1 NAME

Parse::Eyapp::eyapplanguageref - The Eyapp language reference manual

2 THE EYAPP LANGUAGE

Eyapp Grammar

This section describes the syntax of the Eyapp language using its own notation. The grammar extends *yacc* and *yapp* grammars. Semicolons have been omitted to save space. Between C-like comments you can find an (informal) explanation of the language associated with each token.

```
%token ASSOC /* is %(left|right|nonassoc) */
%token BEGINCODE /* is %begin { Perl code ... } */
%token CODE /* is { Perl code ... } */
%token CONFLICT /* is %conflict */
%token DEFAULTACTION /* is %defaultaction */
%token EXPECT /* is %expect */
%token HEADCODE /* is %{ Perl code ... %} */
%token IDENT /* is [A-Za-z_][A-Za-z0-9_]* */
%token LABEL /* is :[A-Za-z0-9_]+ */
%token LITERAL /* is a string literal like 'hello' */
%token METATREE /* is %metatree */
%token NAME /* is %name */
%token NAMINGSCHEME /* is %namingscheme */
%token NOCOMPACT /* is %nocompact */
%token NUMBER /* is \d+ */
%token OPTION /* is (%name\s*([A-Za-z_]\w*)\s*)?\? */
%token PLUS /* is (%name\s*([A-Za-z_]\w*)\s*)?\+ */
%token PREC /* is %prec */
%token PREFIX /* is %prefix\s+([A-Za-z_][A-Za-z0-9_:]*::) */
%token SEMANTIC /* is %semantic\s+token */
%token STAR /* is (%name\s*([A-Za-z_]\w*)\s*)?\* */
%token START /* is %start */
%token STRICT /* is %strict */
%token SYNTACTIC /* is %syntactic\s+token */
%token TAILCODE /* is { Perl code ... } */
%token TOKEN /* is %token */
%token TREE /* is %tree */
%token TYPE /* is %type */
%token UNION /* is %union */
%start eyapp

%%

# Main rule
eyapp:
      head body tail
;
#Common rules:
symbol:
      LITERAL
    | ident #default action
;
ident:
      IDENT
;
# Head section:
head:
      headsec '%%'
;
```

```
headsec:
      #empty  #default action
    | decls #default action
;
decls:
      decls decl #default action
    | decl #default action
;
decl:
      '\n' #default action
    | SEMANTIC typedecl symlist '\n'
    | SYNTACTIC typedecl symlist '\n'
    | TOKEN typedecl toklist '\n'
    | ASSOC typedecl symlist '\n'
    | START ident '\n'
    | PREFIX '\n'
    | WHITES CODE '\n'
    | WHITES REGEXP '\n'
    | WHITES '=' CODE '\n'
    | WHITES '=' REGEXP '\n'
    | NAMINGSCHEME CODE '\n'
    | HEADCODE '\n'
    | UNION CODE '\n'  #ignore
    | DEFAULTACTION CODE '\n'
    | LEXER CODE '\n'
    | TREE '\n'
    | METATREE '\n'
    | STRICT '\n'
    | NOCOMPACT '\n'
    | TYPE typedecl identlist '\n'
    | CONFLICT ident CODE '\n'
    | EXPECT NUMBER '\n'
    | EXPECT NUMBER NUMBER '\n'
    | EXPECTRR NUMBER '\n'
    | error '\n'
;
typedecl:
      #empty
    | '<' IDENT '>'
;
symlist:
      symlist symbol
    | symbol
;
toklist:
      toklist tokendef
    | tokendef
;
tokendef:
      symbol '=' REGEXP
    | symbol '=' CODE
    | symbol
;
identlist:
      identlist ident
    | ident
;
# Rule section
body:
      rulesec '%%'
```

```
    | '%%'
;
rulesec:
      rulesec rules #default action
    | startrules #default action
;
startrules:
      IDENT ':'  rhss ';'
    | error ';'
;
rules:
      IDENT ':' rhss ';'
    | error ';'
;
rhss:
      rhss '|' rule
    | rule
;
rule:
      optname rhs prec epscode
    | optname rhs
;
rhs:
      #empty      #default action (will return undef)
    | rhselts #default action
;
rhselts:
      rhselts rhseltwithid
    | rhseltwithid
;
rhseltwithid:
      rhselt '.' IDENT
    | '$' rhselt
    | '$' error
    | rhselt
;
rhselt:
      symbol
    | code
    | DPREC ident
    | '(' optname rhs ')'
    | rhselt STAR
    | rhselt '<' STAR symbol '>'
    | rhselt OPTION
    | rhselt '<' PLUS symbol '>'
    | rhselt PLUS
;
optname:
      /* empty */
    | NAME IDENT
    | NAME IDENT LABEL
    | NAME LABEL
;
prec:
      PREC symbol
;
epscode:
    | code
;
code:
```

```
      CODE
    | BEGINCODE
;
# Tail section:
tail:
      /*empty*/
    | TAILCODE
;

%%
```

The semantic of `Eyapp` agrees with the semantic of `yacc` and `yapp` for all the common constructions.

Comments

Comments are either Perl style, from `#` up to the end of line, or C style, enclosed between `/*` and `*/`.

Syntactic Variables, Symbolic Tokens and String Literals

Two kind of symbols may appear inside a Parse::Eyapp program: *Non-terminal* symbols or *syntactic variables*, called also *left-hand-side* symbols and *Terminal* symbols, called also *Tokens*.

Tokens are the symbols the lexical analyzer function returns to the parser. There are two kinds of tokens: *symbolic tokens* and *string literals*.

Syntactic variables and *symbolic tokens* identifiers must conform to the regular expression `[A-Za-z][A-Za-z0-9_]*`.

When building the syntax tree (i.e. when running under the `%tree` directive) *symbolic tokens* will be considered *semantic tokens* (see section Syntactic and Semantic tokens). *Symbolic tokens* yield nodes in the Abstract Syntax Tree.

String literals are enclosed in single quotes and can contain almost anything. They will be received by the parser as double-quoted strings. Any special character as `'"'`, `'$'` and `'@'` is escaped. To have a single quote inside a literal, escape it with '\'.

When building the syntax tree (i.e. when running under the `%tree` directive) *string literals* will be considered *syntactic tokens* (see section Syntactic and Semantic tokens). *Syntactic tokens* do not produce nodes in the Abstract Syntax Tree.

The examples used along this document can be found in the directory `examples/eyapplanguageref` accompanying this distribution.

Parts of an `eyapp` Program

An Eyapp program has three parts called head, body and tail:

```
eyapp: head body tail ;
```

Each part is separated from the former by the symbol `%%`:

```
head: headsec '%%'
body: rulesec '%%'
```

3 THE HEAD SECTION

The head section contains a list of declarations

```
headsec:  decl *
```

There are different kinds of declarations.

This reference does not fully describes all the declarations that are shared with `yacc` and *yapp*.

Example of Head Section

In this and the next sections we will describe the basics of the Eyapp language using the file `examples/eyapplanguageref/Cal` that accompanies this distribution. This file implements a trivial calculator. Here is the header section:

```
pl@nereida:~/src/perl/eyapp/examples/eyapplanguageref$ sed -ne '1,/%%/p' Calc.eyp | cat -n
   1  # examples/eyapplanguageref/Calc.eyp
   2  %whites    =  /([ \t]*(?:#.*)?)/
   3  %token NUM =  /([0-9]+(?:\.[0-9]+)?)/
   4  %token VAR =  /([A-Za-z][A-Za-z0-9_]*)/
   5
   6  %right  '='
   7  %left   '-' '+'
   8  %left   '*' '/'
   9  %left   NEG
  10  %right  '^'
  11
  12  %{
  13  my %s; # symbol table
  14  %}
  15
  16  %%
```

Eyapp produces a lexical generator from the descriptions given by the `%token` and `%whites` directives plus the tokens used inside the body section.

```
%whites    =  /([ \t]*(?:#.*)?)/
%token NUM =  /([0-9]+(?:\.[0-9]+)?)/
%token VAR =  /([A-Za-z][A-Za-z0-9_]*)/
```

See section Automatic Generation of Lexical Analyzers for more details.

Declarations and Precedence

Lines 2-5 declare several tokens. The usual way to declare tokens is through the `%token` directive. The declarations `%nonassoc`, `%left` and `%right` not only declare the tokens but also associate a *priority* with them. Tokens declared in the same line have the same precedence. Tokens declared with these directives in lines below have more precedence than those declared above. Thus, in the example above we are saying that `"+"` and `"-"` have the same precedence but higher precedence than =. The final effect of `"-"` having greater precedence than = will be that an expression like:

```
a = 4 - 5
```

will be interpreted as

```
a = (4 - 5)
```

and not as

```
(a = 4) - 5
```

The use of the `%left` indicates that - in case of ambiguity and a match between precedences - the parser must build the tree corresponding to a left parenthesizing. Thus, the expression

```
4 - 5 - 9
```

will be interpreted as

```
(4 - 5) - 9
```

You can refer to the token end-of-input in the header section using the string `''` (for example to give it some priority, see the example in `examples/debuggingtut/typicalrrwithprec.eyp`).

Header Code

Perl code surrounded by `%{` and `%}` can be inserted in the head section. Such code will be inserted in the module generated by `eyapp` near the beginning. Therefore, declarations like the one of the calculator symbol table `%s`

```
7  %{
8  my %s; # symbol table
9  %}
```

will be visible from almost any point in the file.

The Start Symbol of the Grammar

`%start program` declares `program` as the start symbol of the grammar. When `%start` is not used, the first rule in the body section will be used.

Expect

The `%expect #NUMBER` directive works as in `bison` and suppress warnings when the number of Shift/Reduce conflicts is exactly `#NUMBER`.

The directive has been extended to be called with two numbers:

```
%expect NUMSHIFTRED NUMREDRED
```

no warnings will be emitted if the number of shift-reduce conflicts is exactly `NUMSHIFTRED` and the number of reduce-reduce conflicts is `NUMREDRED`.

Type and Union

C oriented declarations like `%type` and `%union` are parsed but ignored.

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 bugyapp2.eyp
     1  %strict
     2  %%
     3  expr: NUM;
     4  %%
pl@nereida:~/LEyapp/examples/eyapplanguageref$ eyapp bugyapp2.eyp
Warning! Non declared token NUM at line 3 of bugyapp2.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.

The %prefix Directive

The %prefix directive is equivalent to the use of the yyprefix. The node classes are prefixed with the specified prefix

```
%prefix Some::Prefix::
```

See the example in examples/eyapplanguageref/alias_and_yyprefix.pl. See also section Grammar Reuse in *Parse::Eyapp::defaultactionsintro* for an example that does not involve the %tree directive.

Default Action Directive

In Parse::Eyapp you can modify the default action using the %defaultaction { Perl code } directive. See section DEFAULT ACTIONS. The examples examples/eyapplanguageref/Postfix.eyp and examples/eyapplanguageref/I illustrate the use of the directive.

Tree Construction Directives

Parse::Eyapp facilitates the construction of concrete syntax trees and abstract syntax trees (abbreviated AST from now on) through the %tree and %metatree directives. See sections ABSTRACT SYNTAX TREES: %tree AND %name and *Parse::Eyapp::translationschemestut*.

Tokens and the Abstract Syntax Tree

The new token declaration directives %syntactic token and %semantic token can change the way eyapp builds the abstract syntax tree. See section Syntactic and Semantic tokens.

The %nocompact directive

This directive influences the generation of the LALR tables. They will not be compacted and the tokens for the DEFAULT reduction will be explicitly set. It can be used to produce an .output file (option -v) with more information.

4 THE BODY

The body section contains the rules describing the grammar:

```
body:   rules * '%%'
rules:  IDENT ':' rhss ';'
rhss:   (optname rhs (prec epscode)?) <+ '|'>
```

Rules

A rule is made of a left-hand-side symbol (the *syntactic variable*), followed by a ':' and one or more *right-hand-sides* (or *productions*) separated by '|' and terminated by a ';' like in:

```
exp:
      exp '+' exp
    | exp '-' exp
    | NUM
;
```

A *production* (*right hand side*) may be empty:

```
input:
      /* empty */
    | input line
;
```

The former two productions can be abbreviated as

```
input:
      line *
;
```

The operators `*`, `+` and `?` are presented in section LISTS AND OPTIONALS.

A *syntactic variable cannot appear more than once as a rule name* (This differs from `yacc`). So you can't write

```
thing: foo bar ;
thing: foo baz ;
```

instead, write:

```
thing:
       foo bar
     | foo baz
;
```

Semantic Values and Semantic Actions

In `Parse::Eyapp` a production rule

```
A -> X_1 X_2 ... X_n
```

can be followed by a *semantic action*:

```
A -> X_1 X_2 ... X_n { Perl Code }
```

Such semantic action is nothing but Perl code that will be treated as an anonymous subroutine. The semantic action associated with production rule `A -> X_1 X_2 ... X_n` is executed after any actions associated with the subtrees of `X_1`, `X_2`, ..., `X_n`. `Eyapp` parsers build the syntax tree using a left-right bottom-up traverse of the syntax tree. Each times the Parser visits the node associated with the production `A -> X_1 X_2 ... X_n` the associated semantic action is called. Associated with each symbol of a *Parse::Eyapp* grammar there is a scalar *Semantic Value* or *Attribute*. The semantic values of terminals are provided by the lexical analyzer. In the calculator example (see file `examples/eyapplanguageref/Calc.yp` in the distribution), the semantic value associated with an expression is its numeric value. Thus in the rule:

```
exp '+' exp { $_[1] + $_[3] }
```

`$_[1]` refers to the attribute of the first `exp`, `$_[2]` is the attribute associated with `'+'`, which is the second component of the pair provided by the lexical analyzer and `$_[3]` refers to the attribute of the second `exp`.

When the semantic action/anonymous subroutine is called, the arguments are as follows:

- `$_[1]` to `$_[n]` are the attributes of the symbols `X_1`, `X_2`, ..., `X_n`. Just as `$1` to `$n` in `yacc`,
- `$_[0]` is the parser object itself. Having `$_[0]` being the parser object itself allows you to call parser methods. Most `yacc` macros have been converted into parser methods. See section METHODS AVAILABLE IN THE GENERATED CLASS in *Parse::Eyapp*.

The returned value will be the attribute associated with the left hand side of the production.

Names can be given to the attributes using the dot notation (see file `examples/eyapplanguageref/CalcSimple.eyp`):

```
exp.left '+' exp.right { $left + $right }
```

See section NAMES FOR ATTRIBUTES for more details about the *dot* and *dollar* notations.

If no action is specified and no `%defaultaction` is specified the default action

```
{ $_[1] }
```

will be executed instead. See section DEFAULT ACTIONS to know more.

