Parse::Eyapp Reference Manual

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

Parse::Eyapp (Extended yapp) is a collection of modules that extends Francois Desarmenien Parse::Yapp 1.05. Eyapp extends yacc/yapp syntax with functionalities line named attributes, EBNF-like expressiones, modifiable default action, automatic syntax tree building, semi-automatic abstract syntax tree building, translation schemes, tree regular expressions, tree transformations, scope analysis, support, directed acyclic graphs and a few more.

This is a reference manual. To read an introduction to `Parse::Eyapp` *see* `Parse::eyapptut`.

5 The Eyapp Language

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

```
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)?
```

Follows the grammar accepted by `eyapp` using its own notation. 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:  decls ?
decls:  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:   rulesec ? '%%'
rulesec:  startrules rules *
startrules:  IDENT ':'  rhss ';'
rules:       IDENT ':' rhss ';'
rhss: rule <+ '|'>
rule:   optname rhs prec epscode
    |   optname rhs
rhs:  rhselts ?
rhselts:  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. For an introduction to the extensions see *Parse::Eyapp::eyapptut*.

6 The Treeregexp Language

A Treeregexp program is made of the repetition of three kind of tree primitives: The treeregexp transformations, auxiliar Perl code and Transformation Families.

```
treeregexplist:  treeregexp*

treeregexp:
    IDENT ':' treereg ('=>' CODE)?  # Treeregexp
  | CODE                            # Auxiliar code
  | IDENT '=' IDENT + ';'           # Transformation families
```

Treeregexp themselves follow the rule:

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

Several instances of this rule can be seen in the example in the SYNOPSIS section. The identifier IDENT gives the name to the rule. At the time of this writing (2006) there are the following kinds of treeregexes:

```
treereg:
      /* tree patterns with children */
    IDENT '(' childlist ')' ('and' CODE)?
  | REGEXP (':' IDENT)? '(' childlist ')' ('and' CODE)?
  | SCALAR '(' childlist ')' ('and' CODE)?
  | '.' '(' childlist ')' ('and' CODE)?
        /* leaf tree patterns */
  | IDENT ('and' CODE)?
  | REGEXP (':' IDENT)? ('and' CODE)?
  | '.' ('and' CODE)?
  | SCALAR ('and' CODE)?
  | ARRAY
  | '*'
```

6.1 Treeregexp rules

When seen a rule like

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

The Treeregexp translator creates a Parse::Eyapp:YATW object that can be later referenced in the user code by the package variable $zero_times.

6.1.1 The treeregexp

The first part of the rule TIMES(NUM($x), ., .) indicates that for a matching to succeed the node being visited must be of type TIMES, have a left child of type NUM and two more children.

If the first part succeeded then the following part takes the control to see if the semantic conditions are satisfied.

6.1.2 Semantic condition

The second part is optional and must be prefixed by the reserved word and followed by a Perl code manifesting the semantic conditions that must be hold by the node to succeed.

6.1.3 Referencing the matching nodes

The node being visited can be referenced/modified inside the semantic actions using `$_[0]`.

The Treeregexp translator automatically creates a set of lexical variables for us. The scope of these variables is limited to the semantic condition and the transformation code.

Thus, the node being visited `$_[0]` can be also referenced using the lexical variable `$TIMES` which is created by he Treeregexp compiler. In the same way a reference to the left child `NUM` will be stored in the lexical variable `$NUM`and a reference to the child of `$NUM` will be stored in `$x`.

In the example the condition states that the attribute of the node associated with `$x` must be zero.

When the same type of node appears several times inside the treeregexp part the associated lexical variable is declared by the Treeregexp compiler as an array. This is the case in the `constantfold` transformation example, where there are two nodes of type `NUM`:

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

Thus variable `$NUM[0]` references the first node and `$NUM[1]` the second.

