1 NAME

Parse::Eyapp::languageintro - Introduction to the Eyapp language

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 the token.

```
eyapp: head body tail ;
symbol: LITERAL  /* A string literal like 'hello' */
    |   ident
ident:  IDENT  /* IDENT is [A-Za-z_][A-Za-z0-9_]* */
head: headsec '%%'
headsec:  decl *
decl:  '\n'
    |   SEMANTIC typedecl symlist '\n'  /* SEMANTIC  is %semantic\s+token      */
    |   SYNTACTIC typedecl symlist '\n' /* SYNTACTIC is %syntactic\s+token     */
    |   TOKEN typedecl symlist '\n'     /* TOKEN     is %token                */
    |   ASSOC typedecl symlist '\n'     /* ASSOC     is %(left|right|nonassoc) */
    |   START ident '\n'                /* START     is %start                */
    |   HEADCODE '\n'                   /* HEADCODE  is %{ Perl code ... %}   */
    |   UNION CODE '\n'                 /* UNION CODE  see yacc/bison         */
    |   DEFAULTACTION CODE '\n'         /* DEFAULTACTION is %defaultaction    */
    |   TREE treeclauses? '\n'          /* TREE      is %tree                 */
    |   METATREE '\n'                   /* METATREE  is %metatree             */
    |   TYPE typedecl identlist '\n'    /* TYPE      is %type                 */
    |   EXPECT NUMBER '\n'              /* EXPECT    is %expect               */
                                        /* NUMBER    is \d+                   */
typedecl:   /* empty */
    |       '<' IDENT '>'
treeclauses: BYPASS ALIAS? | ALIAS BYPASS?
symlist:    symbol +
identlist:  ident +
body: rules * '%%'
rules: IDENT ':' rhss ';'
rhss: rule <+ '|'>
rule:   optname rhs (prec epscode)?
rhs:  rhseltwithid *
rhseltwithid :
      rhselt '.' IDENT
    | '$' rhselt
    | rhselt
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*)?\+  */
optname: (NAME IDENT)?          /* NAME is %name */
       | NOBYPASS IDENT         /* NOBYPASS is %no\s+bypass */
prec: PREC symbol               /* PREC is %prec */
epscode:  code ?
code:
    CODE           /* CODE     is { Perl code ... }         */
  | BEGINCODE      /* BEGINCODE is %begin { Perl code ... } */
```

```
tail:  TAILCODE ?  /* TAILCODE is { Perl code ... } */
```

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: *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).

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).

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 '%%'
```

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/Calc.eyp` that accompanies this distribution. This file implements a trivial calculator. Here is the header section:

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ sed -ne '1,11p' Calc.eyp | cat -n
 1  # examples/Calc.eyp
 2  %right  '='
 3  %left   '-' '+'
 4  %left   '*' '/'
 5  %left   NEG
 6  %right  '^'
 7  %{
 8  my %s; # symbol table
 9  %}
10
11  %%
```

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 parenthesization. Thus, the expression

```
4 - 5 - 9
```

will be interpreted as

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

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 IDENT` declares `IDENT` 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`. See section Solving Ambiguities and Conflicts to know more about Shift/Reduce conflicts.

