
```

the parser receives an end of file. £Why?. Something is going wrong in the communications between lexical analyzer and parser. Let us review the lexical analyzer:

```

pl@nereida:~/LEyapp/examples/debuggingtut$ sed -ne '/lexer/,/^)/p' Debug1.eyy | cat -n
1  __PACKAGE__->lexer(
2    sub {
3      my $self = shift;
4
5      for (${$self->input()}) { # contextualize
6        s{~(\s)}{} and $self->tokenline($1 =~ tr{\n}{});
7
8        return ('', undef) unless $_;

```

```

9         return ($1, $1) if s/^(.)//;
10      }
11      return ('', undef);
12  }
13 );

```

The error is at line 6. Only a single white space is eaten! The second carriage return in the input does not match lines 8 and 9 and the contextualizing `for` finishes. Line 11 then unconditionally returns the `('',undef)` signaling the end of input.

The grammar in file `Debug2.eypp` fixes the problem: Now the analysis seems to work for this kind of input:

```

pl@nereida:~/LEyapp/examples/debuggingtut$ eyapp -b '' Debug2.eypp
pl@nereida:~/LEyapp/examples/debuggingtut$ ./Debug2.pm -t -d
Provide a statement like "D; D; S" and press <CR><CTRL-D>:
D
;
D
;
S
-----
In state 0:
Stack:[0]
Need token. Got >D<
Shift and go to state 4.
-----
In state 4:
Stack:[0,4]
Don't need token.
Reduce using rule 4 (ds --> D): Back to state 0, then go to state 5.
-----
In state 5:
Stack:[0,5]
Need token. Got >;<
Shift and go to state 8.
-----
In state 8:
Stack:[0,5,8]
Need token. Got >D<
Shift and go to state 11.
-----
In state 11:
Stack:[0,5,8,11]
Don't need token.
Reduce using rule 3 (ds --> ds ; D): Back to state 0, then go to state 5.
-----
In state 5:
Stack:[0,5]
Need token. Got >;<
Shift and go to state 8.
-----
In state 8:
Stack:[0,5,8]
Need token. Got >S<
Shift and go to state 1.
-----
In state 1:
Stack:[0,5,8,1]
Need token. Got ><
Reduce using rule 6 (ss --> S): Back to state 8, then go to state 10.

```

```

-----
In state 10:
Stack: [0,5,8,10]
Don't need token.
Reduce using rule 1 (p --> ds ; ss): Back to state 0, then go to state 2.
-----

In state 2:
Stack: [0,2]
Shift and go to state 7.
-----

In state 7:
Stack: [0,2,7]
Don't need token.
Accept.
P(D2(D1(TERMINAL[D]), TERMINAL[D]), S1(TERMINAL[S]))

```

6 THE LR PARSING ALGORITHM: UNDERSTANDING THE OUTPUT OF yydebug

The `YYParse` methods implements the generic LR parsing algorithm. It very much works `Parse::Yapp::YYParse` and as `yacc/bison yyparse`. It accepts almost the same arguments as `Class->new` (Being `Class` the name of the generated class).

The parser uses two tables and a stack. The two tables are called the *action* table and the *goto* table. The stack is used to keep track of the states visited.

At each step the generated parser consults the *action* table and takes one decision: To shift to a new state consuming one token (and pushing the current state in the stack) or to reduce by some production rule. In the last case the parser pops from its stack as many states as symbols are on the right hand side of the production rule. Here is a Perl/C like pseudocode summarizing the activity of `YYParse`:

```

1  my $parser = shift; # The parser object
2  push(@stack, $parser->{startstate});
3  $b = $parser->YYLexer(); # Get the first token
4  FOREVER: {
5      $s = top(0); # Get the state on top of the stack
6      $a = $b;
7      switch ($parser->action[$s->state][$a]) {
8          case "shift t" :
9              my $t;
10             $t->{state} = t;
11             $t->{attr} = $a->{attr};
12             push($t);
13             $b = $parser->YYLexer(); # Call the lexical analyzer
14             break;
15         case "reduce A->alpha" :
16             # Call the semantic action with the attributes of the rhs as args
17             my $semantic = $parser->Semantic{A ->alpha}; # The semantic action
18             my $r;
19             $r->{attr} = $semantic->($parser, top(|alpha|-1)->attr, ... , top(0)->attr);
20
21             # Pop as many states as symbols on the rhs of A->alpha
22             pop(|alpha|);
23
24             # Goto next state
25             $r->{state} = $parser->goto[top(0)][A];
26             push($r);
27             break;
28         case "accept" : return (1);
29         default : $parser->YYError("syntax error");
30     }
31     redo FOREVER;

```

Here $|\alpha|$ stands for the length of α . Function $\text{top}(k)$ returns the state in position k from the top of the stack, i.e. the state at depth k . Function $\text{pop}(k)$ extracts k states from the stack. The call $\text{\$state}\rightarrow\text{attr}$ returns the attribute associated with $\text{\$state}$. The call $\text{\$parser}\rightarrow\text{Semantic}\{\text{A}\rightarrow\alpha\}$ returns the semantic action associated with production $\text{A}\rightarrow\alpha$.

Let us see a trace for the small grammar in `examples/debuggingtut/aSb.yyp`:

```
pl@nereida:~/LEyapp/examples$ /usr/local/bin/paste.pl aSb.yyp aSb.output | head -5
%%                                     | Rules:
S:           { print "S -> epsilon\n" } | -----
      | 'a' S 'b' { print "S -> a S b\n" } | 0:   $start -> S $end
;           |                               | 1:   S -> /* empty */
%%           | 2:   S -> 'a' S 'b'
```

The tables in file `aSb.output` describe the actions and transitions to take:

```
pl@nereida:~/LEyapp/examples$ cat -n aSb.output
. ....
7 States:
8 -----
9 State 0:
10
11     $start -> . S $end      (Rule 0)
12
13     'a'      shift, and go to state 2
14
15     $default      reduce using rule 1 (S)
16
17     S          go to state 1
18
19 State 1:
20
21     $start -> S . $end      (Rule 0)
22
23     $end      shift, and go to state 3
24
25 State 2:
26
27     S -> 'a' . S 'b'      (Rule 2)
28
29     'a'      shift, and go to state 2
30
31     $default      reduce using rule 1 (S)
32
33     S          go to state 4
34
35 State 3:
36
37     $start -> S $end .      (Rule 0)
38
39     $default      accept
40
41 State 4:
42
43     S -> 'a' S . 'b'      (Rule 2)
44
45     'b'      shift, and go to state 5
46
47 State 5:
48
49     S -> 'a' S 'b' .      (Rule 2)
```

```

50
51          $default          reduce using rule 2 (S)
52
53
54 Summary:
55 -----
56 Number of rules           : 3
57 Number of terminals       : 3
58 Number of non-terminals   : 2
59 Number of states         : 6

```

When executed with `yydebug` set and input `aabb` we obtain the following output:

```

pl@nereida:~/LEyapp/examples/debuggingtut$ eyapp -b '' -o use_aSb.pl aSb
pl@nereida:~/LEyapp/examples/debuggingtut$ ./use_aSb.pl -d
Provide a statement like "a a b b" and press <CR><CTRL-D>: aabb

```

```

-----
In state 0:
Stack: [0]
Need token. Got >a<
Shift and go to state 2.
-----
In state 2:
Stack: [0,2]
Need token. Got >a<
Shift and go to state 2.
-----
In state 2:
Stack: [0,2,2]
Need token. Got >b<
Reduce using rule 1 (S --> /* empty */): S -> epsilon
Back to state 2, then go to state 4.
-----
In state 4:
Stack: [0,2,2,4]
Shift and go to state 5.
-----
In state 5:
Stack: [0,2,2,4,5]
Don't need token.
Reduce using rule 2 (S --> a S b): S -> a S b
Back to state 2, then go to state 4.
-----

```

As a result of reducing by rule 2 the three last visited states are popped from the stack, and the stack becomes `[0,2]`. But that means that we are now in state 2 seeing a `S`. If you look at the table above being in state 2 and seeing a `S` we go to state 4.