6.1.4 Transformation code

The third part of the rule is also optional and comes prefixed by the big arrow =>. The Perl code in this section usually transforms the matching tree. To achieve the modification of the tree, the Treeregexp programmer **must use** `$_[0]` and not the lexical variables provided by the translator. Remember that in Perl `$_[0]` is an alias of the actual parameter. The `constantfold` example above **will not work** if we rewrite the code `$_[0] = $NUM[0]` as

```
{ $TIMES = $NUM }
```

6.2 Regexp Treeregexes

The previous `constantfold` example used a classic Perl linear regexp to explicit that we want the root node to match the Perl regexp. The general syntax for `REGEXP` treeregexes patters is:

```
treereg: REGEXP (':' IDENT)? '(' childlist ')' ('and' CODE)?
```

The `REGEXP` must be specified between slashes (other delimiters as `{}` are not accepted). It is legal to specify options after the second slash (like `e`, `i`, etc.).

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 optional identifier after the `REGEXP` indicates the name of the lexical variable that will be held a reference to the node whose type matches `REGEXP`. Variables `$W` and `@W` (if there are more than one REGEXP and or dot treeregexes) will be used instead if no identifier is specified.

6.3 Scalar Treeregexes

A scalar treeregxp is defined writing a Perl scalar inside the treeregexp, like `$x` in `NUM($x)`. A scalar treeregxp immediately matches any node that exists and stores a reference to such node inside the Perl lexical scalar variable. The scope of the variable is limited to the semantic parts of the Treeregexp. Is illegal to use `$W` or `$W_#num` as variable names for scalar treeregexes.

6.4 Dot Treeregexes

A dot matches any node. It can be seen as an abbreviation for scalar treeregexes. The reference to the matching node is stored in the lexical variable `$W`. The variable `@W` will be used instead if there are more than one REGEXP and or dot treeregexes

6.5 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`.

6.6 Star Treeregexp

Deprecated. Don't use it. Is still there but not to endure.

6.7 Transformation Families

Transformations created by `Parse::Eyapp::Treeregexp` can be grouped in families. That is the function of the rule:

```
treeregexp: IDENT '=' IDENT + ';'
```

The next example (file `examples/TSwithtreetransformations3.eyp`) defines the family

```
algebraic_transformations = constantfold zero_times times_zero comasocfold;
```

Follows the code:

```
my $transform = Parse::Eyapp::Treeregexp->new( STRING => q{

  uminus: UMINUS(., NUM($x), .) => { $x->{attr} = -$x->{attr}; $_[0] = $NUM }
  constantfold: /TIMES|PLUS|DIV|MINUS/:bin(NUM($z), ., NUM($y))
     => {
    $z->{attr} = eval  "$z->{attr} $W->{attr} $y->{attr}";
    $_[0] = $NUM[0];
  }
  commutative_add: PLUS($x, ., $y, .)
    => { my $t = $x; $_[0]->child(0, $y); $_[0]->child(2, $t)}
  comasocfold: TIMES(DIV(NUM($x), ., $b), ., NUM($y))
     => {
    $x->{attr} = $x->{attr} * $y->{attr};
    $_[0] = $DIV;
  }
  zero_times: TIMES(NUM($x), ., .) and { $x->{attr} == 0 } => { $_[0] = $NUM }
  times_zero: TIMES(., ., NUM($x)) and { $x->{attr} == 0 } => { $_[0] = $NUM }
  algebraic_transformations = constantfold zero_times times_zero comasocfold;
},
);

$transform->generate();
our ($uminus);
$uminus->s($tree);
```

The transformations belonging to a family are usually applied toghether:

```
$tree->s(@algebraic_transformations);
```

6.8 Code Support

In between Treeregexp rules and family assignments the programmer can insert Perl code between curly brackets. That code usually gives support to the semantic conditions and transformations inside the rules. See for example test 14 in the `t/` directory of the Parse::Eyapp distribution.

```
{
  sub not_semantic {
    my $self = shift;
    return  1 if $self->{token} eq $self->{attr};
    return 0;
  }
}

delete_tokens : TERMINAL and { not_semantic($TERMINAL) }
                         => { $delete_tokens->delete() }
```

7 Parse::Eyapp Methods

A `Parse::Eyapp` object holds the information about the `Eyapp` input grammar: parsing tables, conflicts, semantic actions, etc.