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$ cat -n bugyapp2.eyp
     1  %strict
     2  %%
     3  expr: NUM;
     4  %%
pl@nereida:~/LEyapp/examples$ 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$ cat -n bugyapp3.eyp
     1  %strict
     2  %token NUM
     3  %%
     4  expr: NUM;
     5  %%
pl@nereida:~/LEyapp/examples$ eyapp bugyapp3.eyp
pl@nereida:~/LEyapp/examples$
```

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

Default Action Directive

In `Parse::Eyapp` you can modify the default action using the `%defaultaction { Perl code }` directive. See section Default actions.

Tree Construction Directives

`Parse::Eyapp` facilitates the construction of concrete syntax trees and abstract syntax trees (abbreviated AST from now on) through the `%tree` `%metatree` directives. See section Abstract Syntax Trees : %tree and %name and *Parse::Eyapp::translationschemestut*.

Syntactic and Semantic Tokens

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 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*).

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. Asociated 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/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]` beeing 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/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:

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ sed -ne '1,4p' intermediateaction2.yp
%%
S:  'a' { $_[1]x4 }.mid 'a' { print "$_[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:

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

when given input `aa` the execution will produce as output `aaaa, aaaa, a`.

Example of Body Section

Following with the calculator example, the body is:

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ sed -ne '12,48p' Calc.eyp | cat -n
 1  start:
 2      input { \%s }
 3  ;
 4
 5  input: line *
 6  ;
 7
 8  line:
 9    '\n'         { undef }
10    | exp '\n'   { print "$_[1]\n" if defined($_[1]); $_[1] }
11    | error  '\n'
12        {
13          $_[0]->YYErrok;
14          undef
15        }
16  ;
17
18  exp:
19      NUM
20    | $VAR                    { $s{$VAR} }
21    | $VAR '=' $exp           { $s{$VAR} = $exp }
22    | exp.left '+' exp.right { $left + $right }
23    | exp.left '-' exp.right { $left - $right }
24    | exp.left '*' exp.right { $left * $right }
25    | exp.left '/' exp.right
26      {
27         $_[3] and return($_[1] / $_[3]);
28         $_[0]->YYData->{ERRMSG} = "Illegal division by zero.\n";
29         $_[0]->YYError; # Pretend that a syntactic error ocurred: _Error will be called
30         undef
31      }
32    | '-' $exp %prec NEG     { -$exp }
33    | exp.left '^' exp.right { $left ** $right }
34    | '(' $exp ')'           { $exp }
35  ;
36
37  %%
```

This example does not uses any of the Eyapp extensions (with the exception of the *star list* at line 5) and the dot and dollar notations. Please, see the *Parse::Yapp* pages and elsewhere documentation on *yacc* and *bison* for more information.

Solving Ambiguities and Conflicts

When Eyapp analizes a grammar like:

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ 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:

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ 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 '-'
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ ls -ltr | tail -1
-rw-rw----  1 pl users   1082 2007-02-06 08:26 ambiguities.output
```

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 worked 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).
- The precedences and associativities are associated with 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 character 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 nonassociating implies error.

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.

The Tail

The tail section contains Perl code. Usually the lexical analyzer and the Error management subroutines go there. A better practice however is to isolate both subroutines in a module and use them in the grammar. An example of this is in files `examples/CalcUsingTail.eyp` and `examples/Tail.pm`.

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 accesible inside the lexical analyzer. Specially interesting is the `$_[0]->YYData` method which provides access to the user data area.

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

See below how to write a lexical analyzer (file `examples/Calc.eyp`):

```
 1  sub make_lexer {
 2    my $input = shift;
 3
 4    return sub {
 5      my $parser = shift;
 6
 7      for ($$input) {
 8        m{\G[ \t]*}gc;
 9        m{\G([0-9]+(?:\.[0-9]+)?)}gc   and return ('NUM',$1);
10        m{\G([A-Za-z][A-Za-z0-9_]*)}gc and return ('VAR',$1);
11        m{\G\n}gc                      and do { $lineno++; return ("\n", "\n") };
12        m{\G(.)}gc                     and return ($1,$1);
13
14        return('',undef);
15      }
16    }
17  }
```

The subroutine `make_lexer` creates the lexical analyzer as a closure. The lexer returned by `make_lexer` is used by the `YYParse` method:

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ sed -ne '90,97p' Calc.eyp | cat -n
 1  sub Run {
 2      my($self)=shift;
 3      my $input = shift or die "No input given\n";
 4
 5      return $self->YYParse( yylex => make_lexer($input), yyerror => \&_Error,
 6        #yydebug =>0x1F
 7      );
 8  }
```

The Error Report Subroutine

The Error Report subroutine is also a parser method, and consequently receives as parameter the parser object.

See the error report subroutine for the example in `examples/Calc.eyp`:

```
 1  %%
 2
 3  my $lineno = 1;
 4
 5  sub _Error {
 6    my $parser = shift;
 7
 8      exists $parser->YYData->{ERRMSG}
 9    and do {
10        print $parser->YYData->{ERRMSG};
11        delete $parser->YYData->{ERRMSG};
12        return;
13    };
14    my($token)=$parser->YYCurval;
15    my($what)= $token ? "input: '$token'" : "end of input";
16    my @expected = $parser->YYExpect();
17    local $" = ', ';
```

```
18    print << "ERRMSG";
19
20  Syntax error near $what (lin num $lineno).
21  Expected one of these terminals: @expected
22  ERRMSG
23  }
```