```

-----
In state 4:
Stack: [0,2,4]
Need token. Got >b<
Shift and go to state 5.
-----
In state 5:
Stack: [0,2,4,5]
Don't need token.
Reduce using rule 2 (S --> a S b): S -> a S b
Back to state 0, then go to state 1.
-----
In state 1:
Stack: [0,1]
Need token. Got >><

```

Shift and go to state 3.

```
-----  
In state 3:  
Stack: [0,1,3]  
Don't need token.  
Accept.
```

7 ERRORS INSIDE SEMANTIC ACTIONS

A third type of error occurs when the code inside a semantic action doesn't behave as expected.

The semantic actions are translated in anonymous methods of the parser object. Since they are anonymous we can't use breakpoints as

```
b subname # stop when arriving at sub 'name'  
  
or  
  
c subname # continue up to reach sub 'name''
```

Furthermore the file loaded by the client program is the generated `.pm`. The code in the generated module `Debug.pm` is alien to us - Was automatically generated by `Parse::Eyapp` - and it can be difficult to find where our inserted semantic actions are.

To watch the execution of a semantic action is simple: We use the debugger `f file.eypp` option to switch the viewing filename to our grammar file. The following session uses the example in the directory `examples/Calculator`:

```
pl@nereida:~/LEyapp/examples/Calculator$ perl -wd scripts/calc.pl  
Loading DB routines from perl5db.pl version 1.3  
Editor support available.  
Enter h or 'h h' for help, or 'man perldebug' for more help.
```

```
main::(scripts/calc.pl:8):      Math::Calc->main();  
DB<1> f lib/Math/Calc.eypp  
1      2      3      4      5      6      7      #line 8 "lib/Math/Calc.eypp"  
8  
9:      use base q{Math::Tail};  
10:     my %s; # symbol table
```

Lines 37 to 41 contain the semantic action associated with the production `exp -> VAR` (see file `examples/Calculator/lib`

```
DB<2> 1 37,41  
37:      my $id = $VAR->[0];  
38:      my $val = $s{$id};  
39:      $_[0]->semantic_error("Accessing undefined variable $id at line $VAR->[1].\n")  
40:      unless defined($val);  
41:      return $val;
```

now we set a break at line 37, to see what happens when a non initialized variable is used:

```
DB<3> b 37
```

We issue now the command `c` (continue). The execution continues until line 37 of `lib/Math/Calc.eypp` is reached:

```
DB<4> c  
Expressions. Press CTRL-D (Unix) or CTRL-Z (Windows) to finish:  
a = 2+b # user input  
Math::Calc::CODE(0x191da98)(lib/Math/Calc.eypp:37):  
37:      my $id = $VAR->[0];
```

Now we can issue any debugger commands (like `x`, `p`, etc.) to investigate the internal state of our program and determine what are the reasons of any abnormal behavior.

```

DB<4> n
Math::Calc::CODE(0x191da98)(lib/Math/Calc.eypp:38):
38:         my $val = $s{$id};
DB<4> x $id
0 'b'
DB<5> x %s
empty array

```

8 SOLVING REDUCE-REDUCE CONFLICTS

Reduce-Reduce Conflict: Default rules

Most of the time reduce-reduce conflicts are due to some ambiguity in the grammar, as it is the case for this minimal example:

```

pl@nereida:~/LEyapp/examples/debuggingtut$ sed -ne '/%%/,%%/p' minimalrr.eypp | cat -n
 1  %%
 2  s:
 3      %name S_is_a
 4      'a'
 5      | A
 6  ;
 7  A:
 8      %name A_is_a
 9      'a'
10  ;
11
12  %%

```

In case of a reduce-reduce conflict, *Parse::Eyapp* reduces using the first production in the text:

```

pl@nereida:~/LEyapp/examples/debuggingtut$ eyapp -b '' minimalrr.eypp
1 reduce/reduce conflict
pl@nereida:~/LEyapp/examples/debuggingtut$ ./minimalrr.pm -t
Try "a" and press <CR><CTRL-D>: a
S_is_a

```

If we change the order of the productions

```

pl@nereida:~/LEyapp/examples/debuggingtut$ sed -ne '/%start/,40p' minimalrr2.eypp | cat -n
 1  %start s
 2
 3  %%
 4  A:
 5      %name A_is_a
 6      'a'
 7  ;
 8
 9  s:
10      %name S_is_a
11      'a'
12      | %name A
13      A
14  ;
15  %%

```

the selected production changes:

```

pl@nereida:~/LEyapp/examples/debuggingtut$ eyapp -b '' minimalrr2
1 reduce/reduce conflict
pl@nereida:~/LEyapp/examples/debuggingtut$ ./minimalrr2.pm -t
Try "a" and press <CR><CTRL-D>: a
A(A_is_a)

```

Reduce-Reduce conflicts: typical errors

In this example the programmer has attempted to define a language made of mixed lists IDs and NUMbers :

```
~/LEyapp/examples/debuggingtut$ eyapp -c typicalrr.eyp
%token ID NUM
%tree

%%

s:
    /* empty */
    | s ws
    | s ns
;
ns:
    /* empty */
    | ns NUM
;
ws:
    /* empty */
    | ws ID
;

%%
```

The grammar has several reduce-reduce conflicts:

```
~/LEyapp/examples/debuggingtut$ eyapp -b '' typicalrr.eyp
3 shift/reduce conflicts and 3 reduce/reduce conflicts
```

There are several sources of ambiguity in this grammar:

- Statments like

NUM NUM NUM

are ambiguous. The following two left-most derivations exists:

s ==> ns ns ==> NUM NUM ns => NUM NUM NUM

and

s ==> ns ns ==> NUM ns ==> NUM NUM NUM

the same with phrases like ID ID ID

- The empty word can be generated in many ways. For example:

s => empty

or

s => s ns => s empty => empty

etc.

The generated parser loops forever if feed with a list of identifiers:

```

~/LEyapp/examples/debuggingtut$ ./typicalrr.pm -d
Try inputs "4 5", "a b" and "4 5 a b"(press <CR><CTRL-D> to finish): ab
-----
In state 0:
Stack:[0]
Don't need token.
Reduce using rule 1 (s --> /* empty */): Back to state 0, then go to state 1.
-----
In state 1:
Stack:[0,1]
Need token. Got >ID<
Reduce using rule 4 (ns --> /* empty */): Back to state 1, then go to state 2.
-----
In state 2:
Stack:[0,1,2]
Reduce using rule 3 (s --> s ns): Back to state 0, then go to state 1.
-----
In state 1:
Stack:[0,1]
Reduce using rule 4 (ns --> /* empty */): Back to state 1, then go to state 2.
-----
In state 2:
Stack:[0,1,2]
Reduce using rule 3 (s --> s ns): Back to state 0, then go to state 1.
-----
^C

```

The problem is easily solved designing an equivalent non ambiguous grammar:

```

pl@europa:~/LEyapp/examples/debuggingtut$ cat -n correcttypicalrr.eyp
 1 %token ID NUM
 2
 3 %%
 4 s:
 5     /* empty */
 6     | s ID
 7     | s NUM
 8 ;
 9
10 %%

```

See also these files in the `examples/debuggingtut/` directory:

- `typicalrr2.eyp` is equivalent to `typicalrr.eyp` but has `%name` directives, to have a nicer tree
- `typicalrr_fixed.eyp` eliminates the ambiguity using a combination of priorities and elimination of the redundant empty productions. Explicit precedence via `%prec` directives are given to produce right recursive lists
- `typicalrr_fixed_rightrecursive.eyp` is almost equal to `typicalrr_fixed.eyp` but eliminates the of `%prec` directives by making the list production right recursive

Giving an Explicit Priority to the End-of-Input Token

We can also try to disambiguate the former example using priorities. For that we need to give an explicit priority to the end-of-input token. To refer to the end-of-input token in the header section, use the empty string `''`. In the file `examples/debuggingtut/typicalrrwithprec.eyp` there is a priority based solution:

```

~/LEyapp/examples/debuggingtut$ eyapp -c typicalrrwithprec.eyp
%right LNUM
%right NUM
%right ID
%right '' # The string '' refers to the 'End of Input' token
%tree bypass

```

```

%%
s:
    %name EMPTY
    /* empty */%prec ''
  | %name LIST
    s ws
  | %name LIST
    s ns
;
ns:
    %name EMPTYNUM
    /* empty */%prec NUM
  | %name NUMS
    NUM ns
;
ws:
    %name EMPTYID
    /* empty */%prec LNUM
  | %name IDS
    ID ws
;
%%

```

Observe the use of %right '' in the header section: it gives a priority to the end-of-input token.

```

~/LEyapp/examples/debuggingtut$ ./typicalrrwithprec.pm -t
Try "4 5 a b 2 3" (press <CR><CTRL-D> to finish): 4 5 a b 2 3
^D
LIST(
  LIST(
    LIST(
      EMPTY,
      NUMS(
        TERMINAL[4],
        NUMS(
          TERMINAL[5],
          EMPTYNUM
        )
      )
    ),
    IDS(
      TERMINAL[a],
      IDS(
        TERMINAL[b],
        EMPTYID
      )
    )
  ),
  NUMS(
    TERMINAL[2],
    NUMS(
      TERMINAL[3],
      EMPTYNUM
    )
  )
)
)

```

Reduce-Reduce conflict: Enumerated versus Range declarations in Extended Pascal

The grammar in file examples/debuggingtut/pascalenumeratedvsrange.eyp:

```
~/LEyapp/examples/debuggingtut$ eyapp -c pascalenumeratedvsrange.eyp
%token TYPE DOTDOT ID
%left '+' '-'
%left '*' '/'

%%

type_decl:
    TYPE ID '=' type ';'
;
type:
    '(' id_list ')'
    | expr DOTDOT expr
;
id_list:
    ID
    | id_list ',' ID
;
expr:
    '(' expr ')'
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | ID
;

%%
```

introduces a problem that arises in the declaration of enumerated and subrange types in Pascal:

```
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 and many other Pascal implementations allow arbitrary expressions there. This gives rise to the following situation, containing a superfluous pair of parentheses:

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

Compare this to the following declaration of an enumerated type with only one value:

```
type enum = (a);
```

These two declarations look identical until the .. token. With normal LALR(1) one-token look-ahead 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 consequence is a reduce-reduce conflict, which is summarized in this state of the LALR automata:

State 10:

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

')' [reduce using rule 11 (expr)]
')' reduce using rule 4 (id_list)
',' reduce using rule 4 (id_list)
$default reduce using rule 11 (expr)
```

The grammar in file `pascalenumeratedvsrangesolvedvialex.eyp` solves this particular problem by looking ahead in the lexical analyzer: if the parenthesis is followed by a sequence of comma separated identifiers finished by the closing parenthesis and a semicolon we can conclude that is a enumerated type declaration. For more details, have a look at the file. Another solution using the postponed conflict resolution strategy can be found in file `pascalenumeratedvsrangesolvedviadyn.eyp`.

Reduce-Reduce Conflicts with Unambiguous Grammars

Though not so common, it may occur that a reduce-reduce conflict is not due to ambiguity but to the limitations of the LALR(1) algorithm. The following example illustrates the point:

```
pl@europa:~/LEyapp/examples/debuggingtut$ cat -n rrconflictnamefirst.eypp
 1 %token VAR ',' ':'
 2
 3 %{
 4 use base q{Tail};
 5 %}
 6
 7 %%
 8 def:   param_spec return_spec ','
 9       ;
10 param_spec:
11         type
12         | name_list ':' type
13         ;
14 return_spec:
15         type
16         | name ':' type
17         ;
18 name:   VAR
19         ;
20 type:  VAR
21         ;
22 name_list:
23         name
24         | name ',' name_list
25         ;
26 %%
27
28 __PACKAGE__->main unless caller();
```

This non ambiguous grammar generates a language of sequences like

a, b : e f : e,

The conflict is due to the final comma in:

```
def:   param_spec return_spec ','
```

If you suppress such comma there is no conflict (try it). When compiling with eyapp we get the warning:

```
pl@europa:~/LEyapp/examples/debuggingtut$ eyapp rrconflictnamefirst.eypp
1 reduce/reduce conflict
```

Editing the .output file we can see the conflict is in state 2:

```
46 State 2:
47
48     name -> VAR .   (Rule 6)
49     type -> VAR .   (Rule 7)
50
51     ','      [reduce using rule 7 (type)]
52     VAR      reduce using rule 7 (type)
53     $default      reduce using rule 6 (name)
```

If we look at the grammar we can see that a reduction by

type -> VAR .

may occur with a comma as incoming token but only after the reduction by `param_spec` has taken place. The problem is that the automaton forgets about it. Look the automaton transitions in the `.outputfile`. By making explicit the difference between the first and second `type` we solve the conflict:

```
pl@europa:~/LEyapp/examples/debuggingtut$ cat -n rrconflictnamefirst_fix1.eypp
 1 %token VAR ',' ':'
 2
 3 %{
 4 use base q{Tail};
 5 %}
 6
 7 %%
 8 def:    param_spec return_spec ','
 9        ;
10 param_spec:
11         type
12         | name_list ':' type
13         ;
14 return_spec:
15         typeafter
16         | name ':' typeafter
17         ;
18 name:    VAR
19         ;
20 type:    VAR
21         ;
22 typeafter:    VAR
23         ;
24 name_list:
25         name
26         | name ',' name_list
27         ;
28 %%
29
30 __PACKAGE__->main unless caller();
```

A reduce-reduce conflict is solved in favor of the first production found in the text. If we execute the grammar with the conflict `./rrconflictnamefirst.pm`, we get the correct behavior:

```
pl@europa:~/LEyapp/examples/debuggingtut$ eyapp -b '' rrconflictnamefirst.eypp
1 reduce/reduce conflict
pl@europa:~/LEyapp/examples/debuggingtut$ ./rrconflictnamefirst.pm
Expressions. Press CTRL-D (Unix) or CTRL-Z (Windows) to finish:
a,b:c d:e,
<CTRL-D>
$
```

The program accepts the correct language - in spite of the conflict - due to the fact that the production

```
name:    VAR
```

is listed first.

The parser rejects the correct phrases if we swap the order of the productions writing the `type: VAR` production first,

```
pl@europa:~/LEyapp/examples/debuggingtut$ ./reducereducerconflict.pm
Expressions. Press CTRL-D (Unix) or CTRL-Z (Windows) to finish:
a,b:c d:e,
<CTRL-D>
```

```
Syntax error near input: ',' (lin num 1).
Incoming text:
```

```
===
```

```
b:c d
```

```
===
```

Expected one of these terminals: VAR

Files `reducereducerconflict_fix1.eypp` and `reducereducerconflict_fix2.eypp` offer other solutions to the problem.

9 TOKENS DEPENDING ON THE SYNTACTIC CONTEXT

Usually there is a one-to-one relation between a token and a regexp. Problems arise, however when a token's type depends upon contextual information. An example of this problem comes from PL/I, where statements like this are legal:

```
if then=if then if=then
```

In PL/I this problem arises because keywords like `if` are not reserved and can be used in other contexts. This simplified grammar illustrates the problem:

```
examples/debuggingtut$ eyapp -c PL_I_conflict.eypp
# This grammar deals with the famous ambiguous PL/I phrase:
#           if then=if then if=then
# The (partial) solution uses YYExpect in the lexical analyzer to predict the token
# that fulfills the parser expectatives.
# Compile it with:
# eyapp -b '' PL_I_conflict.eypp
# Run it with;
# ./PL_I_conflict.pm -debug
%strict
%token ID
%tree bypass

%%

stmt:
    ifstmt
    | assignstmt
;
# Exercise: change this production
# for 'if' expr 'then' stmt
# and check with input 'if then=if then if=then'. The problem arises again
ifstmt:
    %name IF
    'if' expr 'then' expr
;
assignstmt:
    id '=' expr
;
expr:
    %name EQ
    id '=' id
    | id
;
id:
    %name ID
    ID
;

%%
```

If the token ambiguity depends only in the syntactic context, the problem can be alleviated using the `YYExpect` method. In case of doubt, the lexical analyzer calls the `YYExpect` method to know which of the several feasible tokens is expected by the parser:

```

examples/debuggingtut$ sed -ne '/sub lex/,/^}/p' PL_I_conflict.eypp
sub lexer {
  my $parser = shift;

  for ($parser->{input}) {    # contextualize
    m{\G\s*(\#.*?)?}gc;

    m{\G([a-zA-Z_]\w*)}gc and do {
      my $id = $1;

      return ('if', 'if') if ($id eq 'if') && is_in('if', $parser->YYExpect);
      return ('then', 'then') if ($id eq 'then') && is_in('then', $parser->YYExpect);

      return ('ID', $id);
    };

    m{\G(.)}gc          and return ($1, $1);

    return('', undef);
  }
}

```

Here follows an example of execution:

```

examples/debuggingtut$ eyapp -b '' PL_I_conflict.eypp
examples/debuggingtut$ ./PL_I_conflict.pm
Expressions. Press CTRL-D (Unix) or CTRL-Z (Windows) to finish:
if then=if then if=then
IF(EQ(ID,ID),EQ(ID,ID))

```

10 LEXICAL TIE-INS

A *lexical tie-in* is a flag which is set to alter the behavior of the lexical analyzer. It is a way to handle context-dependency.

The Parsing of C

The C language has a context dependency: the way an identifier is used depends on what its current meaning is. For example, consider this:

```
T(x);
```

This looks like a function call statement, but if T is a typedef name, then this is actually a declaration of x. How can a parser for C decide how to parse this input?

Here is another example:

```

{
  T * x;
  ...
}

```

What is this, a declaration of x as a pointer to T, or a void multiplication of the variables T and x?

The usual method to solve this problem is to have two different token types, ID and TYPENAME. When the lexical analyzer finds an identifier, it looks up in the symbol table the current declaration of the identifier in order to decide which token type to return: TYPENAME if the identifier is declared as a typedef, ID otherwise. See the ANSI C parser example in the directory `examples/languages/C/ansic.eypp`

A Simple Example

In the "Calc"-like example in `examples/debuggintut/SemanticInfoInTokens.eyy` we have a language with a special construct `hex (hex-expr)`. After the keyword `hex` comes an `expression` in parentheses in which all integers are hexadecimal. In particular, strings in `/[A-F0-9]+/` like `A1B` must be treated as a hex integer unless they were previously declared as variables. Let us see an example of execution:

```
$ eyapp -b '' SemanticInfoInTokens.eyy
$ cat inputforsemanticinfo2.txt
int A2
A2 = HEX(A23);
A2 = HEX(A2)

$ ./SemanticInfoInTokens.pm -f inputforsemanticinfo2.txt -t
EXPS(ASSIGN(TERMINAL[A2],NUM[2595]),ASSIGN(TERMINAL[A2],ID[A2]))
```

The first hex expression `HEX(A23)` is interpreted as the number 2595 while the second `HEX(A2)` refers to previously declared variable `A2`.

An alternative solution to this problem that does not make use of lexical tie-ins - but still uses an attribute `HEXFLAG` for communication between different semantic actions - can be found in the file `examples/debuggintut/Tieins.eyy`.