Actions in Mid-Rule

Actions can be inserted in the middle of a production like in:

```
block: '{'.bracket { $ids->begin_scope(); } declaration*.decs statement*.sts '}' { ... }
```

A middle production action is managed by inserting a new rule in the grammar and associating the semantic action with it:

```
Temp: /* empty */ { $ids->begin_scope(); }
```

Middle production actions can refer to the attributes on its left. They count as one of the components of the production. Thus the program:

```
~/LEyapp/examples/eyapplanguageref$ cat intermediateaction2.yp
%%
S:  'a' { $_[1]x4 }.mid 'a' { print "\n<<$_[2], $mid, $_[3]>>\n"; }
;
%%
```

The auxiliar syntactic variables are named `@#position-#order` where `#position` is the position of the action in the rhs and `order` is an ordinal number. See the `.output` file for the former example:

```
~/LEyapp/examples/eyapplanguageref$ eyapp -v intermediateaction2.yp
~/LEyapp/examples/eyapplanguageref$ sed -ne '1,5p' intermediateaction2.output
Rules:
------
0:  $start -> S $end
1:  S -> 'a' @1-1 'a'
2:  @1-1 -> /* empty */
```

We can build an executable `ia.pl` from the former grammar using `eyapp` option `-C`:

```
~/LEyapp/examples/eyapplanguageref$ eyapp -C -o ia.pl intermediateaction2.yp
```

The `main`, error and lexer methods are provided by `Parse::Eyapp`. When given input `aa` the execution will produce as output `aaaa, aaaa, a`. The option `-d` activates the debug mode, the option `-c` tells the program to get the input from the command line::

```
~/LEyapp/examples/eyapplanguageref$ ./ia.pl -d -c 'aa'
----------------------------------------
In state 0:
Stack: 0
Need token. Got >a<
Shift and go to state 2.
----------------------------------------
In state 2:
Stack: 0->'a'->2
Don't need token.
Reduce using rule 2 (@1-1 --> /* empty */): Back to state 2, then go to state 4.
----------------------------------------
In state 4:
Stack: 0->'a'->2->'@1-1'->4
Need token. Got >a<
Shift and go to state 5.
----------------------------------------
In state 5:
Stack: 0->'a'->2->'@1-1'->4->'a'->5
Don't need token.
Reduce using rule 1 (S --> a @1-1 a):
<<aaaa, aaaa, a>>
Back to state 0, then go to state 1.
----------------------------------------
```

```
In state 1:
Stack: 0->'S'->1
Need token. Got ><
Shift and go to state 3.
--------------------------------------
In state 3:
Stack: 0->'S'->1->''->3
Don't need token.
Accept.
```

Example of Body Section

Following with the calculator example, the body is:

```
pl@nereida:~/src/perl/eyapp/examples/eyapplanguageref$ sed -ne '17,/%%/p' Calc.eyp | cat -n
   1  start:
   2      input { \%s }
   3  ;
   4
   5  input: line *
   6  ;
   7
   8  line:
   9      '\n'       { undef }
  10    | exp '\n'   {
  11                     print "$_[1]\n" if defined($_[1]);
  12                     $_[1]
  13                 }
  14    | error  '\n'
  15        {
  16          $_[0]->YYErrok;
  17          undef
  18        }
  19  ;
  20
  21  exp:
  22      NUM
  23    | $VAR                    { $s{$VAR} }
  24    | $VAR '=' $exp           { $s{$VAR} = $exp }
  25    | exp.left '+' exp.right { $left + $right }
  26    | exp.left '-' exp.right { $left - $right }
  27    | exp.left '*' exp.right { $left * $right }
  28    | exp.left '/' exp.right
  29      {
  30         $_[3] and return($_[1] / $_[3]);
  31         $_[0]->YYData->{ERRMSG} = "Illegal division by zero.\n";
  32         $_[0]->YYError;
  33         undef
  34      }
  35    | '-' $exp %prec NEG     { -$exp }
  36    | exp.left '^' exp.right { $left ** $right }
  37    | '(' $exp ')'           { $exp }
  38  ;
  39
  40  %%
```

This body does not uses any of the Eyapp extensions (with the exception of the * operator at line 5) and the dot and dollar notations.

Solving Ambiguities and Conflicts

When Eyapp analyzes a grammar like:

```
examples/eyapplanguageref$ cat -n ambiguities.eyp
   1  %%
   2  exp:
   3      NUM
   4    | exp '-' exp
   5  ;
   6  %%
```

it will produce a warning announcing the existence of *shift-reduce* conflicts:

```
examples/eyapplanguageref$ eyapp ambiguities.eyp
1 shift/reduce conflict (see .output file)
State 5: reduce by rule 2: exp -> exp '-' exp (default action)
State 5: shifts:
  to state    3 with '-'
```

when `eyapp` finds warnings automatically produces a `.output` file describing the conflict.

What the warning is saying is that an expression like `exp '-' exp` (rule 2) followed by a minus `'-'` can be parsed in more than one way. If we have an input like `NUM - NUM - NUM` the activity of a LALR(1) parser (the family of parsers to which Eyapp belongs) consists of a sequence of *shift and reduce actions*. A *shift action* has as consequence the reading of the next token. A *reduce action* is finding a production rule that matches and substituting the rhs of the production by the lhs. For input `NUM - NUM - NUM` the activity will be as follows (the dot is used to indicate where the next input token is):

```
.NUM - NUM - NUM # shift
 NUM.- NUM - NUM # reduce exp: NUM
 exp.- NUM - NUM # shift
 exp -.NUM - NUM # shift
 exp - NUM.- NUM # reduce exp: NUM
 exp - exp.- NUM # shift/reduce conflict
```

up this point two different decisions can be taken: the next description can be

```
exp.- NUM # reduce by exp: exp '-' exp (rule 2)
```

or:

```
exp - exp -.NUM # shift '-' (to state 3)
```

that is why it is called a *shift-reduce conflict*.

That is also the reason for the precedence declarations in the head section. Another kind of conflicts are *reduce-reduce conflicts*. They arise when more that rhs can be applied for a reduction action.

Eyapp solves the conflicts applying the following rules:

- In a shift/reduce conflict, the default is the shift.
- In a reduce/reduce conflict, the default is to reduce by the earlier grammar production (in the input sequence).
- Precedences and associativities can be given to tokens in the declarations section. This is made by a sequence of lines beginning with one of the directives: `%left`, `%right`, or `%nonassoc`, followed by a list of tokens. All the tokens on the same line have the same precedence and associativity; the lines are listed in order of increasing precedence.
- A precedence and associativity is associated with each grammar production; it is the precedence and associativity of the *last token* or *literal* in the right hand side of the production.
- The `%prec` directive can be used when a rhs is involved in a conflict and has no tokens inside or it has but the precedence of the last token leads to an incorrect interpretation. A rhs can be followed by an optional `%prec token` directive giving the production the precedence of the `token`

```
exp:   '-' exp %prec NEG { -$_[1] }
```

- If there is a shift/reduce conflict, and both the grammar production and the input token have precedence and associativity associated with them, then the conflict is solved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and non associative implies error. The last 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.

To solve a shift-reduce conflict between a production `A -> SOMETHING` and a token `'a'` you can follow this procedure:

1. Edit the `.output` file
2. Search for the state where the conflict between the production and the token is. In our example it looks like:

   ```
   pl@nereida:~/src/perl/YappWithDefaultAction/examples$ sed -ne '56,65p' ambiguities.output
   State 5:

         exp -> exp . '-' exp    (Rule 2)
         exp -> exp '-' exp .    (Rule 2)

         '-'     shift, and go to state 3

         '-'     [reduce using rule 2 (exp)]
         $default        reduce using rule 2 (exp)
   ```

3. Inside the state there has to be a production of the type `A -> SOMETHING.` (with the dot at the end) indicating that a reduction must take place. There has to be also another production of the form `A -> prefix . suffix`, where suffix can *start* with the involved token `'a'`.
4. Decide what action shift or reduce matches the kind of trees you want. In this example we want `NUM - NUM - NUM` to produce a tree like `MINUS(MINUS(NUM, NUM), NUM)` and not `MINUS(NUM, MINUS(NUM, NUM))`. We want the conflict in `exp - exp.- NUM` to be solved in favor of the reduction by `exp: exp '-' exp`. This is achieved by declaring `%left '-'`.

Error Recovery

The token name `error` is reserved for error handling. This name can be used in grammar productions; it suggests places where errors are expected, and recovery can take place:

```
line:
  '\n'         { undef }
  | exp '\n'   { print "$_[1]\n" if defined($_[1]); $_[1] }
  | error  '\n'
      {
        $_[0]->YYErrok;
        undef
      }
```

The parser pops its stack until it enters a state where the token `error` is legal. It then shifts the token `error` and proceeds to discard tokens until finding one that is acceptable. In the example all the tokens until finding a `'\n'` will be skipped. If no special error productions have been specified, the processing will halt.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted. The method `YYErrok` used in the example communicates to the parser that a satisfactory recovery has been reached and that it can safely emit new error messages.

You cannot have a literal *'error'* in your grammar as it would confuse the driver with the *error* token. Use a symbolic token instead.

5 THE TAIL

The tail section contains Perl code. Usually it is empty, but you can if you want put here your own lexical analyzer and error management subroutines. An example of this is in files `examples/eyapplanguageref/List3_tree_d_sem.yp` (the grammar) and `use_list3_tree_d_dem.pl` (the client).

6 THE LEXICAL ANALYZER

The Lexical Analyzer is called each time the parser needs a new token. It is called with only one argument (the parser object) and returns a pair containing the next token and its associated attribute.

The fact that is a method of the parser object means that the parser methods are accessible inside the lexical analyzer.

When the lexical analyzer reaches the end of input, it must return the pair `('', undef)`

Automatic Generation of Lexical Analyzers

By default a lexical analyzer is built. The `eyapp` option `-l` can be used to inhibit the generation of the default lexical analyzer. In such case, one must be explictly provided.

No token Definitions

When no token definitions are given in the head section, the default lexical analyzer simply assumes that the token is the string literal. See this example in file `examples/lexergeneration/simple.yp`:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ cat simple.yp
%%
A:    a
    | A d
;
%%
```

The grammar does not describes the lexical analyzer nor the error default subroutine. Eyapp will generate default lexical and error subroutines:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ eyapp -o simple.pl -TC simple.yp

pl@nereida:~/LEyapp/examples/lexergeneration$ ls -ltr | tail -2
-rw-r--r-- 1 pl pl   27 2010-06-29 10:28 simple.yp
-rwxr-xr-x 1 pl pl 3245 2010-06-29 10:35 simple.pl
```

The option `-T` is equivalent to insert the `%tree` directive in the head section. Since no names were explicitly given to the productions, the names of the productions are built using the pattern `Lhs_is_RHS`.

Option `-C` instructs the `eyapp` compiler to produce an executable by setting the execution permits (see `simple.pl` permits above), inserting the appropriate she bang directive:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ head simple.pl | head -1
#!/usr/bin/perl
```

and inserting a call to the package `main` subroutine at the end of the generated parser:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ tail -6 simple.pl
unless (caller) {
  exit !__PACKAGE__->main('');
}
```

If no `main` was provided, `eyapp` will provide one.

Tokens `a` and `d` are assumed to represent strings `'a'` and `'d'` respectively.

```
pl@nereida:~/LEyapp/examples/lexergeneration$ ./simple.pl -i -t -c 'a d d'
A_is_A_d(A_is_A_d(A_is_a(TERMINAL[a]),TERMINAL[d]),TERMINAL[d])
```

The `main` method provided by `eyapp` accepts several options in the command line:

- `-t` Prints the abstract syntax tree
- `-i` Shows the semantic value associated with each terminal
- `-c string` Indicates that the input is given by the `string` that follows the option

You can get the set of available options using `-help`:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ ./simple.pl -h
Available options:
  --debug                   sets yydebug on
  --nodebug                 sets yydebug off
  --file filepath           read input from filepath
  --commandinput string     read input from string
  --tree                    prints $tree->str
  --notree                  does not print $tree->str
  --info                    When printing $tree->str shows the value of TERMINALs
  --help                    shows this help
  --slurp                   read until EOF reached
  --noslurp                 read until CR is reached
  --argfile                 main() will take the input string from its @_
  --noargfile               main() will not take the input string from its @_
  --yaml                    dumps YAML for $tree: YAML module must be installed
  --margin=i                controls the indentation of $tree->str (i.e. $Parse::Eyapp::Node::INDENT
```

If we try to feed it with an illegal input, an error message is emitted:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ ./simple.pl -i -t -c 'a e d'
Error inside the lexical analyzer near 'e'. Line: 1. File: 'simple.yp'. No match found.
```

In the example above we have taken advantage of the `main` method provided by Eyapp. If we want to keep in control of the parsing process, we can write a client program that makes use of the generated modulino:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ cat -n usesimple.pl
   1  #!/usr/bin/env perl
   2  use warnings;
   3  use strict;
   4
   5  use simple;
   6
   7  # build a parser object
   8  my $parser = simple->new();
   9
  10  # take the input from the command line arguments
  11  # or from STDIN
  12  my $input = join ' ',@ARGV;
  13  $input = <> unless $input;
  14
  15  # set the input
  16  $parser->input($input);
  17
  18  # parse the input and get the AST
  19  my $tree = $parser->YYParse();
  20
  21  print $tree->str()."\n";
```

Here is an example of execution:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ eyapp -T simple.yp
pl@nereida:~/LEyapp/examples/lexergeneration$ ./usesimple.pl a d d
A_is_A_d(A_is_A_d(A_is_a(TERMINAL),TERMINAL),TERMINAL)
```

Token Definitions: Regular Expressions

Eyapp extends the `%token` directive with the syntax:

```
%token TOKENID = /regexp/
```

If such definition is used, an entry with the shape:

```
/\G$regexp/gc and return ('TOKENID', $1);
```

will be added to the generated lexical analyzer. Therefore the string associated with the first parenthesis in `/regexp/` will be used as semantic value for `TOKENID`. If `/regexp/` has no parenthesis `undef` will be the semantic value. See this example:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ cat -n numlist.eyp
   1  %token NUM = /(\d+)/
   2  %token ID  = /(\w+)/
   3
   4  %%
   5  A:
   6        B
   7      | A B
   8  ;
   9
  10  B:
  11        ID
  12      | a
  13      | NUM
  14  ;
  15  %%
```

The order of the `%token` declarations is important. In the example the token `NUM` is a subset of the token `ID`. Since it appears first, it will be tried first:

```
/\G(\d+)/gc and return ('NUM', $1);
/\G(\w+)/gc and return ('ID', $1);
```

Also observe that token `'a'` (line 12) is contained in token `ID`. However, any implicit token like this that appears in the body section and was not declared using an explicit `%token` directive in the head section takes priority over the ones declared. See the behavior of the former program:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ eyapp -TC numlist
pl@nereida:~/LEyapp/examples/lexergeneration$ ./numlist.pm -t -i -c '4 a b'
A_is_A_B(A_is_A_B(A_is_B(B_is_NUM(TERMINAL[4])),B_is_a(TERMINAL[a])),B_is_ID(TERMINAL[b]))
```

The lexical analyzer returned `NUM` and not `ID` when `4` was processed, also it returned `a` and not `ID` when `'a'` was processed.

A `%token` declaration without assignment like in:

```
%token A B
```

is equivalent to

```
%token A   = /(A)/
%token B   = /(B)/
```

(in that order).

