Parse::Eyapp Tutorial

Casiano Rodriguez-Leon

January 25, 2007

Contents

1 NAME

Parse::Eyapp

2 VERSION

1.06503

3 SYNOPSIS

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

sub TERMINAL::info {
  $_[0]{attr}
}

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 EXPRESION_LIST + ';'>  { $_[1] } /* list of expressions separated by ';' */
  ;

  /* The %name directive defines the name of the class to which the node being built belongs */
  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] }  /* Let us simplify a bit the tree */
  ;

  %%
  sub _Error { die "Syntax error near ".($_[0]->YYCurval?$_[0]->YYCurval:"end of file")."\n" }

  sub _Lexer {
    my($parser)=shift; # The parser object

    for ($parser->YYData->{INPUT}) {
      s/^\s+//;
      $_ eq '' and return('',undef);
      s/^([0-9]+(?:\.[0-9]+)?)// and return('NUM',$1);
      s/^([A-Za-z][A-Za-z0-9_]*)// and return('VAR',$1);
      s/^(.)//s and return($1,$1);
    }
  }

  sub Run {
      my($self)=shift;
      $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error, );
  }
}; # 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} = "2*-3+b*0;--2\n"; # Set the input
my $t = $parser->Run;                     # Parse it!
local $Parse::Eyapp::Node::INDENT=2;
print "Syntax Tree:",$t->str;

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

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

local $Parse::Eyapp::Node::INDENT=0;
print "\nSyntax Tree after transformations:\n",$t->str,"\n";
```

4 Introduction to `Parse::Eyapp`

Parse::Eyapp (Extended yapp) is a collection of modules that extends Francois Desarmenien Parse::Yapp 1.05. Eyapp extends yacc/yapp syntax with the functionalities briefly described in this section. This is an introductory tutorial. For a reference guide see *Parse::Eyapp*. If you are not familiar with *yacc* or *yapp* and you can speak Spanish start reading the contents in http://nereida.deioc.ull.es/~pl/perlexamples/chapter_parseeyapp.html.

5 Input from strings

Grammars can be compiled from a file or from source on the fly (See the synopsis section for an example).

6 Names for attributes

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

```
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`. For example:

```
exp:  -' $exp %prec NEG { -$exp }
```

7 Lists and Optionals

Lists, optional lists, list separated by tokens, etc. like in the start rule in the Synopsis example can be used:

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

which defines `line` as the language of non empty lists of `exp` elements separated by semicolons. The use of `%name EXPRESION_LIST` gives a name to the created list. Actually the right hand side of this production has only one element which is the reference to the list. The associated action `{ $_[1] }` makes the generated parser to return the reference to such list.

The former rule is almost equivalent to:

```
line: line ';' exp  { push $_[1]->{children}, $_[3] }
    | exp           { bless { children => [ $_[1] ] }, 'EXPRESION_LIST' }
```

8 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 that follows the directive is executed when reducing by any production for which no explicit action was specified. 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
...
```

8.1 Compiling with eyapp

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
```

9 Abstract Syntax Trees

`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*) However, a production can be *named* using the `%name` directive. Therefore, in the following code:

```
exp:
       %name NUM  NUM
     | %name VAR   VAR
     | %name ASSIGN VAR '=' exp
     . ............................
     | %name UMINUS '-' exp %prec NEG
     |   '(' exp ')'  { $_[2] }  /* Let us simplify a bit the tree */
```

we are explictly naming the productions. Thus, the node corresponding to the production `exp: VAR '=' exp` will be named `ASSIGN`. Explicit actions can be specified by the programmer like in

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

the action receives as arguments the references to the children nodes already built. The programmer can influence the shape of the tree by inserting this explicit actions. In the 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.

When a *explicit user action* returns s.t. that is not a reference no child will be inserted in the father of the current production.

9.1 Displaying Trees

All the node classes build by `%tree` inherit from `Parse::Eyapp::Node` and consequently have acces to the methods provided in such module. Among them is the `str` method which dumps the tree. The *str* method traverses the syntax tree dumping the type of the node being visited in a string. If the node has a method `info` it will be executed and its result concatenated to the string. Thus, in the Synopsis example, by adding the `info` method to the class `TERMINAL`:

```
sub TERMINAL::info {
  $_[0]{attr}
}
```

we achieve the insertion of attributes in the string build by `str` (see the partial output of `synopsis.pl` in section Syntactic and Semantic tokens).

The existence of some methods (like `footnote`) and the values of some package variables influence the behavior of `str`. Among the most important are:

```
@PREFIXES = qw(Parse::Eyapp::Node::); # Prefixes to supress
$INDENT = 0; # 0 = compact, 1 = indent, 2 = indent and include Types in closing parenthesis
$STRSEP = ',';
$DELIMITER = '[';
$FOOTNOTE_HEADER = "\n---------------------------\n";
$FOOTNOTE_SEP = ")\n";
$FOOTNOTE_LEFT = '^{';
$FOOTNOTE_RIGHT = '}';
$LINESEP = 4;
```

9.2 TERMINAL nodes

Nodes named `TERMINAL` correspond to tokens provided by the lexical analyzer. They are `Parse::Eyapp::Node` nodes (hashes) with an attribute `attr` holding the attribute provided by the lexical analyzer. The `attr` method can be used to get/set the attribute.