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

Using an Eyapp Program

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

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

Syntax error near input: ')' (lin num 4).   # c=a*3+4)-5
Expected one of these terminals: -, /, ^, *, +,

7                                           # a = a+1
========= Symbol Table ==============
a = 7
b = 1
c = 22
```

Lists and Optionals

The elements of a 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 Semantic of Lists Operators

The + operator

The grammar:

```
pl@nereida:~/LEyapp/examples$ head -12 List3.yp | cat -n
 1  # List3.yp
 2  %semantic token 'c'
 3  %{
 4  use Data::Dumper;
 5  %}
 6  %%
 7  S:      'c'+  'd'+
 8             {
 9                print Dumper($_[1]);
10                print Dumper($_[2]);
11             }
12  ;
```

Is equivalent to:

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

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_#number
```

whose children are the attributes associated with the items in the plus list. The #number in _PLUS_LIST_#number is the ordinal of the production rule as it appears in the .output file. As it happens when using the %tree directive syntactic tokens are skipped.

When executing the example above but under the %tree directive the ouput changes:

```
pl@nereida:~/LEyapp/examples$ head -3 List3.yp; eyapp List3.yp
# List3.yp
%semantic token 'c'
%tree

pl@nereida:~/LEyapp/examples$ use_list3.pl
ccdd
$VAR1 = bless( {
         'children' => [
           bless( { 'children' => [], 'attr' => 'c', 'token' => 'c' }, 'TERMINAL' ),
           bless( { 'children' => [], 'attr' => 'c', 'token' => 'c' }, 'TERMINAL' )
         ]
       }, '_PLUS_LIST_1' );
$VAR1 = bless( { 'children' => [] }, '_PLUS_LIST_2' );
```

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

When Nodes Dissappear 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 dont 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:

```
pl@nereida:~/LEyapp/examples$ head -19 ListWithRefs1.eyp | cat -n
 1  # ListWithRefs.eyp
 2  %semantic token 'c' 'd'
 3  %{
 4  use Data::Dumper;
 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'
17  ;
18
19  %%
```

To activate the *tree semantic* for lists we use the `yybuildingtree` option of `YYParse`:

```
pl@nereida:~/LEyapp/examples$ tail -7 ListWithRefs1.eyp | cat -n
     1  sub Run {
     2      my($self)=shift;
     3      $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
     4        yybuildingtree => 1,
     5        #, yydebug => 0x1F
     6      );
     7  }
```

The execution gives an ouput like this:

```
pl@nereida:~/LEyapp/examples$ eyapp ListWithRefs1.eyp; use_listwithrefs1.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' => []
               }, '_PLUS_LIST_2' );
_PLUS_LIST_2
```

Though `'d'` was declared semantic the default action assoaciated 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:

```
pl@nereida:~/LEyapp/examples$ head -22 ListWithRefs.eyp | cat -n
 1  # ListWithRefs.eyp
 2  %semantic token 'c'
 3  %{
 4  use Data::Dumper;
 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'
17       {
18         bless { attr => $_[1], children =>[]}, 'DES';
19       }
20  ;
21
22  %%
```

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

```
pl@nereida:~/LEyapp/examples$ eyapp ListWithRefs.eyp; use_listwithrefs.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' => [],
                                          'attr' => 'd'
                                        }, 'DES' ),
                                 bless( {
                                          'children' => [],
                                          'attr' => 'd'
                                        }, 'DES' )
                               ]
               }, '_PLUS_LIST_2' );
_PLUS_LIST_2(DES,DES)
```

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. Thefollowing 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 jsut created nodes. In a scalar context returns a reference to the first of such trees. See an example:

```
pl@nereida:~/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 explictly constructed:

```
pl@nereida:~/LEyapp/examples$ head -28 ListWithRefs2.eyp| cat -n
 1  # ListWithRefs2.eyp
 2  %semantic token 'c'
 3  %{
 4  use Data::Dumper;
 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  %%
```

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

```
pl@nereida:~/LEyapp/examples$ 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 Xs 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 cs and the list of ds. 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 - obivated:

```
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 supressed.

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$ sed -ne '1,27p' CsBetweenCommansAndDWithNames.eyp | cat -n
 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 execution shows the renamed nodes:
pl@nereida:~/LEyapp/examples$ 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 c<_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 explictly 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 antiderivation 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
```

Names for attributes

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

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

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.