Token Definitions via Code

An alternative way to define a token is via Perl code:

```
%token TOKENID = { ... }
```

in such case the code defining `TOKENID` will be inserted verbatim in the corresponding place of the generated lexical analyzer. When the code `{ ... }` is executed, the variable `$_` contains the input being parsed and the special variable `$self` refers to the parser object. The following example is equivalent to the one used in the previous section:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ cat -n tokensemdef.eyp
   1  %token NUM = /(\d+)/
   2  %token ID  = { /\G(\w+)/gc and return ('ID', $1); }
   3
   4  %%
   5  A:
   6        B
   7      | A B
   8  ;
   9
  10  B:
  11        ID
  12      | a
  13      | NUM
  14  ;
  15  %%
```

Follows an example of compilation and execution:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ eyapp -TC tokensemdef.eyp
pl@nereida:~/LEyapp/examples/lexergeneration$ ./tokensemdef.pm -t -i -nos
4 a b
A_is_A_B(A_is_A_B(A_is_B(B_is_NUM(TERMINAL[4])),B_is_a(TERMINAL[a])),B_is_ID(TERMINAL[b]))
```

Token Definitions: Controling whites

By default, the generated lexical analyzer skips white spaces, defined as `/\s*/`. The programmer can change this behavior using the `%whites` directive. The following example permits Perl-like comments in the input:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ cat -n simplewithwhites.eyp
   1  %whites  /(\s*(?:#.*)?\s*)/
   2  %%
   3  A:    a
   4      | A d
   5  ;
   6  %%
```

Follows an example of execution:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ cat -nA input
     1  a # 1$
     2  $
     3  d ^I#2$
pl@nereida:~/LEyapp/examples/lexergeneration$ eyapp -TC simplewithwhites.eyp
pl@nereida:~/LEyapp/examples/lexergeneration$ ./simplewithwhites.pm -t -i -f input
A_is_A_d(A_is_a(TERMINAL[a]),TERMINAL[d])
```

The `%white` directive can be followed by some Perl code defining the white spaces:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ cat -n simplewithwhitescode.eyp
   1  %whites  { /\G(\s*(?:#.*)?\s*)/gc and $self->tokenline($1 =~ tr{\n}{}) }
   2  %%
   3  A:    a
   4      | A d
   5  ;
   6  %%
```

Reading Input from File

You can use the method `YYSlurpFile` to read the input from a file and set the input for the parser to its contents. Yo can also use the `YYInput` method to set the input.

See the example below:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ cat -n usesimplefromfile.pl
   1  #!/usr/bin/env perl
   2  use warnings;
   3  use strict;
   4
   5  use simplewithwhites;
   6
   7  my $parser = simplewithwhites->new();
   8
   9  # take the input from the specified file
  10  my $fn = shift;
  11
  12  $parser->YYSlurpFile($fn);
  13
  14  # parse the input and get the AST
  15  my $tree = $parser->YYParse();
  16
  17  print $tree->str()."\n";
```

First, compile the grammar `simplewithwhites.eyp` presented above:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ eyapp -T simplewithwhites
```

And then run it:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ cat -n  input
   1  a # 1
   2
   3  d       #2
pl@nereida:~/LEyapp/examples/lexergeneration$ ./usesimplefromfile.pl input
A_is_A_d(A_is_a(TERMINAL),TERMINAL)
```

Huge input and Incremental Lexical Analyzers

If your input is huge, try to make use of an incremental lexical analyzer. In an incremental lexer the input is read and parsed in chunks. Read up to a point where it is safe to parse. In the example below, the lexer reads a new line each time we reach the end of the input string `${$parser->YYInput}`. In the case of the arithmetic expressions grammar below, by reading up to `'\n'`, we are sure that the input is not broken in the middle of a token. Instead of having the whole huge input in memory, we only keep a small substring.

```
pl@nereida:~/LEyapp/examples/lexergeneration$ cat -n Incremental.eyp
   1  %right  '='
   2  %left   '-' '+'
   3  %left   '*' '/'
   4  %left   NEG
   5
   6  %tree
   7
   8  %%
   9  input:
  10          |   input $line  { print $line->str."\n" }
  11  ;
  12
  13  line:     '\n'
  14          | exp '\n'
  15          | error '\n'
```

```
 16  ;
 17
 18  exp:        NUM
 19          |   VAR
 20          |   VAR '=' exp
 21          |   exp '+' exp
 22          |   exp '-' exp
 23          |   exp '*' exp
 24          |   exp '/' exp
 25          |   '-' exp %prec NEG
 26          |   '(' exp ')'
 27  ;
 28
 29  %%
 30
 31  sub _Lexer {
 32      my($parser)=shift;
 33
 34      if ($parser->YYEndOfInput) {
 35        my $input = <STDIN>;
 36        return('', undef) unless $input;
 37        $parser->input($input);
 38      };
 39
 40      for (${$parser->YYInput}) {
 41          m/\G[ \t]*/gc;
 42          m/\G([0-9]+(?:\.[0-9]+)?)/gc and return('NUM',$1);
 43          m/\G([A-Za-z][A-Za-z0-9_]*)/gc and return('VAR',$1);
 44          m/\G(.)/gcs and return($1,$1);
 45          return('', undef);
 46      }
 47  }
 48
 49  __PACKAGE__->lexer(\&_Lexer);
```

This approach has limitations. The code will get more tangled if some token can take more than one line. For example, if we extend this language to accept C-like comments `/* ... */` which expands over several lines.

Here follows an example of execution. This is the client program:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ cat useincremental.pl
#!/usr/bin/perl -w
use Incremental;

Incremental->new->YYParse;
```

This is a small test input file:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ cat inputforincremental
a = 2
a+3
b=4
b*2+a
```

Finally, see the results of the execution:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ ./useincremental.pl < inputforincremental
line_4(exp_8(TERMINAL,exp_6(TERMINAL)))
line_4(exp_9(exp_7(TERMINAL),exp_6(TERMINAL)))
line_4(exp_8(TERMINAL,exp_6(TERMINAL)))
line_4(exp_9(exp_11(exp_7(TERMINAL),exp_6(TERMINAL)),exp_7(TERMINAL)))
```

The numbers in the output refer to the production number:

```
pl@nereida:~/LEyapp/examples/lexergeneration$ eyapp -v Incremental.eyp
pl@nereida:~/LEyapp/examples/lexergeneration$ sed -ne '/Rules:/,/^$/p' Incremental.output
Rules:
------
0:      $start -> input $end
1:      input -> /* empty */
2:      input -> input line
3:      line -> '\n'
4:      line -> exp '\n'
5:      line -> error '\n'
6:      exp -> NUM
7:      exp -> VAR
8:      exp -> VAR '=' exp
9:      exp -> exp '+' exp
10:     exp -> exp '-' exp
11:     exp -> exp '*' exp
12:     exp -> exp '/' exp
13:     exp -> '-' exp
14:     exp -> '(' exp ')'
```

Using Several Lexical Analyzers for the Same Parser

At any time during the parsing you can use the method `$parser->YYLexer` to set a new lexical analyzer.

The following grammar starts setting the lexer to sub `Lexer1` (line 44). It later changes the lexer to `Lexer2` (ine 24) after the token `'%%'` is seen. Inside `Lexer2` the token `A` represents a `'B'`. This capability allows the parsing of languages where different sections require different lexical analysis. For example, in `yacc`, carriage returns separates declarations in the header section but is considered a white space inside the body and tail sections. This feature has similar power to the *state* concept of the lexical analyzer generator `flex`.

```
$ cat -n twolexers.eyp
   1  %%
   2  s:  first '%%' second
   3  ;
   4
   5  first:
   6      A first
   7    | A
   8  ;
   9
  10  second:
  11      A second
  12    | A
  13  ;
  14
  15  %%
  16
  17  sub Lexer1 {
  18      my($parser)=shift;
  19
  20      print "In Lexer 1 \n";
  21      for (${$parser->YYInput}) {
  22          m/\G\s*/gc;
  23          m/\G(%%)/gc and do {
  24            $parser->YYLexer(\&Lexer2);
  25            return ($1, undef);
  26          };
  27          m/\G(.)/gcs and return($1,$1);
  28          return('', undef);
  29      }
  30  }
  31
```

```
32  sub Lexer2 {
33      my($parser)=shift;
34
35      print "In Lexer 2 \n";
36      for (${$parser->YYInput}) {
37          m/\G\s*/gc;
38          m/\GB/gc    and return('A','B');
39          m/\G(.)/gcs and die "Error. Expected 'B', found $1\n";
40      }
41          return('', undef);
42  }
43
44  __PACKAGE__->lexer(\&Lexer1);
```

When executed, it behaves like this:

```
$ ./twolexers.pm -t -i -m 1 -c 'A A %% B B'
In Lexer 1
In Lexer 1
In Lexer 1
In Lexer 2
In Lexer 2
In Lexer 2

s_is_first_second(
  first_is_A_first(
    TERMINAL[A],
    first_is_A(
      TERMINAL[A]
    )
  ),
  second_is_A_second(
    TERMINAL[B],
    second_is_A(
      TERMINAL[B]
    )
  )
)
```

The lexer can bechanged at any time. The following example starts using the default lexer generated by `eyapp`. It changes the lexer to `Lexer2`inside an intermediate semantic action (line 7). Inside `Lexer2` the token `A` is interpreted as a word `\w+`.

```
$ cat -n twolexers2.eyp
  1  # Compile it with:
  2  # $ eyapp -TC twolexers2.eyp
  3  # Run it with:
  4  # $ ./twolexers2.pm -t -i -c 'A A %% d3 c2'
  5
  6  %%
  7  s:  first '%%' { $_[0]->YYLexer(\&Lexer2) } second
  8  ;
  9
 10  first:
 11      A first
 12    | A
 13  ;
 14
 15  second:
 16      A second
 17    | A
 18  ;
```

```
 19
 20  %%
 21
 22  sub Lexer2 {
 23      my($parser)=shift;
 24
 25      print "In Lexer 2 \n";
 26      for (${$parser->YYInput}) {
 27          m/\G\s*/gc;
 28          m/\G(\w+)/gc    and return('A',$1);
 29          m/\G(.)/gcs and die "Error. Expected a word,Found $1\n";
 30      }
 31      return('', undef);
 32  }
```

7 THE ERROR REPORT SUBROUTINE

The Error Report subroutine is also a parser attribute, and must be defined. By default `Parse::Eyapp` provides a convenient error handler.

See the *Parse::Yapp* pages and elsewhere documentation on `yacc` and `bison` for more information.

8 USING AN EYAPP GRAMMAR

The following is an example of a program that uses the calculator explained in the two previous sections:

```
pl@nereida:~/LEyapp/examples/eyapplanguageref$ cat -n usecalc.pl
   1  #!/usr/bin/perl -w
   2  use strict;
   3  use Calc;
   4
   5  my $parser = Calc->new();
   6  $parser->input(\<<'EOI'
   7  a = 2*3       # 1: 6
   8  d = 5/(a-6)   # 2: division by zero
   9  b = (a+1)/7   # 3: 1
  10  c=a*3+4)-5    # 4: syntax error
  11  a = a+1       # 5: 7
  12  EOI
  13  );
  14  my $t = $parser->Run();
  15  print "========= Symbol Table ==============\n";
  16  print "$_ = $t->{$_}\n" for sort keys %$t;
```

The output for this program is (the input for each output appear as a Perl comment on the right):

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ eyapp Calc.eyp

pl@nereida:~/LEyapp/examples/eyapplanguageref$ ./usecalc.pl
6
Illegal division by zero.
1

Syntax error near ')' (line number 4).
Expected one of these terminals: '-' '/' '^' '*' '+' '
'
7
========= Symbol Table ==============
a = 7
b = 1
c = 22
```

9 LISTS AND OPTIONALS

The elements of the right hand side of a production (abbreviated *rhs*) can be one of these:

```
rhselt:
      symbol
    | code
    | '(' optname rhs ')'
    | rhselt STAR               /* STAR   is (%name\s*([A-Za-z_]\w*)\s*)?\*  */
    | rhselt '<' STAR symbol '>'
    | rhselt OPTION             /* OPTION is (%name\s*([A-Za-z_]\w*)\s*)?\?  */
    | rhselt '<' PLUS symbol '>'
    | rhselt PLUS               /* PLUS   is (%name\s*([A-Za-z_]\w*)\s*)?\+  */
```

The `STAR`, `OPTION` and `PLUS` operators provide a simple mechanism to express lists:

- In Eyapp the `+` operator indicates one or more repetitions of the element to the left of `+`, thus a rule like:

```
decls:  decl +
```

  is the same as:

```
decls:  decls decl
     |  decl
```

  An additional symbol may be included to indicate lists of elements separated by such symbol. Thus

```
rhss: rule <+ '|'>
```

  is equivalent to:

```
rhss: rhss '|' rule
    | rule
```

- The operators `*` and `?` have their usual meaning: 0 or more for `*` and optionality for `?`. Is legal to parenthesize a `rhs` expression as in:

```
optname: (NAME IDENT)?
```

The + operator

The grammar:

```
~/LEyapp/examples/eyapplanguageref$ cat List3.yp
%semantic token 'c'
%{
use Data::Dumper;
$Data::Dumper::Indent = 1;
%}
%%
S:      'c'+  'd'+
           {
              print Dumper($_[1]);
              print Dumper($_[2]);
           }
;
%%
```

Is equivalent to:

```
~/LEyapp/examples/eyapplanguageref$ eyapp -v List3.yp; head -9 List3.output
Rules:
------
0:  $start -> S $end
1:  PLUS-1 -> PLUS-1 'c'
2:  PLUS-1 -> 'c'
3:  PLUS-2 -> PLUS-2 'd'
4:  PLUS-2 -> 'd'
5:  S -> PLUS-1 PLUS-2
```

By default, the semantic action associated with a `+` returns the lists of attributes to which the `+` applies:

```
pl@nereida:~/LEyapp/examples/eyapplanguageref$ ./use_list3.pl
Try input 'ccdd': ccdd
$VAR1 = [ 'c', 'c' ];
$VAR1 = [ 'd', 'd' ];
```

Observe that, in spite of `'d'` being a syntactic token the actions related with the `d+` element (i.e. the actions associated with the `PLUS-2` productions) create the list of `d`s.

The semantic associated with a `+` changes when one of the tree creation directives is active (for instance `%tree` or `%metatree`) or it has been explicitly requested with a call to the `YYBuildingTree` method:

```
$self->YYBuildingTree(1);
```

Other ways to change the associated semantic are to use the `yybuildingtree` option of `YYParse`:

```
$self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
                  yybuildingtree => 1,
                # yydebug => 0x1F
);
```

In such case the associated semantic action creates a node labelled

```
_PLUS_LIST
```

whose children are the attributes associated with the items in the plus list. As it happens when using the `%tree` directive, *syntactic tokens* are skipped.

When executing the example above but under the `%tree` directive the output changes. The `-T` option tells the `eyapp` compiler to introduce an implicit `%tree` directive>:

```
~/LEyapp/examples/eyapplanguageref$ eyapp -T List3.yp
```

If we now run the client program with input `ccdd` we get a couple of syntax trees:

```
~/LEyapp/examples/eyapplanguageref$ ./use_list3.pl
Try input 'ccdd': ccdd
$VAR1 = bless( {
  'children' => [
    bless( { 'children' => [], 'attr' => 'c', 'token' => 'c' }, 'TERMINAL' ),
    bless( { 'children' => [], 'attr' => 'c', 'token' => 'c' }, 'TERMINAL' )
  ]
}, '_PLUS_LIST' );
$VAR1 = bless( { 'children' => [] }, '_PLUS_LIST' );
```

The node associated with the list of `d`s is empty since terminal `d` wasn't declared semantic.