9.3 User Attributes and System Attributes

All the nodes in the AST are `Parse::Eyapp::Node` nodes. They are hashes that the user can decorate with new keys/attributes. The only reserved words are those listed in the reference section. Basically they have a `children` key. `TERMINAL` nodes have the `attr` key.

9.4 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 `'+'`, `'='`, in the Synopsis example) are considered `syntactic tokens`. Tokens declared by an identifier (like `NUM` or `VAR` in the Synopsis example) are by default considered `semantic tokens`. **Syntactic tokens are eliminated when building the syntactic tree**. Thus, the first print in the former 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
```

9.5 Saving the Information In Syntactic Tokens

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 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 a `TERMINAL::save_attributes` method exists and if so it will be called when visiting a syntactic terminal. The method receives as argument - additionally to the reference to the `TERMINAL` node - 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!
}
```

9.6 The directives `%syntactic token` and `%semantic 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),B,C(TERMINAL))`.

9.7 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 is 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 the NUM node will be bypassed and the child TERMINAL built from the information provided by the lexical analyzer will be renamed 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 ')'
```

As consequence of the blind application of the *bypass rule* 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 is the equivalent of the *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 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.

9.8 Explictly building nodes with the `YYBuildAST` method

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 example defines the expression to access an array element as an identifier followed by a non empty list of binary expressions. 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. 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 }
```

9.9 The `child` and `descendant` methods

Access to the children of the AST is achieved through the `children` and `child` methods. More general is the `descendant` method that returns the descendant of a node given its coordinates. See a session with the debugger:

```
  DB<7> x $t->child(0)->child(0)->child(1)->child(0)->child(2)->child(1)->str
0  '
BLOCK[8:4:test]^{0}(
  CONTINUE[10,10]
)
  DB<8> x $t->descendant('.0.0.1.0.2.1')->str
0  '
BLOCK[8:4:test]^{0}(
  CONTINUE[10,10]
```

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

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 of a small calculator:

```
use Parse::Eyapp;
use Language::AttributeGrammar;

my $grammar = q{
... # priority declarations. Like in previous examples
%tree bypass alias

%%
line: $exp  { $_[1] }
;

exp:
    %name NUM
          $NUM
        | %name VAR
          $VAR
    ............ # as in the bypass example
}; # end grammar

Parse::Eyapp->new_grammar(
  input=>$grammar, classname=>'Rule6', firstline =>7,
);
my $parser = Rule6->new();
$parser->YYData->{INPUT} = "a = -(2*3+5-1)\n";
my $t = $parser->Run;
my $attgram = new Language::AttributeGrammar <<'EOG';
# Compute the expression
NUM:    $/.val = { $<attr> }
TIMES:  $/.val = { $<left>.val * $<right>.val }
```

```
PLUS:   $/.val = { $<left>.val + $<right>.val }
MINUS:  $/.val = { $<left>.val - $<right>.val }
UMINUS: $/.val = { -$<exp>.val }
ASSIGN: $/.val = { $<exp>.val }
EOG

my $res = $attgram->apply($t, 'val');
```

10 Tree Regular Expressions

`Parse::Eyapp` introduces a new language called *Tree Regular Expressions* that easies the transformation of trees. Let us recall the previous example used in the `bypass` section:

```
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
```

The call to the constructor `new` builds a `Parse::Eyapp::Treeregexp` object. The subsequent call to the method `$p->generate` compiles the object producing tree-transformations built according to the specification given in the treeregexp program. A tree transformation is a `Parse::Eyapp::YATW` object. The example contains four tree programa transformations named `constantfold`, `zero_times_whatever`, `whatever_times_zero` and <uminus>. These transformations can be grouped in transformation families. Such families of transformations can be applied to any `Parse::Eyapp::Node` trees. An special variable `@PACKAGE::all` refers to the whole set of transformations in the program. Here `PACKAGE` refers to the package where the transformations live. When no `PACKAGE` argument is specified in the call to `new` - as is the case in this example - the package of the caller is used instead. The call `$t->s(@all)` proceeds to the execution of the method `s` (for substitution) using all the specified transformations. The transformations will be iteratively applied to all nodes of the tree until there are no changes. Summarizing, that means that

- All `UMINUS` nodes whose only child is a number `NUM` will be substituted by the `NUM` node but with the sign changed
- Constant folding will be applied: trees representing constants expressions will be substituted by a `NUM` node representing its value
- All the `TIMES` nodes with one child holding the value 0 will be substituted by that child

10.1 The Syntax of Treeregexp

The example illustrates the syntax of the language. A tree transformation conforms to the syntax:

```
treeregexp:
    IDENT ':' treereg ('and' CODE)?  ('=>' CODE)?
```

like in:

```
zero_times_whatever: TIMES(NUM, .) and { $NUM->{attr} == 0 } => { $_[0] = $NUM }
```

The IDENT is the name given to the tree transformation. A tree transformation is actually a `Parse::Eyapp::YATW` object. After generation time two package objects are created per transformation:

- A subroutine with name `zero_times_whatever` holding the actual code for the tree transformation will be available and
- A scalar variable named `$zero_times_whatever` will refer to the `Parse::Eyapp::YATW` tree transformation object.

These names live in the package specified by the user in the call to `new` through the `PACKAGE` argument. When no package name is specified the name of the caller package is used instead.

After the `IDENT` and the colon comes the **treeregexp**. The treeregexp is a term, that is a parenthesized description of the shape of the tree like `TIMES(NUM, .)` which says: *match nodes of type* `TIMES` *whose left child is a* `NUM` *and whose right child is whatever.* The dot stands for *whatever* and is a treeregexp that matches any node.