```
pl@nereida:~/LEyapp/examples/debuggintut$ sed -ne '5,91p' SemanticInfoInTokens.eyy | cat -n
 1 %strict
 2
 3 %token ID INT INTEGER
 4 %syntactic token HEX
 5
 6 %right '='
 7 %left '+'
 8
 9 %{
10 use base q{DebugTail};
11 my %st;
12 %}
13
14 %tree bypass alias
15
16 %%
17 stmt:
18     decl <* ','> expr <%name EXPS + ','>
19     {
20         # make the symbol table an attribute
21         # of the root node
22         $_[2]->{st} = { %st };
23         $_[2];
24     }
25 ;
26
27 decl:
28     INT ID <+ ','>
29     {
30         # insert identifiers in the symbol table
31         $st{$_->{attr}} = 1 for $_[2]->children();
32     }
33 ;
34
35 expr:
36     %name ID
37     ID
38     | %name NUM
39     INTEGER
40     | %name HEX
```

```

41     HEX '( ' { $_[0]->{HEXFLAG} = 1; } $expr ' )'
42     {
43         $_[0]->{HEXFLAG} = 0;
44         $expr;
45     }
46 | %name ASSIGN
47   id '=' expr
48 | %name PLUS
49   expr '+' expr
50 ;
51
52 id : ID
53 ;
54
55 %%
56
57 # Context-dependant lexer
58 __PACKAGE__->lexer( sub {
59     my $parser = shift;
60     my $hexflag = $parser->{HEXFLAG};
61
62     for (${$parser->input}) { # contextualize
63         m{\G\s*(\#.*?)?}gc;
64
65         m{\G(HEX\b|INT\b)}igc and return (uc($1), $1);
66
67         m{\G\d+}gc and return ('INTEGER', $hexflag? hex($1) : $1);
68
69
70         m{\G([a-zA-Z_]\w*)}gc and do {
71             my $match = $1;
72             $hexflag and !exists($st{$match}) and $match =~ m{^[A-F0-9]+$}gc and return ('INTEGER', $1);
73             return ('ID', $1);
74         };
75
76         m{\G(.)}gc          and return ($1, $1);
77
78         return('','undef');
79     }
80 }
81 );
82
83 *TERMINAL::info = *NUM::info = *ID::info = sub {
84     $_[0]->{attr}
85 };
86
87 __PACKAGE__->main() unless caller();

```

Here the lexical analyzer looks at the value of the attribute `HEXFLAG`; when it is nonzero, all integers are parsed in hexadecimal, and tokens starting with letters are parsed as integers if possible.

References about Lexical tie-ins

For more about lexical tie-ins see also

- http://www.gnu.org/software/bison/manual/html_mono/bison.html#Lexical-Tie_002dins
- http://en.wikipedia.org/wiki/The_lexer_hack
- <http://eli.thegreenplace.net/2007/11/24/the-context-sensitivity-of-cs-grammar/>

11 SOLVING CONFLICTS WITH THE *POSTPONED CONFLICT STRATEGY*

Yacc-like parser generators provide ways to solve shift-reduce mechanisms based on token precedence. No mechanisms are provided for the resolution of reduce-reduce conflicts. The solution for such kind of conflicts is to modify the grammar. The strategy We present here provides a way to broach conflicts that can't be solved using static precedences.

The *Postponed Conflict Resolution Strategy*

The *postponed conflict strategy* presented here can be used whenever there is a shift-reduce or reduce-reduce conflict that can not be solved using static precedences.

Postponed Conflict Resolution: Reduce-Reduce Conflicts

Let us assume we have a reduce-reduce conflict between two productions

```
A -> alpha .
B -> beta .
```

for some token @. Let also assume that production

```
A -> alpha
```

has name `ruleA` and production

```
B -> beta
```

has name `ruleB`.

The postponed conflict resolution strategy consists in modifying the conflictive grammar by marking the points where the conflict occurs with the new `%PREC` directive. In this case at the end of the involved productions:

```
A -> alpha %PREC IsAorB
B -> beta $PREC IsAorB
```

The `IsAorB` identifier is called the *conflict name*.

Inside the head section, the programmer associates with the conflict name a code whose mission is to solve the conflict by dynamically changing the parsing table like this:

```
%conflict IsAorB {
    my $self = shift;

    if (looks_like_A($self)) {
        $self->YYSetReduce('@', 'ruleA' );
    }
    else {
        $self->YYSetReduce('@', 'ruleB' );
    }
}
```

The code associated with the *conflict name* receives the name of *conflict handler*. The code of `looks_like_A` stands for some form of nested parsing which will decide which production applies.

Solving the Enumerated versus Range declarations conflict using the *Posponed Conflict Resolution Strategy*

In file `pascalenumeratedvsrangesolvedviadyn.eyy` we apply the postponed conflict resolution strategy to the reduce reduce conflict that arises in Extended Pascal between the declaration of ranges and the declaration of enumerated types (see section Reduce-Reduce conflict: Enumerated versus Range declarations in Extended Pascal). Here is the solution:

```

~/LEyapp/examples/debuggingtut$ cat -n pascalenumeratedvsrangesolvedviadyn.eypp
 1  %{
 2  =head1 SYNOPSIS
 3
 4  See
 5
 6  =over 2
 7
 8  =item * File pascalenumeratedvsrange.eypp in examples/debuggingtut/
 9
10  =item * The Bison manual L<http://www.gnu.org/software/bison/manual/html\_mono/bison.html>
11
12  =back
13
14  Compile it with:
15
16          eyapp -b '' pascalenumeratedvsrangesolvedviadyn.eypp
17
18  run it with this options:
19
20          ./pascalenumeratedvsrangesolvedviadyn.pm -t
21
22  Try these inputs:
23
24          type r = (x) .. y ;
25          type r = (x+2)*3 .. y/2 ;
26          type e = (x, y, z);
27          type e = (x);
28
29  =cut
30
31  use base q{DebugTail};
32
33  my $ID = qr{[A-Za-z][A-Za-z0-9_]*};
34          # Identifiers separated by commas
35  my $IDLIST = qr{ \s*(?:\s*,\s* $ID)* \s* }x;
36          # list followed by a closing par and a semicolon
37  my $RESTOFLIST = qr{ $IDLIST \) \s* ; }x;
38  %}
39
40  %namingscheme {
41      #Receives a Parse::Eyapp object describing the grammar
42      my $self = shift;
43
44      $self->tokennames(
45          '(' => 'LP',
46          '..' => 'DOTDOT',
47          ',' => 'COMMA',
48          ')' => 'RP',
49          '+' => 'PLUS',
50          '-' => 'MINUS',
51          '*' => 'TIMES',
52          '/' => 'DIV',
53      );
54
55      # returns the handler that will give names
56      # to the right hand sides
57      \&give_rhs_name;
58  }
59

```

```

60 %strict
61
62 %token ID NUM DOTDOT TYPE
63 %left '-' '+'
64 %left '*' '/'
65
66 %tree
67
68 %%
69
70 type_decl : TYPE ID '=' type ';'
71 ;
72
73 type :
74     %name ENUM
75     '(' id_list ')'
76     | %name RANGE
77     expr DOTDOT expr
78 ;
79
80 id_list :
81     %name EnumID
82     ID rangeORenum
83     | id_list ',' ID
84 ;
85
86 expr : '(' expr ')'
87     | expr '+' expr
88     | expr '-' expr
89     | expr '*' expr
90     | expr '/' expr
91     | %name RangeID
92     ID rangeORenum
93     | NUM
94 ;
95
96 rangeORenum: /* empty: postponed conflict resolution */
97     {
98         my $parser = shift;
99         if (${$parser->input()} =~ m{\G(?:= $RESTOFLIST)}gcx) {
100             $parser->YYSetReduce(')', 'EnumID' );
101         }
102         else {
103             $parser->YYSetReduce(')', 'RangeID' );
104         }
105     }
106 ;
107
108 %%
109
110 __PACKAGE__->lexer(
111     sub {
112         my $parser = shift;
113
114         for (${$parser->input()}) { # contextualize
115             m{\G(\s*)}gc;
116             $parser->tokenline($1 =~ tr{\n}{});
117
118             m{\Gtype\b}gic                and return ('TYPE', 'TYPE');
119

```