When Nodes Disappear from Lists

When under the influence of the `%tree` directive the action associated with a list operator is to *flat* the children in a single list.

In the former example, the `d` nodes don't show up since `'d'` is a syntactic token. However, it may happen that changing the status of `'d'` to semantic will not suffice.

When inserting the children, the tree (`%tree`) node construction method (`YYBuildAST`) omits any attribute that is not a reference. Therefore, when inserting explicit actions, it is necessary to guarantee that the returned value is a reference or a semantic token to assure the presence of the value in the lists of children of the node. Certainly you can use this property to prune parts of the tree. Consider the following example:

```
~/LEyapp/examples/eyapplanguageref$ cat ListWithRefs1.eyp
%semantic token 'c' 'd'
%{
use Data::Dumper;
$Data::Dumper::Indent = 1;
%}
%%
S:      'c'+  D+
           {
              print Dumper($_[1]);
              print $_[1]->str."\n";
              print Dumper($_[2]);
              print $_[2]->str."\n";
           }
;

D: 'd'
;

%%

sub Run {
  my ($self) = shift;
  return $self->YYParse( yybuildingtree => 1 );
}
```

To activate the *tree semantic* for lists we use the `yybuildingtree` option of `YYParse` (line 26).
The execution gives an output like this:

```
pl@nereida:~/LEyapp/examples/eyapplanguageref$ eyapp ListWithRefs1.eyp; ./use_listwithrefs1.pl
Try input 'ccdd': ccdd
$VAR1 = bless( {
  'children' => [
    bless( { 'children' => [], 'attr' => 'c', 'token' => 'c' }, 'TERMINAL' ),
    bless( { 'children' => [], 'attr' => 'c', 'token' => 'c' }, 'TERMINAL' )
  ]
}, '_PLUS_LIST' );
_PLUS_LIST(TERMINAL,TERMINAL)
$VAR1 = bless( { 'children' => [] }, '_PLUS_LIST' ); _PLUS_LIST
```

Though `'d'` was declared semantic the default action associated with the production `D: 'd'` in line 16 returns `$_[1]` (that is, the scalar `'d'`). Since it is not a reference it won't be inserted in the list of children of `_PLUS_LIST`.

Recovering the Missing Nodes

The solution is to be sure that the attribute is a reference:

```
~/LEyapp/examples/eyapplanguageref$ cat -n ListWithRefs.eyp
   1  %semantic token 'c'
   2  %{
   3  use Data::Dumper;
   4  $Data::Dumper::Indent = 1;
   5  %}
   6  %%
   7  S:  'c'+  D+
   8        {
   9           print Dumper($_[1]);
  10           print Dumper($_[2]);
  11        }
  12  ;
  13
```

```
14  D: 'd'
15       {
16         bless { attr => $_[1], children =>[]}, 'DES';
17       }
18  ;
19
20  %%
21
22  sub Run {
23    my ($self) = shift;
24    return $self->YYParse( yybuildingtree => 1 );
25  }
```

Now the attribute associated with `D` is a reference and appears in the list of children of `_PLUS_LIST`:

```
~/LEyapp/examples/eyapplanguageref$ eyapp ListWithRefs.eyp; ./use_listwithrefs.pl
Try input 'ccdd': ccdd
$VAR1 = bless( {
  'children' => [
    bless( { 'children' => [], 'attr' => 'c', 'token' => 'c' }, 'TERMINAL' ),
    bless( { 'children' => [], 'attr' => 'c', 'token' => 'c' }, 'TERMINAL' )
  ]
}, '_PLUS_LIST' );
$VAR1 = bless( {
  'children' => [
    bless( { 'children' => [], 'attr' => 'd' }, 'DES' ),
    bless( { 'children' => [], 'attr' => 'd' }, 'DES' )
  ]
}, '_PLUS_LIST' );
```

Building a Tree with `Parse::Eyapp::Node->new`

The former solution consisting on writing *by hand* the code to build the node may suffice when dealing with a single node. Writing by hand the code to build a node is a cumbersome task. Even worst: though the node built in the former example looks like a `Parse::Eyapp` node actually isn't. `Parse::Eyapp` nodes always inherit from `Parse::Eyapp::Node` and consequently have access to the methods in such package. The following execution using the debugger illustrates the point:

```
pl@nereida:~/LEyapp/examples$ perl -wd use_listwithrefs.pl

Loading DB routines from perl5db.pl version 1.28
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(use_listwithrefs.pl:4):  $parser = new ListWithRefs();
  DB<1>  f ListWithRefs.eyp
1       2       #line 3 "ListWithRefs.eyp"
3
4:      use Data::Dumper;
5
6       #line 7 "ListWithRefs.eyp"
7       #line 8 "ListWithRefs.eyp"
8
9:                  print Dumper($_[1]);
10:                 print $_[1]->str."\n";
```

through the command `f ListWithRefs.eyp` we inform the debugger that subsequent commands will refer to such file. Next we execute the program up to the semantic action associated with the production rule `S: 'c'+ D+` (line 9)

```
  DB<2> c 9     # Continue up to line 9 of ListWithRefs.eyp
ccdd
ListWithRefs::CODE(0x84ebe5c)(ListWithRefs.eyp:9):
9:                    print Dumper($_[1]);
```

Now we are in condition to look at the contents of the arguments:

```
  DB<3> x $_[2]->str
0  '_PLUS_LIST_2(DES,DES)'
  DB<4> x $_[2]->child(0)
0  DES=HASH(0x85c4568)
   'attr' => 'd'
   'children' => ARRAY(0x85c458c)
        empty array
```

the `str` method works with the object `$_[2]` since `_PLUS_LIST_2` nodes inherit from `Parse::Eyapp::Node`. However, when we try with the `DES` node we get an error:

```
  DB<6> x $_[2]->child(0)->str
Can't locate object method "str" via package "DES" at \
  (eval 11)[/usr/share/perl/5.8/perl5db.pl:628] line 2, <STDIN> line 1.
  DB<7>
```

More robust than the former solution of building the node *by hand* is to use the constructor `Parse::Eyapp::Node->new`: The method `Parse::Eyapp::Node->new` is uset to build forests of syntactic trees.

It receives a list of terms describing the trees and - optionally - a reference to a subroutine used to set up the attributes of the just created nodes. After the creation of the trees the sub is called by `Parse::Eyapp::Node->new` with arguments the list of references to the nodes (in the order in which they appear in the terms, from left to right). `Parse::Eyapp::Node->new` returns a list of references to the just created nodes. In a scalar context returns a reference to the first of such trees. See an example:

```
~/LEyapp/examples$ perl -MParse::Eyapp -MData::Dumper -wde 0
main::(-e:1):   0
  DB<1> @t = Parse::Eyapp::Node->new('A(C,D) E(F)', sub { my $i = 0; $_->{n} = $i++ for @_ })
  DB<2> $Data::Dumper::Indent = 0
  DB<3> print Dumper($_)."\n" for @t
$VAR1 = bless( {'n' => 0,'children' => [bless( {'n' => 1,'children' => []}, 'C' ),
                                        bless( {'n' => 2,'children' => []}, 'D' )
                                       ]
               }, 'A' );
$VAR1 = bless( {'n' => 1,'children' => []}, 'C' );
$VAR1 = bless( {'n' => 2,'children' => []}, 'D' );
$VAR1 = bless( {'n' => 3,'children' => [bless( {'n' => 4,'children' => []}, 'F' )]}, 'E' );
$VAR1 = bless( {'n' => 4,'children' => []}, 'F' );
```

See the following example in which the nodes associated with `'d'` are explicitly constructed:

```
~/LEyapp/examples/eyapplanguageref$ cat -n ListWithRefs2.eyp
   1  %semantic token 'c'
   2  %{
   3  use Data::Dumper;
   4  $Data::Dumper::Indent = 1;
   5  %}
   6  %%
   7  S:      'c'+  D+
   8              {
   9                 print Dumper($_[1]);
  10                 print $_[1]->str."\n";
  11                 print Dumper($_[2]);
  12                 print $_[2]->str."\n";
  13              }
  14  ;
```

```
 15
 16  D: 'd'.d
 17       {
 18         Parse::Eyapp::Node->new(
 19           'DES(TERMINAL)',
 20            sub {
 21              my ($DES, $TERMINAL) = @_;
 22              $TERMINAL->{attr} = $d;
 23            }
 24         );
 25       }
 26  ;
 27
 28  %%
 29
 30  sub Run {
 31    my ($self) = shift;
 32    return $self->YYParse( yybuildingtree => 1 );
 33  }
```

To know more about `Parse::Eyapp::Node->new` see the section for `Parse::Eyapp::Node->new`
When the former eyapp program is executed produces the following output:

```
$ eyapp ListWithRefs2.eyp; use_listwithrefs2.pl
ccdd
$VAR1 = bless( {
  'children' => [
    bless( { 'children' => [], 'attr' => 'c', 'token' => 'c' }, 'TERMINAL' ),
    bless( { 'children' => [], 'attr' => 'c', 'token' => 'c' }, 'TERMINAL' )
  ]
}, '_PLUS_LIST_1' );
_PLUS_LIST_1(TERMINAL,TERMINAL)
$VAR1 = bless( {
  'children' => [
    bless( {
      'children' => [
        bless( { 'children' => [], 'attr' => 'd' }, 'TERMINAL' )
      ]
    }, 'DES' ),
    bless( {
      'children' => [
        bless( { 'children' => [], 'attr' => 'd' }, 'TERMINAL' )
      ]
    }, 'DES' )
  ]
}, '_PLUS_LIST_2' );
_PLUS_LIST_2(DES(TERMINAL),DES(TERMINAL))
```

The * operator

Any list operator operates on the factor to its left. A list in the right hand side of a production rule counts as a single symbol.

Both operators `*` and `+` can be used with the format `X <* Separator>`. In such case they describe lists of `X`s separated by `separator`. See an example:

```
pl@nereida:~/LEyapp/examples$ head -25 CsBetweenCommansAndD.eyp | cat -n
 1  # CsBetweenCommansAndD.eyp
 2
 3  %semantic token 'c' 'd'
 4
 5  %{
```

```
 6  sub TERMINAL::info {
 7    $_[0]->attr;
 8  }
 9  %}
10  %tree
11  %%
12  S:
13      ('c' <* ',' > 'd')*
14        {
15           print "\nNode\n";
16           print $_[1]->str."\n";
17           print "\nChild 0\n";
18           print $_[1]->child(0)->str."\n";
19           print "\nChild 1\n";
20           print $_[1]->child(1)->str."\n";
21           $_[1]
22        }
23  ;
24
25  %%
```

The rule

```
S: ('c' <* ',' > 'd')*
```

has only two items in its right hand side: the (separated by commas) list of `c`s and the list of `d`s. The production rule is equivalent to:

```
pl@nereida:~/LEyapp/examples$ eyapp -v CsBetweenCommansAndD.eyp
pl@nereida:~/LEyapp/examples$ head -11 CsBetweenCommansAndD.output | cat -n
 1  Rules:
 2  ------
 3  0:      $start -> S $end
 4  1:      STAR-1 -> STAR-1 ',' 'c'
 5  2:      STAR-1 -> 'c'
 6  3:      STAR-2 -> STAR-1
 7  4:      STAR-2 -> /* empty */
 8  5:      PAREN-3 -> STAR-2 'd'
 9  6:      STAR-4 -> STAR-4 PAREN-3
10  7:      STAR-4 -> /* empty */
11  8:      S -> STAR-4
```

The semantic action associated with `*` is to return a reference to a list with the attributes of the matching items.

When working -as in the example - under a tree creation directive it returns a node belonging to a class named `_STAR_LIST_#number` whose children are the items in the list. The `#number` is the ordinal number of the production rule as it appears in the `.output` file. The attributes must be references or associated with semantic tokens to be included in the list. Notice -in the execution of the former example that follows - how the node for `PAREN-3` has been eliminated from the tree. Parenthesis nodes are - generally - obviated:

```
pl@nereida:~/LEyapp/examples$ use_csbetweencommansandd.pl
c,c,cd

Node
_STAR_LIST_4(_STAR_LIST_1(TERMINAL[c],TERMINAL[c],TERMINAL[c]),TERMINAL[d])

Child 0
_STAR_LIST_1(TERMINAL[c],TERMINAL[c],TERMINAL[c])

Child 1
TERMINAL[d]
```

Notice that the comma (since it is a syntactic token) has also been suppressed.

Giving Names to Lists

To set the name of the node associated with a list operator the `%name` directive must precede the operator as in the following example:

```
pl@nereida:~/LEyapp/examples/eyapplanguageref$ sed -ne '1,27p' CsBetweenCommansAndDWithNames.eyp | ca
 1  # CsBetweenCommansAndDWithNames.eyp
 2
 3  %semantic token 'c' 'd'
 4
 5  %{
 6  sub TERMINAL::info {
 7    $_[0]->attr;
 8  }
 9  %}
10  %tree
11  %%
12  Start: S
13  ;
14  S:
15      ('c' <%name Cs * ','> 'd') %name Cs_and_d *
16        {
17           print "\nNode\n";
18           print $_[1]->str."\n";
19           print "\nChild 0\n";
20           print $_[1]->child(0)->str."\n";
21           print "\nChild 1\n";
22           print $_[1]->child(1)->str."\n";
23           $_[1]
24        }
25  ;
26
27  %%
```

The grammar describes the language of sequences

```
c,...,cd c,...,cd c,...,cd ....
```

The right hand side of the production has only one term which is a list, but the factor to which the star applies is itself a list. We are naming the term with the name `Cs_and_d` and the factor with the name `Cs`.

The execution shows the renamed nodes:

```
pl@nereida:~/LEyapp/examples/eyapplanguageref$ use_csbetweencommansanddwithnames.pl
c,c,c,cd

Node
Cs_and_d(Cs(TERMINAL[c],TERMINAL[c],TERMINAL[c],TERMINAL[c]),TERMINAL[d])

Child 0
Cs(TERMINAL[c],TERMINAL[c],TERMINAL[c],TERMINAL[c])

Child 1
TERMINAL[d]
```

Optionals

The `X?` operator stands for the presence or omission of `X`.

The grammar:

```
pl@nereida:~/LEyapp/examples$ head -11 List5.yp | cat -n
     1  %semantic token 'c'
     2  %tree
     3  %%
```

```
 4  S: 'c' 'c'?
 5       {
 6         print $_[2]->str."\n";
 7         print $_[2]->child(0)->attr."\n" if $_[2]->children;
 8       }
 9  ;
10
11  %%
```

is equivalent to:

```
pl@nereida:~/LEyapp/examples$ eyapp -v List5
pl@nereida:~/LEyapp/examples$ head -7 List5.output
Rules:
------
0:      $start -> S $end
1:      OPTIONAL-1 -> 'c'
2:      OPTIONAL-1 -> /* empty */
3:      S -> 'c' OPTIONAL-1
```

When `yybuildingtree` is false the associated attribute is a list that will be empty if CX> does not show up.