Then comes the reserved word `and` and some Perl code specifying the semantic conditions for the node being visited to match

```
{ $NUM->{attr} == 0 }
```

The code can access to the different subtrees using lexical variables whose names match the type of the node. Thus, in the example:

```
zero_times_whatever: TIMES(NUM, .) and { $NUM->{attr} == 0 } => { $_[0] = $NUM }
```

variable `$NUM` refers to the left child while variables `$TIMES` and `$_[0]` will refer refer to the node being visited. When more than one node of the same type exists (for instance `TIMES(NUM,NUM)`) the associated lexical variable changes its type from scalar to array and thus if several `NUM` nodes appear in the term we will speak about `$NUM[0]`, `$NUM[1]`, etc.

10.2 Separated Compilation with `treereg`

A Treeregexp program can be isolated in a file an compiled with the program `treereg`. The default extension is `.trg`. See the following example:

```
nereida:~/src/perl/YappWithDefaultAction/examples> cat -n Shift.trg
 1  # File: Shift.trg
 2  {
 3    sub log2 {
 4      my $n = shift;
 5      return log($n)/log(2);
 6    }
 7
 8    my $power;
 9  }
10  mult2shift: TIMES($e, NUM($m)) and { $power = log2($m->{attr}); (1 << $power) == $m->{attr} }
11    => {
12      $_[0]->delete(1);
13      $_[0]->{shift} = $power;
14      $_[0]->type('SHIFTLEFT');
15    }
```

Note that auxiliary support code can be inserted at any point between transformations (lines 2-6). The code will be inserted (without the defining curly brackets) at that point. Note also that the lexical variable `$power` is visible inside the definition of the `mult2shift` transformation.

A treeregexp like `$e` matches any node. A reference to the node is saved in the lexical variable `$e`. The scope of the variable `$e` is the current tree transformation, i.e. `mult2shift`. Such kind of treeregexps are called **scalar treeregexps**.

The call to the `delete` method at line 12 deletes the second child of the node being visited (i.e. `NUM($m)`).

The call to `type` at line 14 retypes the node as a `SHIFTLEFT` node.

The program is compiled using the script `treereg`:

```
nereida:~/src/perl/YappWithDefaultAction/examples> treereg Shift
nereida:~/src/perl/YappWithDefaultAction/examples> ls -ltr | tail -1
-rw-rw----  1 pl users   1405 2006-11-06 14:09 Shift.pm
```

The module Shift.pm contains the code implementing the tree transformations.

The client program follows:

```
nereida:~/src/perl/YappWithDefaultAction/examples> cat -n useruleandshift.pl
#!/usr/bin/perl -w
use strict;
use Rule6;
use Shift;
{ no warnings; *TERMINAL::info = \&TERMINAL::attr; }

sub SHIFTLEFT::info { $_[0]{shift} }

$Data::Dumper::Indent = 1;
my $parser = new Rule6();
$parser->YYData->{INPUT} = <>;
my $t = $parser->Run;
print "***********\n",$t->str,"\n";
$t->s(@Shift::all);
print "***********\n",$t->str,"\n";
```

Multiplications by a power of two are substituted by the corresponding shifts:

```
nereida:~/src/perl/YappWithDefaultAction/examples> useruleandshift.pl
a=b*8
***********
ASSIGN(TERMINAL[a],TIMES(VAR(TERMINAL[b]),NUM(TERMINAL[8])))
***********
ASSIGN(TERMINAL[a],SHIFTLEFT[3])
```

10.3 Regexp Treeregexps

We can use an ordinary regular expression `regexp` inside the term part of a treeregexp. The `constantfold` transformation in the Synopsis example shows how:

```
constantfold: /TIMES|PLUS|DIV|MINUS/:bin(NUM($x), NUM($y))
  => {
    my $op = $Op{ref($bin)};
    $x->{attr} = eval  "$x->{attr} $op $y->{attr}";
    $_[0] = $NUM[0];
  }
```

The regexp is specified between division slashes `/`. It is legal to specify options after the second slash (like `e`, `i`, etc.). The optional identifier `bin` after the regexp indicates the name for the lexical variable holding a copy that references the node. If no identifier is specified, the special variable `$W` is used instead. If the treeregexp has several anonymous regexp or dot treeregexps they will be stored in the array variable `@W`.

The operation of the ordinary string oriented regexps are slightly modified when they are used inside a treeregexp. **by default the option `x` will be assumed**. The treeregexp compiler will automatically insert it. Use the new option `X` (upper case X) if you want to supress such behavior. **There is no need also to insert `\b` word anchors** to delimit identifiers: all the identifiers in a regexp treeregexp are automatically surrounded by `\b`. Use the option `B` (upper case B) to supress this behavior.

The following fragment of the type checking stage of a simple compiler shows that `x` is implictly assumed:

```
# Binary Operations
bin: / PLUS
      |MINUS
      |TIMES
      |DIV
      |MOD
```

```
      |GT
      |GE
      |LE
      |EQ
      |NE
      |LT
      |AND
      |EXP
      |OR
     /($x, $y)
  => {
    $x = char2int($_[0], 0);
    $y = char2int($_[0], 1);

    if (($x->{t} == $INT) and ( $y->{t} == $INT)) {
      $_[0]->{t} = $INT;
      return 1;
    }
    type_error("Incompatible types with operator '".($_[0]->lexeme)."'", $_[0]->line);
  }
```

With the natural Perl regexp semantic the language reserved word `WHILE` would match the regexp (see the `LE` for *less or equal*) leading to an erroneous type checking. The automatic insertion of word anchors prevent it.