```

120         m{\G($ID)}gc                and return ('ID', $1);
121
122         m{\G([0-9+]})gc             and return ('NUM', $1);
123
124         m{\G\.\.}gc                 and return ('DOTDOT', '..');
125
126         m{\G(.)}gc                  and return ($1, $1);
127
128         return('',undef);
129     }
130 }
131 );
132
133 unless (caller()) {
134     $Parse::Eyapp::Node::INDENT = 1;
135     my $prompt = << 'EOP';
136     Try this input:
137         type
138         r
139         =
140         (x)
141         ..
142         y
143         ;
144
145     Here other inputs you can try:
146
147         type r = (x+2)*3 .. y/2 ;
148         type e = (x, y, z);
149         type e = (x);
150
151     Press CTRL-D (CTRL-W in windows) to produce the end-of-file
152     EOP
153     __PACKAGE__->main($prompt);
154 }

```

This example also illustrates how to modify the default production naming schema. Follows the result of several executions:

```
~/LEyapp/examples/debuggingtut$ ./pascalenumeratedvsrangesolvedviadyn.pm -t
```

Try this input:

```

type
r
=
(x)
..
y
;

```

Here other inputs you can try:

```

type r = (x+2)*3 .. y/2 ;
type e = (x, y, z);
type e = (x);

```

Press CTRL-D (CTRL-W in windows) to produce the end-of-file

```
type r = (x+2)*3 .. y/2 ;
```

```
^D
```

```

type_decl_is_TYPE_ID_type(
  TERMINAL[TYPE],
  TERMINAL[r],
  RANGE(

```


As you remember the conflict was:

```
~/LEyapp/examples/debuggingtut$ sed -ne '/^St.*13:/,/^St.*14/p' DynamicallyChangingTheParser.output
State 13:
```

```
ds -> D conflict . ';' ds (Rule 6)
ds -> D conflict . (Rule 7)

';' shift, and go to state 16

';' [reduce using rule 7 (ds)]
```

State 14:

The conflict handler below sets the LR action to reduce by the production with name D1

```
ds -> D
```

in the presence of token ';' if indeed is the last 'D', that is, if:

```
`${$self->input()} =~ m{^\s*;\s*$}
```

Otherwise we set the shift action via a call to the YYSetShift method.

```
~/LEyapp/examples/debuggingtut$ sed -ne '30,$p' DynamicallyChangingTheParser.eyy | cat -n
 1 %token D S
 2
 3 %tree bypass
 4
 5 # Expect just 1 shift-reduce conflict
 6 %expect 1
 7
 8 %%
 9 p: %name PROG
10     block +
11 ;
12
13 block:
14     %name BLOCK_DS
15     '{' ds ';' ss '}'
16     | %name BLOCK_S
17     '{' ss '}'
18 ;
19
20 ds:
21     %name D2
22     D conflict ';' ds
23     | %name D1
24     D conflict
25 ;
26
27 ss:
28     %name S2
29     S ';' ss
30     | %name S1
31     S
32 ;
33
34 conflict:
35     /* empty. This action solves the conflict using dynamic precedence */
36     {
37         my $self = shift;
38
```

```

39         if (${$self->input()} =~ m{^\s*;\s*S}) {
40             $self->YYSetReduce(';', 'D1' )
41         }
42         else {
43             $self->YYSetShift(';')
44         }
45
46         undef; # skip this node in the AST
47     }
48 ;
49
50 %%
51
52 my $prompt = 'Provide a statement like "{D; S} {D; D; S}" and press <CR><CTRL-D>: ';
53 __PACKAGE__->main($prompt) unless caller;

```

12 TREE EQUALITY

The more the time invested writing tests the less the time spent debugging. This section deals with the *Parse::Eyapp::Node* method `equal` which can be used to test that the trees have the shape we expect.

`$node->equal`

A call `$tree1->equal($tree2)` compare the two trees `$tree1` and `$tree2`. Two trees are considered equal if their root nodes belong to the same class, they have the same number of children and the children are (recursively) equal.

In Addition to the two trees the programmer can specify pairs `attribute_key => equality_handler`:

```
$tree1->equal($tree2, attr1 => \&handler1, attr2 => \&handler2, ...)
```

In such case the definition of equality is more restrictive: Two trees are considered equal if

- Their root nodes belong to the same class,
- They have the same number of children
- For each of the specified attributes occur that for both nodes the existence and definition of the key is the same
- Assuming the key exists and is defined for both nodes, the equality handlers return true for each of its attributes and
- The respective children are (recursively) equal.

An attribute handler receives as arguments the values of the attributes of the two nodes being compared and must return true if, and only if, these two attributes are considered equal. Follows an example:

```

examples/Node$ cat -n equal.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use Parse::Eyapp::Node;
 4
 5  my $string1 = shift || 'ASSIGN(VAR(TERMINAL))';
 6  my $string2 = shift || 'ASSIGN(VAR(TERMINAL))';
 7  my $t1 = Parse::Eyapp::Node->new($string1, sub { my $i = 0; $_->{n} = $i++ for @_ });
 8  my $t2 = Parse::Eyapp::Node->new($string2);
 9
10  # Without attributes
11  if ($t1->equal($t2)) {
12      print "\nNot considering attributes: Equal\n";
13  }
14  else {
15      print "\nNot considering attributes: Not Equal\n";

```

```

16 }
17
18 # Equality with attributes
19 if ($t1->equal($t2, n => sub { return $_[0] == $_[1] }))) {
20     print "\nConsidering attributes: Equal\n";
21 }
22 else {
23     print "\nConsidering attributes: Not Equal\n";
24 }

```

When the former program is run without arguments produces the following output:

```
examples/Node$ equal.pl
```

```
Not considering attributes: Equal
```

```
Considering attributes: Not Equal
```

Using equal During Testing

During the development of your compiler you add new stages to the existing ones. The consequence is that the AST is decorated with new attributes. Unfortunately, this implies that tests you wrote using `is_deeply` and comparisons against formerly correct abstract syntax trees are no longer valid. This is due to the fact that `is_deeply` requires both tree structures to be equivalent in every detail and that our new code produces a tree with new attributes.

Instead of `is_deeply` use the `equal` method to check for partial equivalence between abstract syntax trees. You can follow these steps:

- Dump the tree for the source inserting `Data::Dumper` statements
- Carefully check that the tree is really correct
- Decide which attributes will be used for comparison
- Write the code for the expected value editing the output produced by `Data::Dumper`
- Write the handlers for the attributes you decided. Write the comparison using `equal`.

Tests using this methodology will not fail even if later code decorating the AST with new attributes is introduced.

See an example that checks an abstract syntax tree produced by the simple compiler (see `examples/typechecking/Simple` for a really simple source:

```
Simple-Types/script$ cat prueba27.c
int f() {
}

```

The first thing is to obtain a description of the tree, that can be done executing the compiler under the control of the Perl debugger, stopping just after the tree has been built and dumping the tree with `Data::Dumper`:

```

pl@nereida:~/Lbook/code/Simple-Types/script$ perl -wd usetypes.pl prueba27.c
main::(usetypes.pl:5): my $filename = shift || die "Usage:\n$0 file.c\n";
DB<1> c 12
main::(usetypes.pl:12): Simple::Types::show_trees($t, $debug);
DB<2> use Data::Dumper
DB<3> $Data::Dumper::Purity = 1
DB<4> p Dumper($t)
$VAR1 = bless( {
    .....,
    }, 'PROGRAM' );
.....

```

Once we have the shape of a correct tree we can write our tests:

examples/Node\$ cat -n testequal.pl

```
1  #!/usr/bin/perl -w
2  use strict;
3  use Parse::Eyapp::Node;
4  use Data::Dumper;
5  use Data::Compare;
6
7  my $debugging = 0;
8
9  my $handler = sub {
10     print Dumper($_[0], $_[1]) if $debugging;
11     Compare($_[0], $_[1])
12 };
13
14 my $t1 = bless( {
15     'types' => {
16         'CHAR' => bless( { 'children' => [] }, 'CHAR' ),
17         'VOID' => bless( { 'children' => [] }, 'VOID' ),
18         'INT' => bless( { 'children' => [] }, 'INT' ),
19         'F(X_0(),INT)' => bless( {
20             'children' => [
21                 bless( { 'children' => [] }, 'X_0' ),
22                 bless( { 'children' => [] }, 'INT' ) ]
23             }, 'F' )
24     },
25     'symboltable' => { 'f' => { 'type' => 'F(X_0(),INT)', 'line' => 1 } },
26     'lines' => 2,
27     'children' => [
28         bless( {
29             'symboltable' => {},
30             'fatherblock' => {},
31             'children' => [],
32             'depth' => 1,
33             'parameters' => [],
34             'function_name' => [ 'f', 1 ],
35             'symboltableLabel' => {},
36             'line' => 1
37         }, 'FUNCTION' )
38     ],
39     'depth' => 0,
40     'line' => 1
41 }, 'PROGRAM' );
42 $t1->{'children'}[0]['fatherblock'] = $t1;
43
44 # Tree similar to $t1 but without some attributes (line, depth, etc.)
45 my $t2 = bless( {
46     'types' => {
47         'CHAR' => bless( { 'children' => [] }, 'CHAR' ),
48         'VOID' => bless( { 'children' => [] }, 'VOID' ),
49         'INT' => bless( { 'children' => [] }, 'INT' ),
50         'F(X_0(),INT)' => bless( {
51             'children' => [
52                 bless( { 'children' => [] }, 'X_0' ),
53                 bless( { 'children' => [] }, 'INT' ) ]
54             }, 'F' )
55     },
56     'symboltable' => { 'f' => { 'type' => 'F(X_0(),INT)', 'line' => 1 } },
57     'children' => [
58         bless( {
59             'symboltable' => {},
```

```

60         'fatherblock' => {},
61         'children' => [],
62         'parameters' => [],
63         'function_name' => [ 'f', 1 ],
64     }, 'FUNCTION' )
65     ],
66     }, 'PROGRAM' );
67 $t2->{'children'}[0]{'fatherblock'} = $t2;
68
69 # Tree similar to $t1 but without some attributes (line, depth, etc.)
70 # and without the symboltable and types attributes used in the comparison
71 my $t3 = bless( {
72     'types' => {
73         'CHAR' => bless( { 'children' => [] }, 'CHAR' ),
74         'VOID' => bless( { 'children' => [] }, 'VOID' ),
75         'INT' => bless( { 'children' => [] }, 'INT' ),
76         'F(X_0(),INT)' => bless( {
77             'children' => [
78                 bless( { 'children' => [] }, 'X_0' ),
79                 bless( { 'children' => [] }, 'INT' ) ]
80             }, 'F' )
81     },
82     'children' => [
83         bless( {
84             'symboltable' => {},
85             'fatherblock' => {},
86             'children' => [],
87             'parameters' => [],
88             'function_name' => [ 'f', 1 ],
89         }, 'FUNCTION' )
90     ],
91     }, 'PROGRAM' );
92
93 $t3->{'children'}[0]{'fatherblock'} = $t2;
94
95 # Without attributes
96 if (Parse::Eyapp::Node::equal($t1, $t2)) {
97     print "\nNot considering attributes: Equal\n";
98 }
99 else {
100     print "\nNot considering attributes: Not Equal\n";
101 }
102
103 # Equality with attributes
104 if (Parse::Eyapp::Node::equal(
105     $t1, $t2,
106     symboltable => $handler,
107     types => $handler,
108 )
109 ) {
110     print "\nConsidering attributes: Equal\n";
111 }
112 else {
113     print "\nConsidering attributes: Not Equal\n";
114 }
115
116 # Equality with attributes
117 if (Parse::Eyapp::Node::equal(
118     $t1, $t3,
119     symboltable => $handler,

```

```

120     types => $handler,
121   )
122 ) {
123     print "\nConsidering attributes: Equal\n";
124 }
125 else {
126     print "\nConsidering attributes: Not Equal\n";
127 }

```

The code defining tree `$t1` was obtained from an output using `Data::Dumper`. The code for trees `$t2` and `$t3` was written using cut-and-paste from `$t1`. They have the same shape than `$t1` but differ in their attributes. Tree `$t2` shares with `$t1` the attributes `symboltable` and `types` used in the comparison and so `equal` returns `true` when compared. Since `$t3` differs from `$t1` in the attributes `symboltable` and `types` the call to `equal` returns `false`.

13 FORMATTING *Parse::Eyapp* PROGRAMS

I use these rules for indenting *Parse::Eyapp* programs:

- Use uppercase identifiers for tokens, lowercase identifiers for syntactic variables
- The syntactic variable that defines the rule must be at in a single line at the leftmost position:

```

synvar:
    'a' othervar 'c'
  | 'b' anothervar SOMETOKEN
;

```

The separation bar `|` goes indented relative to the left side of the rule. Each production starts two spaces from the bar. The first right hand side is aligned with the rest.

- The semicolon `;` must also be in its own line at column 0
- If there is an empty production it must be the first one and must be commented

```

syntacvar:
    /* empty */
  | 'a' othervar 'c'
  | 'b' anothervar
;

```

- Only very short semantic actions can go in the same line than the production. Semantic actions requiring more than one line must go in its own indented block like in:

```

exp:
    $NUM           { $NUM->[0] }
  | $VAR
    {
      my $id = $VAR->[0];
      my $val = $s{$id};
      $_[0]->semantic_error("Accessing undefined variable $id at line $VAR->[1].\n")
      unless defined($val);
      return $val;
    }
  | $VAR '=' $exp  { $s{$VAR->[0]} = $exp }
  | exp.x '+' exp.y { $x + $y }
  | exp.x '-' exp.y { $x - $y }
  | exp.x '*' exp.y { $x * $y }
  | exp.x '/' .barr exp.y
    {
      return($x/$y) if $y;
      $_[0]->semantic_error("Illegal division by zero at line $barr->[1].\n");
    }

```

```

        undef
    }
    | '-' $exp %prec NEG { -$exp }
    | exp.x '^' exp.y    { $x ** $y }
    | '(' $exp ')'      { $exp }
;

```

14 SEE ALSO

- The project home is at <http://code.google.com/p/parse-eyapp/>. Use a subversion client to anonymously check out the latest project source code:

```
svn checkout http://parse-eyapp.googlecode.com/svn/trunk/ parse-eyapp-read-only
```

- The tutorial *Parsing Strings and Trees with Parse::Eyapp* (An Introduction to Compiler Construction in seven pages) in <http://nereida.deioc.ull.es/~pl/eyapsimple/>
- *Parse::Eyapp*, *Parse::Eyapp::eyapplanguageref*, *Parse::Eyapp::debuggingtut*, *Parse::Eyapp::defaultactionsintro*, *Parse::Eyapp::translationschemestut*, *Parse::Eyapp::Driver*, *Parse::Eyapp::Node*, *Parse::Eyapp::YATW*, *Parse::Eyapp::Treeregexp*, *Parse::Eyapp::Scope*, *Parse::Eyapp::Base*, *Parse::Eyapp::datagenerationtut*
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/languageintro.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/debuggingtut.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/eyapplanguageref.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/Treeregexp.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/Node.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/YATW.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/Eyapp.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/Base.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/translationschemestut.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/treematchingtut.pdf>
- perldoc *eyapp*,
- perldoc *treereg*,
- perldoc *vgg*,
- The Syntax Highlight file for vim at http://www.vim.org/scripts/script.php?script_id=2453 and <http://nereida.deioc.ull.es>
- *Analisis Lexico y Sintactico*, (Notes for a course in compiler construction) by Casiano Rodriguez-Leon. Available at <http://nereida.deioc.ull.es/~pl/perlexamples/> Is the more complete and reliable source for Parse::Eyapp. However is in Spanish.
- *Parse::Yapp*,
- Man pages of yacc(1) and bison(1), <http://www.delorie.com/gnu/docs/bison/bison.html>
- *Language::AttributeGrammar*
- *Parse::RecDescent*.
- *HOP::Parser*
- *HOP::Lexer*
- ocaml yacc tutorial at <http://plus.kaist.ac.kr/~shoh/ocaml/ocamllex-ocaml yacc/ocaml yacc-tutorial/ocaml yacc-tutorial.html>

15 REFERENCES

- The classic Dragon's book *Compilers: Principles, Techniques, and Tools* by Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman (Addison-Wesley 1986)
- *CS2121: The Implementation and Power of Programming Languages* (See <http://www.cs.man.ac.uk/~pjj>, <http://www.cs.man.ac.uk/~pjj/complang/g2lr.html> and <http://www.cs.man.ac.uk/~pjj/cs2121/ho/ho.html>) by Pete Jinks

16 CONTRIBUTORS

- Hal Finkel <http://www.halssoftware.com/>
- G. Williams <http://kasei.us/>
- Thomas L. Shinnick <http://search.cpan.org/~tshinnic/>
- Frank Leray

17 AUTHOR

Casiano Rodriguez-Leon (casiano@ull.es)

18 ACKNOWLEDGMENTS

This work has been supported by CEE (FEDER) and the Spanish Ministry of *Educacion y Ciencia* through *Plan Nacional I+D+I* number TIN2005-08818-C04-04 (ULL::OPLINK project <http://www.oplink.ull.es/>). Support from Gobierno de Canarias was through GC02210601 (*Grupos Consolidados*). The University of La Laguna has also supported my work in many ways and for many years.

A large percentage of code is verbatim taken from *Parse::Yapp* 1.05. The author of *Parse::Yapp* is Francois Desarmenien.

I wish to thank Francois Desarmenien for his *Parse::Yapp* module, to my students at La Laguna and to the Perl Community. Thanks to the people who have contributed to improve the module (see CONTRIBUTORS in *Parse::Eyapp*). Thanks to Larry Wall for giving us Perl. Special thanks to Juana.

19 LICENCE AND COPYRIGHT

Copyright (c) 2006-2008 Casiano Rodriguez-Leon (casiano@ull.es). All rights reserved.

Parse::Yapp copyright is of Francois Desarmenien, all rights reserved. 1998-2001

These modules are free software; you can redistribute it and/or modify it under the same terms as Perl itself. See *perlartistic*.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Index

- \$node->equal, 34
- A Simple Example, 26
- ACKNOWLEDGMENTS, 40
- An eyapp Program with Errors, 3
- AUTHOR, 40

- CONFLICTS AND AMBIGUITIES, 2
- CONTRIBUTORS, 40

- Detecting Conflicts, 5

- ERRORS DURING TREE CONSTRUCTION, 10
- ERRORS INSIDE SEMANTIC ACTIONS, 16

- Focusing in the Grammar, 4
- FORMATTING Parse::Eyapp PROGRAMS, 38

- Giving an Explicit Priority to the End-of-Input Token, 19

- INTRODUCTION, 1

- LEXICAL TIE-INS, 25
- LICENCE AND COPYRIGHT, 40

- NAME, 1

- Postponed Conflict Resolution: Reduce-Reduce Conflicts, 28
- Postponed Conflict Resolution: Shift-Reduce Conflicts, 32

- Reduce-Reduce Conflict: Default rules, 17
- Reduce-Reduce conflict: Enumerated versus Range declarations in Extended Pascal, 20
- Reduce-Reduce Conflicts with Unambiguous Grammars, 22
- Reduce-Reduce conflicts: typical errors, 18
- REFERENCES, 40
- References about Lexical tie-ins, 27

- SEE ALSO, 39
- SOLVING CONFLICTS WITH THE POSTPONED CONFLICT STRATEGY, 28
- SOLVING REDUCE-REDUCE CONFLICTS, 17
- Solving Shift-Reduce Conflicts by Factorizing, 8
- Solving Shift-Reduce Conflicts By Looking Ahead, 9
- Solving the Enumerated versus Range declarations conflict using the Posponed Conflict Resolution Strategy, 28
- Studying the .output file, 5

- THE %strict DIRECTIVE, 1
- The Importance of the FOLLOW Set, 8
- THE LR PARSING ALGORITHM: UNDERSTANDING THE OUTPUT OF yydebug, 13
- The Parsing of C, 25
- The Postponed Conflict Resolution Strategy, 28

- TOKENS DEPENDING ON THE SYNTACTIC CONTEXT, 24
- TREE EQUALITY, 34

- Understanding Priorities, 2
- Using equal During Testing, 35