Under the `%tree` directive the action creates an `_OPTIONAL` node:

```
pl@nereida:~/LEyapp/examples$ use_list5.pl
cc
_OPTIONAL_1(TERMINAL)
c
pl@nereida:~/LEyapp/examples$ use_list5.pl
c
_OPTIONAL_1
```

Parenthesis

Any substring on the right hand side of a production rule can be grouped using a parenthesis. The introduction of a parenthesis implies the introduction of an additional syntactic variable whose only production is the sequence of symbols between the parenthesis. Thus the grammar:

```
pl@nereida:~/LEyapp/examples$ head -6 Parenthesis.eyp | cat -n
   1  %%
   2  S:
   3        ('a' S ) 'b'  { shift; [ @_ ] }
   4      | 'c'
   5  ;
   6  %%
```

is equivalent to:

```
pl@nereida:~/LEyapp/examples$ eyapp -v Parenthesis.eyp; head -6 Parenthesis.output
Rules:
------
0:      $start -> S $end
1:      PAREN-1 -> 'a' S
2:      S -> PAREN-1 'b'
3:      S -> 'c'
```

By default the semantic rule associated with a parenthesis returns an anonymous list with the attributes of the symbols between the parenthesis:

```
pl@nereida:~/LEyapp/examples$ cat -n use_parenthesis.pl
     1  #!/usr/bin/perl -w
     2  use Parenthesis;
     3  use Data::Dumper;
```

```
     4
     5  $Data::Dumper::Indent = 1;
     6  $parser = Parenthesis->new();
     7  print Dumper($parser->Run);
pl@nereida:~/LEyapp/examples$ use_parenthesis.pl
acb
$VAR1 = [
  [ 'a', 'c' ], 'b'
];
pl@nereida:~/LEyapp/examples$ use_parenthesis.pl
aacbb
$VAR1 = [
  [
    'a',
    [ [ 'a', 'c' ], 'b' ]
  ],
  'b'
];
```

when working under a tree directive or when the attribute `buildingtree` is set via the `YYBuildingtree` method the semantic action returns a node with children the attributes of the symbols between parenthesis. As usual attributes which aren't references will be skipped from the list of children. See an example:

```
pl@nereida:~/LEyapp/examples$ head -23 List2.yp | cat -n
 1  %{
 2  use Data::Dumper;
 3  %}
 4  %semantic token 'a' 'b' 'c'
 5  %tree
 6  %%
 7  S:
 8        (%name AS 'a' S )'b'
 9          {
10            print "S -> ('a' S )'b'\n";
11            print "Attribute of the first symbol:\n".Dumper($_[1]);
12            print "Attribute of the second symbol: $_[2]\n";
13            $_[0]->YYBuildAST(@_[1..$#_]);
14          }
15      | 'c'
16          {
17            print "S -> 'c'\n";
18            my $r = Parse::Eyapp::Node->new(qw(C(TERMINAL)), sub { $_[1]->attr('c') }) ;
19            print Dumper($r);
20            $r;
21          }
22  ;
23  %%
```

The example shows (line 8) how to rename a `_PAREN` node. The `%name CLASSNAME` goes after the opening parenthesis.

The call to `YYBuildAST` at line 13 with argumetns the attributes of the symbols on the right hand side returns the node describing the current production rule. Notice that line 13 can be rewritten as:

```
goto &Parse::Eyapp::Driver::YYBuildAST;
```

At line 18 the node for the rule is explicitly created using `Parse::Eyapp::Node->new`. The handler passed as second argument is responsible for setting the value of the atribute `attr` of the just created `TERMINAL` node.

Let us see an execution:

```
pl@nereida:~/LEyapp/examples$ use_list2.pl
aacbb
S -> 'c'
```

```
  $VAR1 = bless( {
    'children' => [
      bless( {
        'children' => [],
        'attr' => 'c'
      }, 'TERMINAL' )
    ]
  }, 'C' );
```

the first reduction occurs by the non recursive rule. The execution shows the tree built by the call to `Parse::Eyapp::Node-new>` at line 18.

The execution continues with the reduction or reverse derivation by the rule `S -> ('a' S )'b'`. The action at lines 9-14 dumps the attribute associated with `('a' S)` - or, in other words, the attribute associated with the variable `PAREN-1`. It also dumps the attribute of `'b'`:

```
  S -> ('a' S )'b'
  Attribute of the first symbol:
  $VAR1 = bless( {
      'children' => [
        bless( { 'children' => [], 'attr' => 'a', 'token' => 'a' }, 'TERMINAL' ),
        bless( { 'children' => [ bless( { 'children' => [], 'attr' => 'c' }, 'TERMINAL' )
       ]
     }, 'C' )
    ]
  }, 'AS' );
Attribute of the second symbol: b
```

The last reduction shown is by the rule: `S -> ('a' S )'b'`:

```
  S -> ('a' S )'b'
  Attribute of the first symbol:
  $VAR1 = bless( {
    'children' => [
      bless( { 'children' => [], 'attr' => 'a', 'token' => 'a' }, 'TERMINAL' ),
      bless( {
        'children' => [
          bless( {
            'children' => [
              bless( { 'children' => [], 'attr' => 'a', 'token' => 'a' }, 'TERMINAL' ),
              bless( {
                'children' => [
                  bless( { 'children' => [], 'attr' => 'c' }, 'TERMINAL' )
                ]
              }, 'C' )
            ]
          }, 'AS' ),
          bless( { 'children' => [], 'attr' => 'b', 'token' => 'b' }, 'TERMINAL' )
        ]
      }, 'S_2' )
    ]
  }, 'AS' );
  Attribute of the second symbol: b
```

Actions Inside Parenthesis

Though is a practice to avoid, since it clutters the code, it is certainly permitted to introduce actions between the parenthesis, as in the example below:

```
  pl@nereida:~/LEyapp/examples$ head -16 ListAndAction.eyp | cat -n
   1  # ListAndAction.eyp
   2  %{
   3  my $num = 0;
```

```
 4  %}
 5
 6  %%
 7  S:      'c'
 8              {
 9                print "S -> c\n"
10              }
11      |    ('a' {$num++; print "Seen <$num> 'a's\n"; $_[1] }) S 'b'
12              {
13                print "S -> (a ) S b\n"
14              }
15  ;
16  %%
```

This is the output when executing this program with input `aaacbbb`:

```
pl@nereida:~/LEyapp/examples$ use_listandaction.pl
aaacbbb
Seen <1> 'a's
Seen <2> 'a's
Seen <3> 'a's
S -> c
S -> (a ) S b
S -> (a ) S b
S -> (a ) S b
```

10 NAMES FOR ATTRIBUTES

Attributes can be referenced by meaningful names using the *dot notation* instead of using the classic error-prone positional approach:

```
rhs:  rhseltwithid *
rhseltwithid :
      rhselt '.' IDENT
    | '$' rhselt
    | rhselt
```

for example:

```
exp : exp.left '-' exp.right  { $left - $right }
```

By qualifying the first appearance of the syntactic variable `exp` with the notation `exp.left` we can later refer inside the actions to the associated attribute using the lexical variable `$left`.

The *dollar notation* `$A` can be used as an abbreviation of `A.A`.

11 DEFAULT ACTIONS

When no action is specified both `yapp` and `eyapp` implicitly insert the semantic action `{ $_[1] }`. In `Parse::Eyapp` you can modify such behavior using the `%defaultaction { Perl code }` directive. The `{ Perl code }` clause that follows the `%defaultaction` directive is executed when reducing by any production for which no explicit action was specified.

An Example of Default Action: Translator from Infix to Postfix

See an example that translates an infix expression like `a=b*-3` into a postfix expression like `a b 3 NEG * = :`

```
# File Postfix.eyp (See the examples/ directory)
%right  '='
%left   '-' '+'
%left   '*' '/'
%left   NEG
```

```
%defaultaction { return  "$left $right $op"; }

%%
line: $exp  { print "$exp\n" }
;

exp:        $NUM  { $NUM }
        |   $VAR  { $VAR }
        |   VAR.left '='.op exp.right
        |   exp.left '+'.op exp.right
        |   exp.left '-'.op exp.right
        |   exp.left '*'.op exp.right
        |   exp.left '/'.op exp.right
        |   '-' $exp %prec NEG { "$exp NEG" }
        |   '(' $exp ')' { $exp }
;

%%

# Support subroutines as in the Synopsis example
...
```

The file containing the Eyapp program must be compiled with eyapp:

```
nereida:~/src/perl/YappWithDefaultAction/examples> eyapp Postfix.eyp
```

Next, you have to write a client program:

```
nereida:~/src/perl/YappWithDefaultAction/examples> cat -n usepostfix.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3  use Postfix;
     4
     5  my $parser = new Postfix();
     6  $parser->Run;
```

Now we can run the client program:

```
nereida:~/src/perl/YappWithDefaultAction/examples> usepostfix.pl
Write an expression: -(2*a-b*-3)
2 a * b 3 NEG * - NEG
```

Default Actions, %name and YYName

In eyapp each production rule has a name. The name of a rule can be explicitly given by the programmer using the %name directive. For example, in the piece of code that follows the name ASSIGN is given to the rule exp: VAR '=' exp.

When no explicit name is given the rule has an implicit name. The implicit name of a rule is shaped by concatenating the name of the syntactic variable on its left, an underscore and the ordinal number of the production rule Lhs_# as it appears in the .output file. Avoid giving names matching such pattern to production rules. The patterns /${lhs}_\d+$/ where ${lhs} is the name of the syntactic variable are reserved for internal use by eyapp.

```
pl@nereida:~/LEyapp/examples$ cat -n Lhs.eyp
 1  # Lhs.eyp
 2
 3  %right  '='
 4  %left   '-' '+'
 5  %left   '*' '/'
 6  %left   NEG
 7
 8  %defaultaction {
 9    my $self = shift;
```

```
10    my $name = $self->YYName();
11    bless { children => [ grep {ref($_)} @_] }, $name;
12  }
13
14  %%
15  input:
16              /* empty */
17                { [] }
18          |   input line
19                {
20                  push @{$_[1]}, $_[2] if defined($_[2]);
21                  $_[1]
22                }
23  ;
24
25  line:     '\n'       { }
26          | exp '\n'   {  $_[1] }
27  ;
28
29  exp:
30              NUM   { $_[1] }
31          |   VAR   { $_[1] }
32          |   %name ASSIGN
33              VAR '=' exp
34          |   %name PLUS
35              exp '+' exp
36          |   %name MINUS
37              exp '-' exp
38          |   %name TIMES
39              exp '*' exp
40          |   %name DIV
41              exp '/' exp
42          |   %name UMINUS
43              '-' exp %prec NEG
44          |  '(' exp ')'  { $_[2] }
45  ;
```

Inside a semantic action the name of the current rule can be recovered using the method `YYName` of the parser object.

The default action (lines 8-12) computes as attribute of the left hand side a reference to an object blessed in the name of the rule. The object has an attribute `children` which is a reference to the list of children of the node. The call to `grep`

```
11    bless { children => [ grep {ref($_)} @_] }, $name;
```

excludes children that aren't references. Notice that the lexical analyzer only returns references for the `NUM` and `VAR` terminals:

```
59  sub _Lexer {
60      my($parser)=shift;
61
62      for ($parser->YYData->{INPUT}) {
63          s/^[ \t]+//;
64          return('',undef) unless $_;
65          s/^([0-9]+(?:\.[0-9]+)?)//
66                  and return('NUM', bless { attr => $1}, 'NUM');
67          s/^([A-Za-z][A-Za-z0-9_]*)//
68                  and return('VAR',bless {attr => $1}, 'VAR');
69          s/^(.)//s
70                  and return($1, $1);
71      }
72      return('',undef);
```

```
73  }
```

follows the client program:

```
pl@nereida:~/LEyapp/examples$ cat -n uselhs.pl
     1  #!/usr/bin/perl -w
     2  use Lhs;
     3  use Data::Dumper;
     4
     5  $parser = new Lhs();
     6  my $tree = $parser->Run;
     7  $Data::Dumper::Indent = 1;
     8  if (defined($tree)) { print Dumper($tree); }
     9  else { print "Cadena no válida\n"; }
```

When executed with input `a=(2+3)*b` the parser produces the following tree:

```
ASSIGN(TIMES(PLUS(NUM[2],NUM[3]), VAR[b]))
```

See the result of an execution:

```
pl@nereida:~/LEyapp/examples$ uselhs.pl
a=(2+3)*b
$VAR1 = [
  bless( {
    'children' => [
      bless( { 'attr' => 'a' }, 'VAR' ),
      bless( {
        'children' => [
          bless( {
            'children' => [
              bless( { 'attr' => '2' }, 'NUM' ),
              bless( { 'attr' => '3' }, 'NUM' )
            ]
          }, 'PLUS' ),
          bless( { 'attr' => 'b' }, 'VAR' )
        ]
      }, 'TIMES' )
    ]
  }, 'ASSIGN' )
];
```

The name of a production rule can be changed at execution time. See the following example:

```
$ sed -n '29,50p' YYNameDynamic.eyp | cat -n
   1  exp:
   2              NUM   { $_[1] }
   3          |   VAR   { $_[1] }
   4          |   %name ASSIGN
   5              VAR '=' exp
   6          |   %name PLUS
   7              exp '+' exp
   8          |   %name MINUS
   9              exp '-' exp
  10                {
  11                  my $self = shift;
  12                  $self->YYName('SUBTRACT'); # rename it
  13                  $self->YYBuildAST(@_); # build the node
  14                }
  15          |   %name TIMES
  16              exp '*' exp
  17          |   %name DIV
```

```
 18              exp '/' exp
 19          |   %name UMINUS
 20              '-' exp %prec NEG
 21          |  '(' exp ')'  { $_[2] }
 22  ;
```

When the client program is executed we can see the presence of the `SUBTRACT` nodes:

```
pl@nereida:~/LEyapp/examples$ useyynamedynamic.pl
2-b
$VAR1 = [
  bless( {
    'children' => [
      bless( {
        'attr' => '2'
      }, 'NUM' ),
      bless( {
        'attr' => 'b'
      }, 'VAR' )
    ]
  }, 'SUBTRACT' )
];
```