10.4 Matching Trees

Both the transformation objects in `Parse::Eyapp::YATW` and the nodes in `Parse::Eyapp::Node` have a method named `m` for matching.

For a `Parse::Eyapp::YATW` object, the method -when called in a list context- returns a list of `Parse::Eyapp::Node::Match` nodes referencing the nodes of the actual tree that have matched. The nodes in the list are organized in a hierarchy.

The nodes are sorted in the list of trees (a forest) according to a depth-first visit of the actual tree `$t`.

In a scalar context `m` returns the first element of the list.

Let us denote by `$t` the actual tree being searched and `$r` one of the `Parse::Eyapp::Node::Match` nodes in the resulting forest. Then we have the following methods:

- The method `$r->node` return the node `$t` of the actual tree that matched
- The method `$r->father` returns the tree in the matching forest. The father is defined by this property: `$r->father->node` is the nearest ancestor of `$r->node` that matched with the treeregexp pattern. That is, there is no ancestor that matched between `$r->node` and `$r->father->node`. Otherwise `$r->father` is `undef`
- The method `$r->coord` returns the coordinates of the actual tree that matched using s.t similar to the Dewey notation. for example, the coordinate `".1.3.2"` denotes the node `$t->child(1)->child(3)->child(2)`, where `$t` is the root of the search.
- The method `$r->depth` returns the depth of `$r->node` in `$t`.
- When called as a `Parse::Eyapp::Node` method, `$r->names` returns the array of names of the transformations that matched.

The following example illustrates a use of `m` as a `Parse::Eyapp:YATW` method. It solves a problem of scope analysis in a C compiler: matching each `RETURN` statement with the function that surrounds it. The treeregexp used is:

```
retscope: /FUNCTION|RETURN/
```

and the code that solves the problem is:

```
# Scope Analysis: Return-Function
my @returns = $retscope->m($t);
for (@returns) {
  my $node = $_->node;
```

```
  if (ref($node) eq 'RETURN') {
    my $function = $_->father->node;
    $node->{function}  = $function;
    $node->{t} = $function->{t};
  }
}
```

The first line gets a list of `Parse::Eyapp::Node::Match` nodes describing the actual nodes that matched `/FUNCTION|RETURN/`. If the node described by `$_` is a `'RETURN'` node, the expresion `$_->father->node` must necessarily point to the function node that surrounds it.

The second example shows the use of `m` as a `Parse::Eyapp::Node` method.

```
nereida:~/src/perl/YappWithDefaultAction/examples> cat -n m2.pl
#!/usr/bin/perl -w
use strict;
use Rule6;
use Parse::Eyapp::Treeregexp;

Parse::Eyapp::Treeregexp->new( STRING => q{
  fold: /times|plus|div|minus/i:bin(NUM($n), NUM($m))
  zero_times_whatever: TIMES(NUM($x), .) and { $x->{attr} == 0 }
  whatever_times_zero: TIMES(., NUM($x)) and { $x->{attr} == 0 }
})->generate();

# Syntax analysis
my $parser = new Rule6();
print "Expression: "; $parser->YYData->{INPUT} = <>;
my $t = $parser->Run;
local $Parse::Eyapp::Node::INDENT = 1;
print "Tree:",$t->str,"\n";

# Search
my $m = $t->m(our ($fold, $zero_times_whatever, $whatever_times_zero));
print "Match Node:",$m->str,"\n";
```

When executed with input `0*0*0` the program generates this output:

```
nereida:~/src/perl/YappWithDefaultAction/examples> m2.pl
Expression: 0*0*0
Tree:
TIMES(
  TIMES(
    NUM(
      TERMINAL
    ),
    NUM(
      TERMINAL
    )
  ),
  NUM(
    TERMINAL
  )
)
Match Node:
Match[TIMES:0:whatever_times_zero](
  Match[TIMES:1:fold,zero_times_whatever,whatever_times_zero]
)
```

The representation of `Match` nodes by `str` deserves a comment. `Match` nodes have their own `info` method. It returns a string containing the concatenation of the class of `$r->node` (i.e. the actual node that matched), the depth (`$r->depth`) and the names of the transformations that matched (as provided by the method `$r->names`)

10.5 The SEVERITY option of Parse::Eyapp::Treeregexp::new

The SEVERITY option of Parse::Eyapp::Treeregexp::new controls the way matching succeeds regarding the number of children. To illustrate its use let us consider the following example. The grammar Rule6 used by the example is similar to the one in the Synopsis example.

```
nereida:~/src/perl/YappWithDefaultAction/examples> cat -n numchildren.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use Rule6;
 4  use Parse::Eyapp::Treeregexp;
 5  use Parse::Eyapp::Node;
 6
 7  sub TERMINAL::info { $_[0]{attr} }
 8
 9  my $severity = shift || 0;
10  my $parser = new Rule6();
11  $parser->YYData->{INPUT} = shift || '0*2';
12  my $t = $parser->Run;
13
14  my $transform = Parse::Eyapp::Treeregexp->new(
15    STRING => q{
16      zero_times_whatever: TIMES(NUM($x)) and { $x->{attr} == 0 } => { $_[0] = $NUM }
17    },
18    SEVERITY => $severity,
19    FIRSTLINE => 15,
20  )->generate;
21
22  $t->s(our @all);
23
24  print $t->str,"\n";
```

The program gets the severity level from the command line (line 9). The specification of the term TIMES(NUM($x)) inside the transformation zero_times_whatever does not clearly state that TIMES must have two children. There are several interpretations of the treregexp depending on the level fixed for SEVERITY:

- 0: TIMES must have at least one child. Don't care if it has more.
- 1: TIMES must have exactly one child.
- 2: TIMES must have exactly one child. When visit a TIMES node with a different number of children issue a warning.
- 3: TIMES must have exactly one child. When visit a TIMES node with a different number of children issue an error.