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 vÃąlida\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:

```
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                {
39                  my $self = shift;
40                  $self->YYName('SUBSTRACT'); # rename it
41                  $self->YYBuildAST(@_); # build the node
42                }
43          |   %name TIMES
44              exp '*' exp
45          |   %name DIV
46              exp '/' exp
```

```
47          |   %name UMINUS
48              '-' exp %prec NEG
49          |  '(' exp ')'  { $_[2] }
50  ;
```

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

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

Abstract Syntax Trees : %tree and %name

`Parse::Eyapp` facilitates the construction of concrete syntax trees and abstract syntax trees (abbreviated AST from now on) through the `%tree` directive. Nodes in the AST are blessed in the production `name`. By default the name of a production is 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@nereida:~/src/perl/YappWithDefaultAction/examples$ sed -ne '9,28p' 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:

```
_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`:

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ sed -ne '28,42p' treewithoutnames.output
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`.

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:

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ 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 explictly 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:

```
        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 (`[+*?]`).

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`. See section 'Compiling with eyapp and treereg' in *Parse::Eyapp* for an example.

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:

```
Variable:
    %name  VARARRAY
    $ID ('[' binary ']') <%name INDEXSPEC +>
      {
        my $self = shift;
        my $node =  $self->YYBuildAST(@_);
        $node->{line} = $ID->[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 supress nodes in the AST being built. See the following example (file `examples/returnnonode.yp`):

```
nereida:~/src/perl/YappWithDefaultAction/examples> sed -ne '1,11p' returnnonode.yp | cat -n
 1  %tree
 2  %semantic token 'a' 'b'
 3  %%
 4  S:  /* empty */
 5      | S A
 6      | S B
 7  ;
 8  A : 'a'
```

```
 9  ;
10  B : 'b' { }
11  ;
```

since the action at line 10 returns `undef` the `B : 'b'` subtree will not be inserted in the AST:

```
nereida:~/src/perl/YappWithDefaultAction/examples> usereturnnonode.pl
ababa
S_2(S_3(S_2(S_3(S_2(S_1,A_4(TERMINAL[a]))),A_4(TERMINAL[a]))),A_4(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):

```
nereida:~/src/perl/YappWithDefaultAction/examples> \
         sed -ne '1,10p' intermediateactiontree.yp | cat -n
 1  %semantic token 'a' 'b'
 2  %tree bypass
 3  %%
 4  S:    /* empty */
 5      | S A.f { $f->{attr} = "A"; $f; } A
 6      | S B.f { $f->{attr} = "B"; $f; } B
 7  ;
 8  A : %name A 'a'
 9  ;
10  B : %name B 'b'
```

See the client program running:

```
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";
nereida:~/src/perl/YappWithDefaultAction/examples> useintermediateactiontree.pl
aabbaa
S_2(S_4(S_2(S_1,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` diferences 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 former *Parse::Eyapp* `/SYNOPSIS` example:

```
$parser->YYData->{INPUT} = "2*-3+b*0;--2\n";
my $t = $parser->Run;
local $Parse::Eyapp::Node::INDENT=2;
print "Syntax Tree:",$t->str;
```

gives as result the following output:

```
nereida:~/src/perl/YappWithDefaultAction/examples> synopsis.pl
Syntax Tree:
EXPRESION_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
) # EXPRESION_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 identifer, 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 `examples/Types.eyp`) 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 EXPRESION_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

************
EXPRESION_LIST(ASSIGN(TERMINAL[a],PLUS(TIMES(NUM[2],UMINUS(NUM[3])),TIMES(VAR[b],NUM[0]))))
************
EXPRESION_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 dolar 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 wil 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.

As an example, 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);
```

3 SEE ALSO

- *Parse::Eyapp*, *Parse::Eyapp::eyapplanguageref*, *Parse::Eyapp::debugingtut*, *Parse::Eyapp::defaultactionsintro*, *Parse::Eyapp::translationschemestut*, *Parse::Eyapp::Driver*, *Parse::Eyapp::Node*, *Parse::Eyapp::YATW*, *Parse::Eyapp::Treeregexp*, *Parse::Eyapp::Scope*, *Parse::Eyapp::Base*,
- 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/MatchingTrees.pdf
- 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/
- 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),
- Man pages of bison(1),
- *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

4 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

5 AUTHOR

Casiano Rodriguez-Leon (casiano@ull.es)

6 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. Special thanks to my family and Larry Wall.

7 LICENCE AND COPYRIGHT

Index