12 GRAMMAR REUSE

Reusing Grammars Using Inheritance

An method to reuse a grammar is via inheritance. The client inherits the generated parser module and expands it with methods that inherit or overwrite the actions. Here is an example. Initially we have this Eyapp grammar:

```
pl@europa:~/LEyapp/examples/recycle$ cat -n NoacInh.eyp
   1  %left   '+'
   2  %left   '*'
   3
   4  %defaultaction {
   5    my $self = shift;
   6
   7    my $action = $self->YYName;
   8
   9    $self->$action(@_);
  10  }
  11
  12  %%
  13  exp:        %name NUM
  14                NUM
  15          |   %name PLUS
  16                exp '+' exp
  17          |   %name TIMES
  18                exp '*' exp
  19          |   '(' exp ')'
  20                { $_[2] }
  21  ;
  22
  23  %%
  24
  25  sub _Error {
  26    my($token)=$_[0]->YYCurval;
  27    my($what)= $token ? "input: '$token'" : "end of input";
  28    my @expected = $_[0]->YYExpect();
  29
  30    local $" = ', ';
```

```
31    die "Syntax error near $what. Expected one of these tokens: @expected\n";
32  }
33
34
35  my $x = '';
36
37  sub _Lexer {
38    my($parser)=shift;
39
40    for ($x) {
41      s/^\s+//;
42      $_ eq '' and return('',undef);
43
44      s/^([0-9]+(?:\.[0-9]+)?)//   and return('NUM',$1);
45      s/^([A-Za-z][A-Za-z0-9_]*)// and return('VAR',$1);
46      s/^(.)//s                    and return($1,$1);
47    }
48  }
49
50  sub Run {
51    my($self)=shift;
52    $x = shift;
53    my $debug = shift;
54
55    $self->YYParse(
56      yylex => \&_Lexer,
57      yyerror => \&_Error,
58      yydebug => $debug,
59    );
60  }
```

The following program defines two classes: `CalcActions` that implements the actions for the calculator and package `PostActions` that implements the actions for the infix to postfix translation. This way we have an example that reuses the former grammar twice:

```
pl@europa:~/LEyapp/examples/recycle$ cat -n icalcu_and_ipost.pl
   1  #!/usr/bin/perl -w
   2  package CalcActions;
   3  use strict;
   4  use base qw{NoacInh};
   5
   6  sub NUM {
   7    return $_[1];
   8  }
   9
  10  sub PLUS {
  11    $_[1]+$_[3];
  12  }
  13
  14  sub TIMES {
  15    $_[1]*$_[3];
  16  }
  17
  18  package PostActions;
  19  use strict;
  20  use base qw{NoacInh};
  21
  22  sub NUM {
  23    return $_[1];
  24  }
  25
```

```
 26  sub PLUS {
 27    "$_[1] $_[3] +";
 28  }
 29
 30  sub TIMES {
 31    "$_[1] $_[3] *";
 32  }
 33
 34  package main;
 35  use strict;
 36
 37  my $calcparser = CalcActions->new();
 38  print "Write an expression: ";
 39  my $x = <STDIN>;
 40  my $e = $calcparser->Run($x);
 41
 42  print "$e\n";
 43
 44  my $postparser = PostActions->new();
 45  my $p = $postparser->Run($x);
 46
 47  print "$p\n";
```

The subroutine used as default action in `NoacInh.eyp` is so useful that is packed as the *Parse::Eyapp::Driver* method `YYDelegateaction`.

See files `examples/recycle/NoacYYDelegateaction.eyp` and `examples/recycle/icalcu_and_ipost_yydel.pl` for an example of use of `YYDelegateaction`.

Reusing Grammars by Dynamic Substitution of Semantic Actions

The methods `YYSetaction` and `YYAction` of the parser object provide a way to selectively substitute some actions of a given grammar. Let us consider once more a postfix to infix translator:

```
pl@europa:~/LEyapp/examples/recycle$ cat -n PostfixWithActions.eyp
   1  # File PostfixWithActions.eyp
   2  %right  '='
   3  %left   '-' '+'
   4  %left   '*' '/'
   5  %left   NEG
   6
   7  %%
   8  line: $exp  { print "$exp\n" }
   9  ;
  10
  11  exp:        $NUM
  12                  { $NUM }
  13          |   $VAR
  14                  { $VAR }
  15          |   %name ASSIGN
  16                VAR.left '='exp.right
  17                  { "$_[3] &$_[1] ASSIGN"; }
  18          |   %name PLUS
  19                exp.left '+'exp.right
  20                  { "$_[1] $_[3] PLUS"; }
  21          |   %name MINUS
  22                exp.left '-'exp.right
  23                  { "$_[1] $_[3] MINUS"; }
  24          |   %name TIMES
  25                exp.left '*'exp.right
  26                  { "$_[1] $_[3] TIMES"; }
  27          |   %name DIV
```

```
 28                exp.left '/'exp.right
 29                  { "$_[1] $_[3] DIV"; }
 30          |   %name NEG '-' $exp %prec NEG
 31                  { "$exp NEG" }
 32          |   '(' $exp ')'
 33                  { $exp }
 34  ;
 35
 36  %%
 37
 38  sub _Error {
 39    my($token)=$_[0]->YYCurval;
 40    my($what)= $token ? "input: '$token'" : "end of input";
 41    my @expected = $_[0]->YYExpect();
 42
 43    local $" = ', ';
 44    die "Syntax error near $what. Expected one of these tokens: @expected\n";
 45  }
 46
 47  my $x;
 48
 49  sub _Lexer {
 50    my($parser)=shift;
 51
 52    for ($x) {
 53      s/^\s+//;
 54      $_ eq '' and return('',undef);
 55
 56      s/^([0-9]+(?:\.[0-9]+)?)//   and return('NUM',$1);
 57      s/^([A-Za-z][A-Za-z0-9_]*)// and return('VAR',$1);
 58      s/^(.)//s                    and return($1,$1);
 59    }
 60  }
 61
 62  sub Run {
 63    my($self)=shift;
 64    $x = shift;
 65    $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
 66      #yydebug => 0xFF
 67    );
 68  }
```

The program `rewritepostfixwithactions.pl` uses the former grammar to translate infix expressions to postfix expressions. It also implements a calculator reusing the grammar in `PostfixWithActions.eyp`. It does so using the `YYSetaction` method. The semantic actions for the productions named

- ASSIGN
- PLUS
- TIMES
- DIV
- NEG

are selectively substituted by the appropriate actions, while the other semantic actions remain unchanged:

```
pl@europa:~/LEyapp/examples/recycle$ cat -n rewritepostfixwithactions.pl
   1  #!/usr/bin/perl
   2  use warnings;
   3  use PostfixWithActions;
   4
```

```
 5  my $debug = shift || 0;
 6  my $pparser = PostfixWithActions->new();
 7  print "Write an expression: ";
 8  my $x = <STDIN>;
 9
10  # First, trasnlate to postfix ...
11  $pparser->Run($x, $debug);
12
13  # And then selectively substitute
14  # some semantic actions
15  # to obtain an infix calculator ...
16  my %s;            # symbol table
17  $pparser->YYSetaction(
18    ASSIGN => sub { $s{$_[1]} = $_[3] },
19    PLUS   => sub { $_[1] + $_[3] },
20    TIMES  => sub { $_[1] * $_[3] },
21    DIV    => sub { $_[1] / $_[3] },
22    NEG    => sub { -$_[2] },
23  );
24
25  $pparser->Run($x, $debug);
```

When running this program the output is:

```
examples/recycle$ ./rewritepostfixwithactions.pl
Write an expression: 2*3+4
2 3 TIMES 4 PLUS
10
examples/recycle$ rewritepostfixwithactions.pl
Write an expression: a = 2*(b = 3+5)
2 3 5 PLUS &b ASSIGN TIMES &a ASSIGN
16
```

13 ABSTRACT SYNTAX TREES: %tree AND %name

%tree Default Names

`Parse::Eyapp` facilitates the construction of concrete syntax trees and abstract syntax trees (abbreviated AST from now on) through the `%tree` directive. Actually, the `%tree` directive is equivalent to a call to the `YYBuildAST` method of the parser object.

Any production production rule `A->XYZ` can be named using a directive `%name someclass`.

When reducing by a production rule `A->XYZ` the `%tree` directive (i.e., the `YYBuildAST` method) builds an anonymous hash blessed in `someclass`. The hash has an attribute `children` containing the references to the AST trees associated with the symbols in the right hand side `X`, C>Y>, etc.

If no explicit name was given to the production rule, `YYBuildAST` blesses the node in the class name resulting from the concatenation of the left hand side and the production number. The production number is the ordinal number of the production as they appear in the associated `.output` file (see option `-v` of *eyapp*). For example, given the grammar:

```
pl@europa:~/LEyapp/examples/eyapplanguageref$ sed -ne '8,27p' treewithoutnames.pl
my $grammar = q{
  %right  '='     # Lowest precedence
  %left   '-' '+' # + and - have more precedence than = Disambiguate a-b-c as (a-b)-c
  %left   '*' '/' # * and / have more precedence than + Disambiguate a/b/c as (a/b)/c
  %left   NEG     # Disambiguate -a-b as (-a)-b and not as -(a-b)
  %tree           # Let us build an abstract syntax tree ...

  %%
  line: exp <+ ';'>  { $_[1] } /* list of expressions separated by ';' */
  ;
```

```
  exp:
       NUM           |   VAR       | VAR '=' exp
    | exp '+' exp    | exp '-' exp |  exp '*' exp
    | exp '/' exp
    | '-' exp %prec NEG
    |   '(' exp ')'  { $_[2] }
  ;

  %%
```

The tree produced by the parser when feed with input `a=2*b` is:

```
pl@europa:~/LEyapp/examples/eyapplanguageref$ ./treewithoutnames.pl

************
_PLUS_LIST(exp_6(TERMINAL[a],exp_9(exp_4(TERMINAL[2]),exp_5(TERMINAL[b]))))
************
```

If we want to see the correspondence between names and rules we can generate and check the corresponding file `.output` setting the `outputfile` of `Parse::Eyapp`:

```
Parse::Eyapp->new_grammar( # Create the parser package/class
  input=>$grammar,
  classname=>'Calc', # The name of the package containing the parser
  firstline=>9,      # String $grammar starts at line 9 (for error diagnostics)
  outputfile=>'treewithoutnames'
);
```

The grammar with the expanded rules appears in the `.output` file:

```
lusasoft@LusaSoft:~/src/perl/Eyapp/examples/eyapplanguageref$ sed -ne '28,42p' treewithoutnames.outpu
Rules:
------
0:      $start -> line $end
1:      PLUS-1 -> PLUS-1 ';' exp
2:      PLUS-1 -> exp
3:      line -> PLUS-1
4:      exp -> NUM
5:      exp -> VAR
6:      exp -> VAR '=' exp
7:      exp -> exp '+' exp
8:      exp -> exp '-' exp
9:      exp -> exp '*' exp
10:     exp -> exp '/' exp
11:     exp -> '-' exp
12:     exp -> '(' exp ')'
```

We can see now that the node `exp_9` corresponds to the production `exp -> exp '*' exp`. Observe also that the Eyapp production:

```
                              line: exp <+ ';'>
actually produces the productions:

                        1:      PLUS-1 -> PLUS-1 ';' exp
                        2:      PLUS-1 -> exp
```

and that the name of the class associated with the non empty list is `_PLUS_LIST`.

%tree Giving Explicit Names

A production rule can be *named* using the `%name IDENTIFIER` directive. For each production rule a namespace/package is created. *The* `IDENTIFIER` *is the name of the associated package.* Therefore, by modifying the former grammar with additional `%name` directives:

```
lusasoft@LusaSoft:~/src/perl/Eyapp/examples/eyapplanguageref$ sed -ne '8,26p' treewithnames.pl
my $grammar = q{
  %right  '='     # Lowest precedence
  %left   '-' '+' # + and - have more precedence than = Disambiguate a-b-c as (a-b)-c
  %left   '*' '/' # * and / have more precedence than + Disambiguate a/b/c as (a/b)/c
  %left   NEG     # Disambiguate -a-b as (-a)-b and not as -(a-b)
  %tree           # Let us build an abstract syntax tree ...

  %%
  line: exp <%name EXPS + ';'>  { $_[1] } /* list of expressions separated by ';' */
  ;

  exp:
      %name NUM    NUM           | %name VAR   VAR         | %name ASSIGN VAR '=' exp
    | %name PLUS   exp '+' exp   | %name MINUS exp '-' exp | %name TIMES  exp '*' exp
    | %name DIV    exp '/' exp
    | %name UMINUS '-' exp %prec NEG
    |   '(' exp ')'  { $_[2] }
  ;
```

we are explicitly naming the productions. Thus, all the node instances corresponding to the production `exp:  VAR '=' exp` will belong to the class `ASSIGN`. Now the tree for `a=2*b` becomes:

```
lusasoft@LusaSoft:~/src/perl/Eyapp/examples/eyapplanguageref$ ./treewithnames.pl

************
EXPS(ASSIGN(TERMINAL[a],TIMES(NUM(TERMINAL[2]),VAR(TERMINAL[b]))))
************
```

Observe how the list has been named `EXPS`. The `%name` directive prefixes the list operator (`[+*?]`).

TERMINAL Nodes

Nodes named `TERMINAL` are built from the tokens provided by the lexical analyzer. `Parse::Eyapp` follows the same protocol than *Parse::Yapp* for communication between the parser and the lexical analyzer: A couple `($token, $attribute)` is returned by the lexical analyzer. These values are stored under the keys `token` and `attr`. `TERMINAL` nodes as all `Parse::Eyapp::Node` nodes also have the attribute `children` but is - almost always - empty.

Explicit Actions Inside %tree

Explicit actions can be specified by the programmer like in this line from the *Parse::Eyapp* SYNOPSIS example:

```
    |   '(' exp ')'  { $_[2] }  /* Let us simplify a bit the tree */
```

Explicit actions receive as arguments the references to the children nodes already built. The programmer can influence the shape of the tree by inserting these explicit actions. In this example the programmer has decided to simplify the syntax tree: the nodes associated with the parenthesis are discarded and the reference to the subtree containing the proper expression is returned. Such manoeuvre is called *bypassing*. See section The bypass clause and the %no bypass directive to know more about *automatic bypassing*

Explicitly Building Nodes With YYBuildAST

Sometimes the best time to decorate a node with some attributes is just after being built. In such cases the programmer can take *manual control* building the node with `YYBuildAST` to inmediately proceed to decorate it.

The following example illustrates the situation (see file `lib/Simple/Types.eyp` inside `examples/typechecking/Simple-T`

```
$ sed -n '397,408p' lib/Simple/Types.eyp
Variable:
    %name VAR
    ID
  | %name  VARARRAY
    $ID ('[' binary ']') <%name INDEXSPEC +>
      {
        my $self = shift;
        my $node =  $self->YYBuildAST(@_);
        $node->{line} = $ID->[1];# $_[1]->[1]
        return $node;
      }
;
```

This production rule defines the expression to access an array element as an identifier followed by a non empty list of binary expressions `Variable: ID ('[' binary ']')+`. Furthermore, the node corresponding to the list of indices has been named `INDEXSPEC`.

When no explicit action is inserted a binary node will be built having as first child the node corresponding to the identifier `$ID` and as second child the reference to the list of binary expressions. The children corresponding to `'['` and `']'` are discarded since they are -by default- *syntactic tokens* (see section Syntactic and Semantic tokens). However, the programmer wants to decorate the node being built with a `line` attribute holding the line number in the source code where the identifier being used appears. The call to the `Parse::Eyapp::Driver` method `YYBuildAST` does the job of building the node. After that the node can be decorated and returned.