Observe the change in behavior according to the level of SEVERITY:

```
nereida:~/src/perl/YappWithDefaultAction/examples> numchildren.pl 0 '0*2'
NUM(TERMINAL[0])
nereida:~/src/perl/YappWithDefaultAction/examples> numchildren.pl 1 '0*2'
TIMES(NUM(TERMINAL[0]),NUM(TERMINAL[2]))
nereida:~/src/perl/YappWithDefaultAction/examples> numchildren.pl 2 '0*2'
Warning! found node TIMES with 2 children.
Expected 1 children (see line 16 of numchildren.pl)"
TIMES(NUM(TERMINAL[0]),NUM(TERMINAL[2]))
nereida:~/src/perl/YappWithDefaultAction/examples> numchildren.pl 3 '0*2'
Error! found node TIMES with 2 children.
Expected 1 children (see line 16 of numchildren.pl)"
 at (eval 2) line 29
```

10.6 Array Treeregexp Expressions

The Treeregexp language permits expressions like:

```
A(@a,B($x),@c)
```

After the matching variable `@A` contains the shortest prefix of `$A->children` that does not match `B($x)`. The variable `@c` contains the remaining sufix of `$A->children`.

The following example uses array treereg expressions to move an assignment out of loop (to be correct, we have to guarantee that the assignment is an invariant of the loop). See lines 98-111:

```
nereida:~/src/perl/YappWithDefaultAction/examples> \
      cat -n moveinvariantoutofloopcomplexformula.pl
  1  #!/usr/bin/perl -w
  2  use strict;
  3  use Parse::Eyapp;
  4  use Parse::Eyapp::Treeregexp;
  5
  6  my $grammar = q{
 ..      ........... # as usual, but including WHILE loops
 80  }; # end grammar
 ..  ..........................................................................
 93  my $program = "a =1000; c = 1; while (a) { c = c*a; b = 5; a = a-1 }\n";
 94  $parser->YYData->{INPUT} = $program;
 95  my $t = $parser->Run;
 96  my @output = split /\n/, $t->str;
 97
 98  my $p = Parse::Eyapp::Treeregexp->new( STRING => q{
 99    moveinvariant: BLOCK(
100                     @prests,
101                     WHILE(VAR($b), BLOCK(@a, ASSIGN($x, NUM($e)), @c)),
102                     @possts
103                   )
104      => {
105           my $assign = $ASSIGN;
106           $BLOCK[1]->delete($ASSIGN);
107           $BLOCK[0]->insert_before($WHILE, $assign);
108         }
109    },
110    FIRSTLINE => 99,
111  );
112  $p->generate();
113  $moveinvariant->s($t);
114  my @output2 = split /\n/, $t->str;
115
116  my ($node1, $node2);
117  format STDOUT_TOP =
118                          PROGRAM
119  -------------------------------------------------------
120  @||||||||||||||||||||||||||||||||||||||||||||||||||||||
121  $program
122  -------------------------------------------------------
123  Before                     |    After
124  ---------------------------|---------------------------
125  .
126
127  format STDOUT =
128  @<<<<<<<<<<<<<<<<<<<<<<<<<<@|@<<<<<<<<<<<<<<<<<<<<<<<<<
129  $node1,                    '|',$node2
130  .
131
132  for (1..$#output) {
```

```
133    $node1 = $output[$_];
134    $node2 = $output2[$_];
135    write;
136  }
```