Actually, the `%tree` directive is semantically equivalent to:

```
%default action { goto &Parse::Eyapp::Driver::YYBuildAST }
```

Returning non References Under `%tree`

When a *explicit user action returns s.t. that is not a reference no node will be inserted*. This fact can be used to suppress nodes in the AST being built. See the following example (file `examples/returnnonode.yp`):

```
$ sed -ne '1,17p' returnnonode.yp | cat -n
   1  %tree
   2  %semantic token 'a' 'b'
   3  %%
   4  S:    %name EMPTY
   5          /* empty */
   6      | %name AES
   7          S A
   8      | %name BES
   9          S B
  10  ;
  11  A : %name A
  12        'a'
  13  ;
  14  B : %name B
  15        'b' { }
  16  ;
  17  %%
```

since the action at line 15 returns `undef` the `B : 'b'` subtree will not be inserted in the AST:

```
$ usereturnnonode.pl
ababa
AES(BES(AES(BES(AES(EMPTY,A(TERMINAL[a]))),A(TERMINAL[a]))),A(TERMINAL[a]))
```

Observe the absence of `B`s and `'b'`s.

Intermediate actions and %tree

Intermediate actions can be used to change the shape of the AST (prune it, decorate it, etc.) but the value returned by them is ignored. The grammar below has two intermediate actions. They modify the attributes of the node to its left and return a reference `$f` to such node (lines 5 and 6):

```
$ sed -ne '1,15p' intermediateactiontree.yp | cat -n
   1  %semantic token 'a' 'b'
   2  %tree bypass
   3  %%
   4  S:    %name EMPTY
   5         /* empty */
   6      | %name SA
   7         S A.f { $f->{attr} = "A"; $f; } A
   8      | %name SB
   9         S B.f { $f->{attr} = "B"; $f; } B
  10  ;
  11  A : %name A 'a'
  12  ;
  13  B : %name B 'b'
  14  ;
  15  %%
```

See the client program:

```
nereida:~/src/perl/YappWithDefaultAction/examples> cat -n useintermediateactiontree.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use Parse::Eyapp;
 4  use intermediateactiontree;
 5
 6  { no warnings;
 7  *A::info = *B::info = sub { $_[0]{attr} };
 8  }
 9
10  my $parser = intermediateactiontree->new();
11  my $t = $parser->Run;
12  print $t->str,"\n";
```

When it runs produces this output:

```
$ useintermediateactiontree.pl
aabbaa
SA(SB(SA(EMPTY,A[A],A[a]),B[B],B[b]),A[A],A[a])
```

The attributes of left `A`s have been effectively changed by the intermediate actions from `'a'` to `'A'`. However no further children have been inserted.

Syntactic and Semantic tokens

`Parse::Eyapp` differences between `syntactic tokens` and `semantic tokens`. By default all tokens declared using string notation (i.e. between quotes like `'+'`, `'='`) are considered *syntactic tokens*. Tokens declared by an identifier (like `NUM` or `VAR`) are by default considered *semantic tokens*. **Syntactic tokens do not yield to nodes in the syntactic tree**. Thus, the first print in the section *Parse::Eyapp* SYNOPSIS example:

```
$ cat -n synopsis.pl
   1  #!/usr/bin/perl -w
   2  use strict;
   3  use Parse::Eyapp;
   4  use Parse::Eyapp::Treeregexp;
   5
   6  sub TERMINAL::info {
   7    $_[0]{attr}
```

```
 8 }
 9
10 my $grammar = q{
11   %right  '='     # Lowest precedence
12   %left   '-' '+' # + and - have more precedence than = Disambiguate a-b-c as (a-b)-c
13   %left   '*' '/' # * and / have more precedence than + Disambiguate a/b/c as (a/b)/c
14   %left   NEG     # Disambiguate -a-b as (-a)-b and not as -(a-b)
15   %tree           # Let us build an abstract syntax tree ...
16
17   %%
18   line:
19       exp <%name EXPRESSION_LIST + ';'>
20         { $_[1] } /* list of expressions separated by ';' */
21   ;
22
23   /* The %name directive defines the name of the class */
24   exp:
25       %name NUM
26       NUM
27     | %name VAR
28       VAR
29     | %name ASSIGN
30       VAR '=' exp
31     | %name PLUS
32       exp '+' exp
33     | %name MINUS
34       exp '-' exp
35     | %name TIMES
36       exp '*' exp
37     | %name DIV
38       exp '/' exp
39     | %name UMINUS
40       '-' exp %prec NEG
41     | '(' exp ')'
42         { $_[2] }  /* Let us simplify a bit the tree */
43   ;
44
45   %%
46   sub _Error { die "Syntax error near ".($_[0]->YYCurval?$_[0]->YYCurval:"end of file")."\n" }
47
48   sub _Lexer {
49     my($parser)=shift; # The parser object
50
51     for ($parser->YYData->{INPUT}) { # Topicalize
52       m{\G\s+}gc;
53       $_ eq '' and return('',undef);
54       m{\G([0-9]+(?:\.[0-9]+)?)}gc and return('NUM',$1);
55       m{\G([A-Za-z][A-Za-z0-9_]*)}gc and return('VAR',$1);
56       m{\G(.)}gcs and return($1,$1);
57     }
58     return('',undef);
59   }
60
61   sub Run {
62       my($self)=shift;
63       $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error, );
64   }
65 }; # end grammar
66
67 our (@all, $uminus);
```

```
 68
 69  Parse::Eyapp->new_grammar( # Create the parser package/class
 70    input=>$grammar,
 71    classname=>'Calc', # The name of the package containing the parser
 72    firstline=>7       # String $grammar starts at line 7 (for error diagnostics)
 73  );
 74  my $parser = Calc->new();                # Create a parser
 75  $parser->YYData->{INPUT} = "2*-3+b*0;--2\n"; # Set the input
 76  my $t = $parser->Run;                    # Parse it!
 77  local $Parse::Eyapp::Node::INDENT=2;
 78  print "Syntax Tree:",$t->str;
 79
 80  # Let us transform the tree. Define the tree-regular expressions ..
 81  my $p = Parse::Eyapp::Treeregexp->new( STRING => q{
 82      { #  Example of support code
 83        my %Op = (PLUS=>'+', MINUS => '-', TIMES=>'*', DIV => '/');
 84      }
 85      constantfold: /TIMES|PLUS|DIV|MINUS/:bin(NUM($x), NUM($y))
 86        => {
 87          my $op = $Op{ref($bin)};
 88          $x->{attr} = eval  "$x->{attr} $op $y->{attr}";
 89          $_[0] = $NUM[0];
 90        }
 91      uminus: UMINUS(NUM($x)) => { $x->{attr} = -$x->{attr}; $_[0] = $NUM }
 92      zero_times_whatever: TIMES(NUM($x), .) and { $x->{attr} == 0 } => { $_[0] = $NUM }
 93      whatever_times_zero: TIMES(., NUM($x)) and { $x->{attr} == 0 } => { $_[0] = $NUM }
 94    },
 95    OUTPUTFILE=> 'main.pm'
 96  );
 97  $p->generate(); # Create the tranformations
 98
 99  $t->s($uminus); # Transform UMINUS nodes
100  $t->s(@all);    # constant folding and mult. by zero
101
102  local $Parse::Eyapp::Node::INDENT=0;
103  print "\nSyntax Tree after transformations:\n",$t->str,"\n";
```

gives as result the following output:

```
nereida:~/src/perl/YappWithDefaultAction/examples> synopsis.pl
Syntax Tree:
EXPRESSION_LIST(
  PLUS(
    TIMES(
      NUM(
        TERMINAL[2]
      ),
      UMINUS(
        NUM(
          TERMINAL[3]
        )
      ) # UMINUS
    ) # TIMES,
    TIMES(
      VAR(
        TERMINAL[b]
      ),
      NUM(
        TERMINAL[0]
      )
    ) # TIMES
```

```
  ) # PLUS,
  UMINUS(
    UMINUS(
      NUM(
        TERMINAL[2]
      )
    ) # UMINUS
  ) # UMINUS
) # EXPRESSION_LIST
```

`TERMINAL` nodes corresponding to tokens that were defined by strings like `'='`, `'-'`, `'+'`, `'/'`, `'*'`, `'('` and `')'` do not appear in the tree. `TERMINAL` nodes corresponding to tokens that were defined using an identifier, like `NUM` or `VAR` are, by default, *semantic tokens* and appear in the AST.

Changing the Status of a Token

The new token declaration directives `%syntactic token` and `%semantic token` can change the status of a token. For example (file `15treewithsyntactictoken.pl` in the `examples/` directory), given the grammar:

```
%syntactic token b
%semantic token 'a' 'c'
%tree

%%

S: %name ABC
     A B C
 | %name BC
     B C
;

A: %name A
     'a'
;

B: %name B
     b
;

C: %name C
    'c'
;
%%
```

the tree build for input `abc` will be `ABC(A(TERMINAL[a]),B,C(TERMINAL[c]))`.

Saving the Information of Syntactic Tokens in their Father

The reason for the adjective `%syntactic` applied to a token is to state that the token influences the shape of the syntax tree but carries no other information. When the syntax tree is built the node corresponding to the token is discarded.

Sometimes the difference between syntactic and semantic tokens is blurred. For example the line number associated with an instance of the syntactic token `'+'` can be used later -say during type checking- to emit a more accurate error diagnostic. But if the node was discarded the information about that line number is no longer available. When building the syntax tree `Parse::Eyapp` (namely the method `Parse::Eyapp::YYBuildAST`) checks if the method `TERMINAL::save_attributes` exists and if so it will be called when dealing with a *syntactic token*. The method receives as argument - additionally to the reference to the attribute of the token as it is returned by the lexical analyzer - a reference to the node associated with the left hand side of the production. Here is an example (file `lib/Simple/Types.eyp` in `examples/typechecking/Simple-Types-XXX.tar.gz`) of use:

```
sub TERMINAL::save_attributes {
  # $_[0] is a syntactic terminal
  # $_[1] is the father.
  push @{$_[1]->{lines}}, $_[0]->[1]; # save the line number
}
```

The `bypass` clause and the `%no bypass` directive

The shape of the tree can be also modified using some `%tree` clauses as `%tree bypass` which will produce an automatic *bypass* of any node with only one child at tree-construction-time.

A *bypass operation* consists in *returning the only child of the node being visited to the father of the node and re-typing (re-blessing) the node in the name of the production* (if a name was provided).

A node may have only one child at tree-construction-time for one of two reasons.

- The first occurs when the right hand side of the production was already unary like in:

```
exp:
    %name NUM  NUM
```

  Here - if the `bypass` clause is used - the `NUM` node will be bypassed and the child `TERMINAL` built from the information provided by the lexical analyzer will be renamed/reblessed as `NUM`.

- Another reason for a node to be *bypassed* is the fact that though the right hand side of the production may have more than one symbol, only one of them is not a syntactic token like in:

```
exp: '(' exp ')'
```

A consequence of the global scope application of `%tree bypass` is that undesired bypasses may occur like in

```
exp : %name UMINUS
      '-' $exp %prec NEG
```

though the right hand side has two symbols, token `'-'` is a syntactic token and therefore only `exp` is left. The *bypass* operation will be applied when building this node. This *bypass* can be avoided applying the `no bypass ID` directive to the corresponding production:

```
exp : %no bypass UMINUS
      '-' $exp %prec NEG
```

The following example (file `examples/bypass.pl`) is the equivalent of the Parse::Eyapp SYNOPSIS example but using the `bypass` clause instead:

```
use Parse::Eyapp;
use Parse::Eyapp::Treeregexp;

sub TERMINAL::info { $_[0]{attr} }
{ no warnings; *VAR::info = *NUM::info = \&TERMINAL::info; }

my $grammar = q{
  %right  '='     # Lowest precedence
  %left   '-' '+'
  %left   '*' '/'
  %left   NEG     # Disambiguate -a-b as (-a)-b and not as -(a-b)
  %tree bypass    # Let us build an abstract syntax tree ...

  %%
  line: exp <%name EXPRESSION_LIST + ';'>  { $_[1] }
  ;

  exp:
      %name NUM  NUM            | %name VAR   VAR         | %name ASSIGN VAR '=' exp
    | %name PLUS exp '+' exp    | %name MINUS exp '-' exp | %name TIMES  exp '*' exp
    | %name DIV     exp '/' exp
    | %no bypass UMINUS
      '-' $exp %prec NEG
    |   '(' exp ')'
  ;
```

```
  %%
  # sub _Error, _Lexer and Run like in the synopsis example
  # ...
}; # end grammar

our (@all, $uminus);

Parse::Eyapp->new_grammar( # Create the parser package/class
  input=>$grammar,
  classname=>'Calc', # The name of the package containing the parser
  firstline=>7       # String $grammar starts at line 7 (for error diagnostics)
);
my $parser = Calc->new();                # Create a parser
$parser->YYData->{INPUT} = "a=2*-3+b*0\n"; # Set the input
my $t = $parser->Run;                    # Parse it!

print "\n************\n".$t->str."\n************\n";

# Let us transform the tree. Define the tree-regular expressions ..
my $p = Parse::Eyapp::Treeregexp->new( STRING => q{
  { #  Example of support code
    my %Op = (PLUS=>'+', MINUS => '-', TIMES=>'*', DIV => '/');
  }
  constantfold: /TIMES|PLUS|DIV|MINUS/:bin(NUM, NUM)
    => {
      my $op = $Op{ref($_[0])};
      $NUM[0]->{attr} = eval  "$NUM[0]->{attr} $op $NUM[1]->{attr}";
      $_[0] = $NUM[0];
    }
  zero_times_whatever: TIMES(NUM, .) and { $NUM->{attr} == 0 } => { $_[0] = $NUM }
  whatever_times_zero: TIMES(., NUM) and { $NUM->{attr} == 0 } => { $_[0] = $NUM }
  uminus: UMINUS(NUM) => { $NUM->{attr} = -$NUM->{attr}; $_[0] = $NUM }
  },
  OUTPUTFILE=> 'main.pm'
);
$p->generate(); # Create the tranformations

$t->s(@all);    # constant folding and mult. by zero

print $t->str,"\n";
```

when running this example with input `"a=2*-3+b*0\n"` we obtain the following output:

```
nereida:~/src/perl/YappWithDefaultAction/examples> bypass.pl

************
EXPRESSION_LIST(ASSIGN(TERMINAL[a],PLUS(TIMES(NUM[2],UMINUS(NUM[3])),TIMES(VAR[b],NUM[0]))))
************
EXPRESSION_LIST(ASSIGN(TERMINAL[a],NUM[-6]))
```

As you can see the trees are more compact when using the `bypass` directive.

The `alias` clause of the `%tree` directive

Access to children in *Parse::Eyapp* is made through the `child` and `children` methods. There are occasions however where access by name to the children may be preferable. The use of the `alias` clause with the `%tree` directive creates accessors to the children with names specified by the programmer. The *dot and dollar notations* are used for this. When dealing with a production like:

```
A:
   %name A_Node
   Node B.bum N.pum $Chip
```

methods `bum`, `pum` and `Chip` will be created for the class `A_Node`. Those methods will provide access to the respective child (first, second and third in the example). The methods are build at compile-time and therefore later transformations of the AST modifying the order of the children may invalidate the use of these getter-setters.