The call to the method `delete` at line 106 deletes the `ASSIGN` child of the second `BLOCK`. The copy saved in `$assign` is inserted as a child of the first block before the loop. Here is the output:

```
nereida:~/src/perl/YappWithDefaultAction/examples> \
      moveinvariantoutofloopcomplexformula.pl | cat -n
  1                          PROGRAM
  2  -------------------------------------------------------
  3   a =1000; c = 1; while (a) { c = c*a; b = 5; a = a-1 }
  4  -------------------------------------------------------
  5  Before                      |    After
  6  ----------------------------|--------------------------
  7  BLOCK(                      | BLOCK(
  8    ASSIGN(                   |   ASSIGN(
  9      TERMINAL[a],            |     TERMINAL[a],
 10      NUM(                    |     NUM(
 11        TERMINAL[1000]        |       TERMINAL[1000]
 12      )                       |     )
 13    ) # ASSIGN,               |   ) # ASSIGN,
 14    ASSIGN(                   |   ASSIGN(
 15      TERMINAL[c],            |     TERMINAL[c],
 16      NUM(                    |     NUM(
 17        TERMINAL[1]           |       TERMINAL[1]
 18      )                       |     )
 19    ) # ASSIGN,               |   ) # ASSIGN,
 20    WHILE(                    |   ASSIGN(
 21      VAR(                    |     TERMINAL[b],
 22        TERMINAL[a]           |     NUM(
 23      ),                      |       TERMINAL[5]
 24      BLOCK(                  |     )
 25        ASSIGN(               |   ) # ASSIGN,
 26          TERMINAL[c],        |   WHILE(
 27          TIMES(              |     VAR(
 28            VAR(              |       TERMINAL[a]
 29              TERMINAL[c]     |     ),
 30            ),                |     BLOCK(
 31            VAR(              |       ASSIGN(
 32              TERMINAL[a]     |         TERMINAL[c],
 33            )                 |         TIMES(
 34          ) # TIMES           |           VAR(
 35        ) # ASSIGN,           |             TERMINAL[c]
 36        ASSIGN(               |           ),
 37          TERMINAL[b],        |           VAR(
 38          NUM(                |             TERMINAL[a]
 39            TERMINAL[5]       |           )
 40          )                   |         ) # TIMES
 41        ) # ASSIGN,           |       ) # ASSIGN,
 42        ASSIGN(               |       ASSIGN(
 43          TERMINAL[a],        |         TERMINAL[a],
 44          MINUS(              |         MINUS(
 45            VAR(              |           VAR(
 46              TERMINAL[a]     |             TERMINAL[a]
 47            ),                |           ),
 48            NUM(              |           NUM(
 49              TERMINAL[1]     |             TERMINAL[1]
 50            )                 |           )
 51          ) # MINUS           |         ) # MINUS
```

```
        ) # ASSIGN              |       ) # ASSIGN
      ) # BLOCK                 |     ) # BLOCK
    ) # WHILE                   |   ) # WHILE
  ) # BLOCK                     | ) # BLOCK
```

11 Translation Schemes

Eyapp allows through the `%metatree` directive the creation of *Translation Schemes* as described in the Dragon's book. Instead of executing the semantic actions associated with the productions, the syntax tree is built. Semantic actions aren't executed. Instead they are inserted as nodes of the syntax tree. The main difference with ordinary nodes being that the attribute of such a `CODE` node is a reference to the anonymous subroutine representing the semantic action. The tree is later traversed in depth-first order using the `$t->translation_scheme` method: each time a `CODE` node is visited the action is executed.

The following example parses a tiny subset of a typical *typed language* and decorates the syntax tree with a new attribute `t` holding the type of each declared variable:

```
use strict; # File examples/trans_scheme_simple_decls4.pl
use Data::Dumper;
use Parse::Eyapp;
our %s; # symbol table

my $ts = q{
  %token FLOAT INTEGER NAME

  %{
  our %s;
  %}

  %metatree

  %%
  Dl:  D <* ';'>
  ;

  D : $T { $L->{t} = $T->{t} } $L
  ;

  T : FLOAT    { $lhs->{t} = "FLOAT" }
    | INTEGER  { $lhs->{t} = "INTEGER" }
  ;

  L : $NAME
        { $NAME->{t} = $lhs->{t}; $s{$NAME->{attr}} = $NAME }
    | $NAME { $NAME->{t} = $lhs->{t}; $L->{t} = $lhs->{t} } ',' $L
        { $s{$NAME->{attr}} = $NAME }
  ;
  %%
}; # end $ts

sub Error { die "Error sintáctico\n"; }

{ # Closure of $input, %reserved_words and $validchars
  my $input = "";
  my %reserved_words = ();
  my $validchars = "";

  sub parametrize__scanner {
    $input = shift;
    %reserved_words = %{shift()};
    $validchars = shift;
  }
```

```
  sub scanner {
    $input =~ m{\G\s+}gc;                        # skip whites
    if ($input =~ m{\G([a-z_A_Z]\w*)\b}gc) {
      my $w = uc($1);                    # upper case the word
      return ($w, $w) if exists $reserved_words{$w};
      return ('NAME', $1);               # not a reserved word
    }
    return ($1, $1) if ($input =~ m/\G([$validchars])/gc);
    die "Not valid token: $1\n" if ($input =~ m/\G(\S)/gc);
    return ('', undef); # end of file
  }
} # end closure

Parse::Eyapp->new_grammar(input=>$ts,classname=>'main',outputfile=>'Types.pm');
my $parser = main->new(yylex => \&scanner, yyerror => \&Error);

parametrize__scanner(
  "float x,y;\ninteger a,b\n",
  { INTEGER => 'INTEGER', FLOAT => 'FLOAT'},
  ",;"
);

my $t = $parser->YYParse() or die "Syntax Error analyzing input";

$t->translation_scheme;

$Data::Dumper::Indent = 1;
$Data::Dumper::Terse = 1;
$Data::Dumper::Deepcopy  = 1;
$Data::Dumper::Deparse = 1;
print Dumper($t);
print Dumper(\%s);
```

Inside a Translation Scheme the lexical variable `$lhs` refers to the attribute of the father.

11.1 Execution Stages of a Translation Scheme

The execution of a Translation Scheme can be divided in the following stages:

1. During the first stage the grammar is analyzed and the parser is built:

   ```
   Parse::Eyapp->new_grammar(input=>$ts,classname=>'main',outputfile=>'Types.pm');
   ```

   This stage is called *Class Construction Time*

2. A parser conforming to the generated grammar is built

   ```
   my $parser = main->new(yylex => \&scanner, yyerror => \&Error);
   ```

   This stage is called *Parser Construction Time*

3. The next phase is *Tree construction time*. The input is set and the tree is built:

   ```
   parametrize__scanner(
      "float x,y;\ninteger a,b\n",
      { INTEGER => 'INTEGER', FLOAT => 'FLOAT'},
      ",;"
    );

    my $t = $parser->YYParse() or die "Syntax Error analyzing input";
   ```

4. The last stage is *Execution Time*. The tree is traversed in depth first order and the `CODE` nodes are executed.

```
$t->translation_scheme;
```

This combination of bottom-up parsing with depth first traversin leads to a semantic behavior similar to LL and top-down parsers but with several differences:

- The grammar can be left-recursive
- At the time of executing the action the syntax tree is already built, therefore we can refer to nodes on the right side of the action like in:

```
D : $T { $L->{t} = $T->{t} } $L
```

11.2 The `%begin` directive

The `%begin { code }` directive can be used when building a translation scheme, i.e. when under the control of the `%metatree` directive. It indicates that such `code` will be executed at tree construction time. Therefore the code receives as arguments the references to the nodes of the branch than is being built. Usually the code assist in the construction of the tree. Line 39 of the following code shows an example. The action `{ $exp }` simplifies the syntax tree bypassing the parenthesis node. The example also illustrates the combined use of default actions and translation schemes.

```
nereida:~/src/perl/YappWithDefaultAction/examples> \
               cat -n trans_scheme_default_action.pl
  1  #!/usr/bin/perl -w
  2  use strict;
  3  use Data::Dumper;
  4  use Parse::Eyapp;
  5  use IO::Interactive qw(is_interactive);
  6
  7  my $translationscheme = q{
  8  %{
  9  # head code is available at tree construction time
 10  use Data::Dumper;
 11  our %sym; # symbol table
 12  %}
 13
 14  %defaultaction { $lhs->{n} = eval " $left->{n} $_[2]->{attr} $right->{n} " }
 15
 16  %metatree
 17
 18  %right   '='
 19  %left   '-' '+'
 20  %left   '*' '/'
 21
 22  %%
 23  line:       %name EXP
 24                exp <+ ';'> /* Expressions separated by semicolons */
 25                  { $lhs->{n} = $_[1]->Last_child->{n} }
 26  ;
 27
 28  exp:
 29              %name PLUS
 30                exp.left '+' exp.right
 31          |   %name MINUS
 32                exp.left '-' exp.right
 33          |   %name TIMES
 34                exp.left '*' exp.right
 35          |   %name DIV
 36                exp.left '/' exp.right
 37          |   %name NUM   $NUM
 38                  { $lhs->{n} = $NUM->{attr} }
 39          |   '(' $exp ')'  %begin { $exp }
```

```
        |   %name VAR
              $VAR
                { $lhs->{n} = $sym{$VAR->{attr}}->{n} }
        |   %name ASSIGN
              $VAR ’=’ $exp
                { $lhs->{n} = $sym{$VAR->{attr}}->{n} = $exp->{n} }

;

%%
# tail code is available at tree construction time
sub _Error {
  die "Syntax error.\n";
}

sub _Lexer {
    my($parser)=shift;

    for ($parser->YYData->{INPUT}) {
        defined($_) or  return(’’,undef);

        s/^\s*//;
        s/^([0-9]+(?:\.[0-9]+)?)// and return(’NUM’,$1);
        s/^([A-Za-z][A-Za-z0-9_]*)// and return(’VAR’,$1);
        s/^(.)// and return($1,$1);
        s/^\s*//;
    }
}

sub Run {
    my($self)=shift;
    return $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error );
}
}; # end translation scheme

$Data::Dumper::Indent = 1;
$Data::Dumper::Terse = 1;
$Data::Dumper::Deepcopy  = 1;
my $p = Parse::Eyapp->new_grammar(
  input=>$translationscheme,
  classname=>’main’,
  firstline => 6,
  outputfile => ’main.pm’);
die $p->qtables() if $p->Warnings;
my $parser = main->new();
print "Write a sequence of arithmetic expressions: " if is_interactive();
$parser->YYData->{INPUT} = <>;
my $t = $parser->Run() or die "Syntax Error analyzing input";
$t->translation_scheme;
my $treestring = Dumper($t);
our %sym;
my $symboltable = Dumper(\%sym);
print <<"EOR";
***********Tree*************
$treestring
******Symbol table**********
$symboltable
************Result**********
$t->{n}

```

```
EOR
```

12 Scope Analysis with `Parse::Eyapp::Scope`

`Parse::Eyapp` provides support for *Scope Analysis* through the module `Parse::Eyapp::Scope`. *Scope Analysis* solves the problem of *matching* each instance or use of an object in the source text with the definition that applies to such instance. Since it is a *matching* problem it can sometimes easily solved using `m` as it was explained in section Matching Trees.