The `%prefix` directive used in line 7 of the following example is equivalent to the use of the `yyprefix`. The node classes are prefixed with the specified prefix: `R::S::` in this example.

```
cat -n alias_and_yyprefix.pl
    1  #!/usr/local/bin/perl
    2  use warnings;
    3  use strict;
    4  use Parse::Eyapp;
    5
    6  my $grammar = q{
    7    %prefix R::S::
    8
    9    %right  '='
   10    %left   '-' '+'
   11    %left   '*' '/'
   12    %left   NEG
   13    %tree bypass alias
   14
   15    %%
   16    line: $exp  { $_[1] }
   17    ;
   18
   19    exp:
   20        %name NUM
   21              $NUM
   22      | %name VAR
   23              $VAR
   24      | %name ASSIGN
   25              $VAR '=' $exp
   26      | %name PLUS
   27              exp.left '+' exp.right
   28      | %name MINUS
   29              exp.left '-' exp.right
   30      | %name TIMES
   31              exp.left '*' exp.right
   32      | %name DIV
   33              exp.left '/' exp.right
   34      | %no bypass UMINUS
   35              '-' $exp %prec NEG
   36      |   '(' exp ')'  { $_[2] } /* Let us simplify a bit the tree */
   37    ;
   38
   39    %%
   ..    ....
   76  }; # end grammar
   77
   78
   79  Parse::Eyapp->new_grammar(
   80    input=>$grammar,
   81    classname=>'Alias',
   82    firstline =>7,
   83    outputfile => 'main',
   84  );
   85  my $parser = Alias->new();
   86  $parser->YYData->{INPUT} = "a = -(2*3+5-1)\n";
   87  my $t = $parser->Run;
   88  $Parse::Eyapp::Node::INDENT=0;
```

```
 89  print $t->VAR->str."\n";             # a
 90  print "***************\n";
 91  print $t->exp->exp->left->str."\n";  # 2*3+5
 92  print "***************\n";
 93  print $t->exp->exp->right->str."\n"; # 1
```

The tree `$t` for the expression `"a = -(2*3+5-1)\n"` is:

```
R::S::ASSIGN(
   R::S::TERMINAL,
   R::S::UMINUS(
     R::S::MINUS(
       R::S::PLUS(R::S::TIMES(R::S::NUM,R::S::NUM),R::S::NUM),
       R::S::NUM
     )
   )
)
```

The `R::S::ASSIGN` class has methods `VAR` (see line 89 above) and `exp` (see lines 91 and 93) to refer to its two children. The result of the execution is:

```
$ alias_and_yyprefix.pl
R::S::TERMINAL
***************
R::S::PLUS(R::S::TIMES(R::S::NUM,R::S::NUM),R::S::NUM)
***************
R::S::NUM
```

As a second example of the use of `%alias`, the CPAN module *Language::AttributeGrammar* provides AST decorators from an attribute grammar specification of the AST. To work *Language::AttributeGrammar* requires named access to the children of the AST nodes. Follows an example (file `examples/CalcwithAttributeGrammar.pl`) of a small calculator:

```
pl@nereida:~/LEyapp/examples$ cat -n CalcwithAttributeGrammar.pl
   1  #!/usr/bin/perl -w
   2  use strict;
   3  use Parse::Eyapp;
   4  use Data::Dumper;
   5  use Language::AttributeGrammar;
   6
   7  my $grammar = q{
   8  %{
   9  # use Data::Dumper;
  10  %}
  11  %right  '='
  12  %left   '-' '+'
  13  %left   '*' '/'
  14  %left   NEG
  15  %tree bypass alias
  16
  17  %%
  18  line: $exp  { $_[1] }
  19  ;
  20
  21  exp:
  22      %name NUM
  23            $NUM
  24          | %name VAR
  25            $VAR
  26          | %name ASSIGN
  27            $VAR '=' $exp
  28          | %name PLUS
```

```
29            exp.left '+' exp.right
30          | %name MINUS
31            exp.left '-' exp.right
32          | %name TIMES
33            exp.left '*' exp.right
34          | %name DIV
35            exp.left '/' exp.right
36          | %no bypass UMINUS
37            '-' $exp %prec NEG
38    |   '(' $exp ')'  { $_[2] } /* Let us simplify a bit the tree */
39  ;
40
41  %%
42
43  sub _Error {
44          exists $_[0]->YYData->{ERRMSG}
45      and do {
46          print $_[0]->YYData->{ERRMSG};
47          delete $_[0]->YYData->{ERRMSG};
48          return;
49      };
50      print "Syntax error.\n";
51  }
52
53  sub _Lexer {
54      my($parser)=shift;
55
56          $parser->YYData->{INPUT}
57      or  $parser->YYData->{INPUT} = <STDIN>
58      or  return('',undef);
59
60      $parser->YYData->{INPUT}=~s/^\s+//;
61
62      for ($parser->YYData->{INPUT}) {
63          s/^([0-9]+(?:\.[0-9]+)?)//
64                  and return('NUM',$1);
65          s/^([A-Za-z][A-Za-z0-9_]*)//
66                  and return('VAR',$1);
67          s/^(.)//s
68                  and return($1,$1);
69      }
70  }
71
72  sub Run {
73      my($self)=shift;
74      $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
75                      #yydebug =>0xFF
76                    );
77  }
78  }; # end grammar
79
80
81  $Data::Dumper::Indent = 1;
82  Parse::Eyapp->new_grammar(
83    input=>$grammar,
84    classname=>'Rule6',
85    firstline =>7,
86    outputfile => 'Calc.pm',
87  );
88  my $parser = Rule6->new();
```

```
 89  $parser->YYData->{INPUT} = "a = -(2*3+5-1)\n";
 90  my $t = $parser->Run;
 91  print "\n***** Before ******\n";
 92  print Dumper($t);
 93
 94  my $attgram = new Language::AttributeGrammar <<'EOG';
 95
 96  # Compute the expression
 97  NUM:    $/.val = { $<attr> }
 98  TIMES:  $/.val = { $<left>.val * $<right>.val }
 99  PLUS:   $/.val = { $<left>.val + $<right>.val }
100  MINUS:  $/.val = { $<left>.val - $<right>.val }
101  UMINUS: $/.val = { -$<exp>.val }
102  ASSIGN: $/.val = { $<exp>.val }
103  EOG
104
105  my $res = $attgram->apply($t, 'val');
106
107  $Data::Dumper::Indent = 1;
108  print "\n***** After ******\n";
109  print Dumper($t);
110  print Dumper($res);
```

`CalcwithAttributeGrammar.pl`

The program computes the tree for expression for expression `a = -(2*3+5-1)` which is:

```
ASSIGN(TERMINAL,UMINUS(MINUS(PLUS(TIMES(NUM,NUM),NUM),NUM)))
```

The children of the binary nodes can be accessed through the `left` and `right` methods.

About the Encapsulation of Nodes

There is no encapsulation of nodes. The user/client knows that they are hashes that can be decorated with new keys/attributes. All nodes in the AST created by `%tree` are `Parse::Eyapp::Node` nodes. The only reserved field is `children` which is a reference to the array of children. You can always create a `Node` class *by hand* by inheriting from `Parse::Eyapp::Node`.

14 SOLVING CONFLICTS WITH THE *POSTPONED CONFLICT* STRATEGY

Yacc-like parser generators provide ways to solve shift-reduce mechanims 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 to 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 then 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.eyp` 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.eyp
   1  %{
   2  =head1 SYNOPSIS
   3
   4  See
   5
   6  =over 2
   7
   8  =item * File pascalenumeratedvsrange.eyp in examples/debuggintut/
   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.eyp
  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(
    expr_is_expr_TIMES_expr(
      expr_is_LP_expr_RP(
        expr_is_expr_PLUS_expr(
          RangeID(
            TERMINAL[x]
          ),
          expr_is_NUM(
            TERMINAL[2]
          )
        )
      ),
      expr_is_NUM(
        TERMINAL[3]
      )
    ),
    TERMINAL[..],
    expr_is_expr_DIV_expr(
      RangeID(
        TERMINAL[y]
      ),
      expr_is_NUM(
        TERMINAL[2]
      )
    )
```

```
  )
)
~/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 e = (x);
^D
type_decl_is_TYPE_ID_type(
  TERMINAL[TYPE],
  TERMINAL[e],
  ENUM(
    EnumID(
      TERMINAL[x]
    )
  )
)
```

Postponed Conflict Resolution: Shift-Reduce Conflicts

The program in `examples/debuggingtut/DynamicallyChangingTheParser2.eyp` illustrates how the postponed conflict strategy is used for shift-reduce conflicts. This is an extension of the grammar in `examples/debuggingtut/Debu`
The generated language is constituted by sequences like:

```
{ D; D; S; S; S; } {D; S} { S }
```

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*S}
```

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

```
~/LEyapp/examples/debuggingtut$ sed -ne '30,$p' DynamicallyChangingTheParser.eyp | 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;
```

15 NAMING SCHEMES

Explicit names can be given to grammar productions via the `%name` directive. An alternative to explicitly gave names to rules is to define a *naming scheme* via the Eyapp directive `%namingscheme`. This can be helpful when you inherit a large grammar and want to quickly build a parser. The ANSI C parser in

`examples/languages/C/ansic.eyp` is a good example. Another example is the Pascal parser in `examples/languages/pascal`

The Eyapp directive `%namingscheme` is followed by some Perl code. Such Perl code must return a reference to a subroutine that will be called each time a new production right hand side is parsed. The subroutine returns the name for the production.

The Perl code defining the handler receives a `Parse::Eyapp` object that describes the grammar. The code after the `%namingscheme` directive is evaluated during the early phases of the compilation of the input grammar. As an example of how to set a naming scheme, see lines 22-38 below (you can find this example and others in the directory `examples/naming` of the accompanying distribution):

```
lusasoft@LusaSoft:~/src/perl/Eyapp/examples/naming$ cat -n GiveNamesToCalc.eyp
   1  # GiveNamesToCalc.eyp
   2  %right  '='
   3  %left   '-' '+'
   4  %left   '*' '/'
   5  %left   NEG
   6  %right  '^'
   7
   8  %tree bypass
   9
  10  %{
  11  use base q{Tail};
  12
  13  sub exp_is_NUM::info {
  14    my $self = shift;
  15
  16    $self->{attr}[0];
  17  }
  18
  19  *exp_is_VAR::info = *var_is_VAR::info = \&exp_is_NUM::info;
  20  %}
  21
  22  %namingscheme {
  23    #Receives a Parse::Eyapp object describing the grammar
  24    my $self = shift;
  25
  26    $self->tokennames(
  27      '=' => 'ASSIGN',
  28      '+' => 'PLUS',
  29      '*' => 'TIMES',
  30      '-' => 'MINUS',
  31      '/' => 'DIV',
  32      '^' => 'EXP',
  33    );
  34
  35    # returns the handler that will give names
  36    # to the right hand sides
  37    \&give_token_name;
  38  }
  39  %%
  40
  41  line:
  42      exp
  43  ;
  44
  45  exp:
  46      NUM
  47    | VAR
  48    | var '=' exp
  49    | exp '+' exp
  50    | exp '-' exp
  51    | exp '*' exp
```

```
52    | exp '/' exp
53    | %no bypass exp_is_NEG
54       '-' exp %prec NEG
55    | exp '^' exp
56    | '(' exp ')'
57  ;
58
59  var:
60      VAR
61  ;
62  %%
63
64  unless (caller) {
65    my $t = __PACKAGE__->main(@ARGV);
66    print $t->str."\n";
67  }
```

The example uses a naming scheme that is provided by `Parse::Eyapp`: `Parse::Eyapp::Grammar::give_token_name`. The current provided naming schemes handlers are:

- `give_default_name`: The name of the production is the name of the Left Hand Side of the Production Rule concatenated with an underscore and the index of the production
- `give_lhs_name`: The name of the production is the name of the Left Hand Side of the Production Rule (this is the naming scheme used by the `%tree` directive when no explicit name was given)
- `give_token_name`: The name of the production is the Left Hand Side of the Production Rule followed by the word `_is_` followed by the concatenation of the names of the tokens in the right and side (separated by underscores).

All of these handlers are implemented inside the class `Parse::Eyapp::Grammar`. There is no need at line 37 to explicit the class name prefix since the naming scheme code is evaluated inside such class:

```
22  %namingscheme {
23    #Receives a Parse::Eyapp object describing the grammar
24    my $self = shift;
25
26    $self->tokennames(
27      '=' => 'ASSIGN',
28      '+' => 'PLUS',
29      '*' => 'TIMES',
30      '-' => 'MINUS',
31      '/' => 'DIV',
32      '^' => 'EXP',
33    );
34
35    # returns the handler that will give names
36    # to the right hand sides
37    \&give_token_name;
38  }
```

As it is illustrated in this example, the method `tokennames` of `Parse::Eyapp` objects provide a way to give identifier names to tokens that are defined by strings. When we execute the former module/program (modulino) with input `a=2*-3` we got the following output:

```
lusasoft@LusaSoft:~/src/perl/Eyapp/examples/naming$ eyapp -b '' GiveNamesToCalc.eyp
lusasoft@LusaSoft:~/src/perl/Eyapp/examples/naming$ ./GiveNamesToCalc.pm
Expressions. Press CTRL-D (Unix) or CTRL-Z (Windows) to finish:
a=2*-3
line_is_exp(var_is_VAR[a],exp_is_TIMES(exp_is_NUM[2],exp_is_NEG(exp_is_NUM[3])))
```

For each production rule the handler is called with arguments:

- the `Parse::Eyapp` object,
- the production index (inside the grammar),
- the left hand side symbol and a reference to a list with the symbols in the right hand side.

The following code of some version of `give_token_name` exemplifies how a naming scheme handler can be written:

```
lusasoft@LusaSoft:~/src/perl/Eyapp$ sed -ne '101,132p' lib/Parse/Eyapp/Grammar.pm | cat -n
   1  sub give_token_name {
   2    my ($self, $index, $lhs, $rhs) = @_;
   3
   4    my @rhs = @$rhs;
   5    $rhs = '';
   6
   7    unless (@rhs) { # Empty RHS
   8      return $lhs.'_is_empty';
   9    }
  10
  11    my $names = $self->{GRAMMAR}{TOKENNAMES} || {};
  12    for (@rhs) {
  13      if ($self->is_token($_)) {
  14        s/^'(.*)'$/$1/;
  15        my $name = $names->{$_} || '';
  16        unless ($name) {
  17          $name = $_ if /^\w+$/;
  18        }
  19        $rhs .= "_$name" if $name;
  20      }
  21    }
  22
  23    unless ($rhs) { # no 'word' tokens in the RHS
  24      for (@rhs) {
  25        $rhs .= "_$_" if /^\w+$/;
  26      }
  27    }
  28
  29    # check if another production with such name exists?
  30    my $name = $lhs.'_is'.$rhs;
  31    return $name;
  32  }
```

16 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*
- ocamlyacc tutorial at http://plus.kaist.ac.kr/~shoh/ocaml/ocamllex-ocamlyacc/ocamlyacc-tutorial/ocamlyacc-tutorial.html

17 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

18 CONTRIBUTORS

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

19 AUTHOR

Casiano Rodriguez-Leon (casiano@ull.es)

20 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.

21 LICENCE AND COPYRIGHT

Index