The following pieces of code show how to implement scope analysis for a C-like language using `Parse::Eyapp::Scope`

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> \
                          sed -n -e '131,149p' Types.eyp | cat -n
 1  sub reset_file_scope_vars {
 2    %st = (); # reset symbol table
 3    ($tokenbegin, $tokenend) = (1, 1);
 4    %type = ( INT  => Parse::Eyapp::Node->hnew('INT'),  # like new but
 5              CHAR => Parse::Eyapp::Node->hnew('CHAR'), # creates a DAG
 6              VOID => Parse::Eyapp::Node->hnew('VOID'),
 7            );
 8    $depth = 0;
 9    $ids = Parse::Eyapp::Scope->new(
10             SCOPE_NAME => 'block',
11             ENTRY_NAME => 'info',
12             SCOPE_DEPTH => 'depth',
13    );
14    $loops = Parse::Eyapp::Scope->new(
15             SCOPE_NAME => 'exits',
16    );
17    $ids->begin_scope();
18    $loops->begin_scope();
19  }
```

Of course you have to include a directive

```
use Parse::Eyapp::Scope
```

in your client program.

The calls to `Parse::Eyapp::Scope->new` method (lines 9-13 and 14-16 in the code above) create two *Scope Manager* objects. One scope manager to solve the scope problem for variables (`$ids`) and another to solve the scope problem for loops (`$loops`). The scope problem for loops consists in matching each instance of a `BREAK` or `CONTINUE` with the enclosing loop. The beginning of a scope is set by calling to the `begin_scope` method (lines 17 and 18). The end of a scope is signalled by a call to the method `end_scope`. Of course, sub `reset_file_scope_vars` must be executed at the proper time:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> \
                            sed -n -e '170,203p' Types.eyp | cat -n
program: /* program -> definition +  */
      {
        reset_file_scope_vars();
      }
    definition<%name PROGRAM +>.program
      {
        $program->{symboltable} = { %st };  # creates a copy of the s.t.
        $program->{depth} = 0;
        $program->{line}  = 1;
        $program->{types} = { %type };
        $program->{lines} = $tokenend;

        my ($nondec, $declared) = $ids->end_scope($program->{symboltable}, $program, 'type');

        # Type checking: add a direct pointer to the data-structure
```

```
16          # describing the type
17          $_->{t} = $type{$_->{type}} for @$declared;
18
19          if (@$nondec) {
20            warn "Identifier ".$_->key." not declared at line ".$_->line."\n" for @$nondec;
21            die "\n";
22          }
23
24          my $out_of_loops = $loops->end_scope($program);
25          if (@$out_of_loops) {
26            warn "Error: ".ref($_)." outside of loop at line $_->{line}\n" for @$out_of_loops;
27            die "\n";
28          }
29
30          # Check that are not dangling breaks
31          reset_file_scope_vars();
32
33          $program;
34        }
```

Observe the different ways of calling `end_scope` (lines 13 and 24). When a hash table is provided as first argument the declared symbols will be automatically inserted in it. In such case the classes of the nodes being inserted must have a `key` method that computes the key for such node.

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> \
                           sed -n -e '651,657p' Types.eyp | cat -n
sub VAR::key {
  my $self = shift;

  return $self->child(0)->{attr}[0];
}

*VARARRAY::key = *FUNCTIONCALL::key = \&VAR::key;
```

Each instance of an *scoped object* must be declared as belonging to the current scope using the `scope_instance` method. The following is an example for the `$loops` scope manager object:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> \
                           sed -n -e '335,346p' Types.eyp | cat -n
statement:
    expression ';' { $_[1] }
  | ';'
  | %name BREAK
    $BREAK ';'
      {
        my $self = shift;
        my $node = $self->YYBuildAST(@_);
        $node->{line} = $BREAK->[1];
        $loops->scope_instance($node);
        return $node;
      }
```

and the following illustrates the same for the `$ids` scope manager:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> \
                            sed -n -e '410,425p' Types.eyp | cat -n
Primary:
    %name INUM
    INUM
  | %name CHARCONSTANT
    CHARCONSTANT
  | $Variable
```

```
        {
          $ids->scope_instance($Variable);
          return $Variable
        }
    | '(' expression ')' { $_[2] }
    | $function_call
        {
          $ids->scope_instance($function_call);
          return $function_call  # bypass
        }
```

Of course, in each place where a new scope begins/ends the corresponding calls to `begin_scope` and `end_scope` must be issued. See the following code:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> \
                            sed -n -e '277,302p' Types.eyp | cat -n
block:  /* Production is: block -> '{' declaration * statement * '}' */
    '{'.bracket
      { $ids->begin_scope(); }
     declaration<%name DECLARATIONS *>.decs statement<%name STATEMENTS *>.sts '}'
      {
        my %st;

        for my $lst ($decs->children) {

            # control duplicated declarations
          my $message;
          die $message if $message = is_duplicated(\%st, $lst);

          %st = (%st, %$lst);
        }
        $sts->{symboltable} = \%st;
        $sts->{line} = $bracket->[1];
        $sts->type("BLOCK") if (%st);
        my ($nondec, $dec) = $ids->end_scope(\%st, $sts, 'type');

        # Type checking: add a direct pointer to the data-structure
        # describing the type
        $_->{t} = $type{$_->{type}} for @$dec;

        return $sts;
      }
```

13 SEE ALSO

- perldoc *Parse::Eyapp*
- The `Eyapp.pdf` and `eyapptut.pdf` files accompanying this distribution
- perldoc *eyapp*,
- perldoc *treereg*,
- *Análisis Léxico y Sintáctico*, (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*.

14 REFERENCES

- The classic book "Compilers: Principles, Techniques, and Tools" by Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman (Addison-Wesley 1986)

15 AUTHOR

Casiano Rodriguez-Leon (casiano@ull.es)

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

16 ACKNOWLEDGMENTS

This work has been supported by CEE (FEDER) and the Spanish Ministry of *Educación 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.

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.

17 LICENCE AND COPYRIGHT

Index