7.1 Parse::Eyapp->new_grammar

To translate an Eyapp grammar you must use either the *eyapp* script or call the class constructor `new_grammar`. The `Parse::Eyapp` method `Parse::Eyapp->new_grammar(input=>$grammar)` creates a package containing the code that implements a LALR parser for the input grammar:

```
my $p = Parse::Eyapp->new_grammar(
  input=>$translationscheme,
  classname=>'Grammar',
);
die $p->Warnings if $p->Warnings;
my $new_parser_for_grammar = Grammar->new();
```

The method returns a `Parse::Eyapp` object. A `Parse::Eyapp` object describes the grammar and the parsing tables.

You can check the object to see if there were problems during the construction of the parser for your grammar:

```
            die $p->qtables() if $p->Warnings;
```

The call to `Parse::Eyapp->new_grammar` generates a class/package containing the parser for your input grammar. Such package lives in the namespace determined by the `classname` argument of `new_grammar`. The method `Warnings` returns the warnings produced during the parsing. The absence of warnings indicates the correctness of the input program.

- input

The string containing the input

- classname

The name of the package that will held the code for the LALR parser. The package of the caller will be used as default if none is specified.

- firstline

For error diagnostics. The line where the definition of the Eyapp grammar starts.

- linenumbers

Include/not include `# line directives` in the generated code

```
my $p = Parse::Eyapp->new_grammar(
  input=>$translationscheme,
  classname=>'main',
  firstline => 6,
  outputfile => 'main');
die $p->Warnings if $p->Warnings;
```

- outputfile

If defined the generated code fill be dumped in the specified filename (with extension .pm) and the LALR information ambigueties and conflicts) in the specified filename with extension .output.

7.2 $eyapp->qtables

Returns a string containing information on warnings, ambiguities, conflicts, rules and the generated DFA tables. Is the same information in `file.output` when using the command `eyapp -v file.eyp`.

```
  my $p = Parse::Eyapp->new_grammar(
    input=>$eyappprogram,
    classname=>'SimpleC',
    outputfile => 'SimpleC.pm',
    firstline=>12,
  );

  print $p->qtables() if $p->Warnings;
```

7.3 $eyapp->outputtables

It receives two arguments

```
  $eyapp->outputtables($path, $base)
```

Similar to `qtables` but prints the information on warnings, conflicts and rules to the specified `$path/$file`.

7.4 $eyapp->Warnings

Returns the warnings resulting from compiling the grammar:

```
  my $p = Parse::Eyapp->new_grammar(
    input=>$translationscheme,
    classname=>'main',
    firstline => 6,
    outputfile => 'main'
  );
  die $p->Warnings if $p->Warnings;
```

Returns the empty string if there were no conflicts.

7.5 $eyapp->ShowDfa

Returns a string with the information about the LALR generated DFA. Returns the empty string if there were no conflicts.

7.6 $eyapp->Summary

Returns a string with summary information about the compilation of the grammar. No arguments.

7.7 $eyapp->Conflicts

Returns a string with summary information about the conflicts that arised when compiling the grammar. No arguments.

7.8 $eyapp->DfaTable

Returns a string with the parsing tables

8 Methods Available in the Generated `Class`

This section describes the methods and objects belonging to the class generated either using *eyapp* or `Parse::Eyapp->new_gr`
In the incoming paragraphs we will assume that `Class` was the value selected for the `classname` argument when `Parse::Eyapp->new_grammar` was called.

Objects belonging to `Class` are the actual parsers for the input grammar.

8.1 Class->new

The method `Class->new` returns a new LALR parser object. Here `Class` stands for the name of the class containing the parser. See an example of call:

```
my $parser = main->new(yyprefix => 'Parse::Eyapp::Node::',
                       yylex    => \&main::_Lexer,
                       yyerror  => \&main::_Error,
                       yydebug => 0x1F,
);
```

The meaning of the arguments used in the example are as follows:

- yyprefix

Used with %tree or %metatree. When used, the type names of the nodes of the syntax tree will be build prefixing the value associated to `yyprefix` to the name of the production rule. The name of the production rule is either explicitly given through a %name directive or the concatenation of the left hand side of the rule with the ordinal of the right hand side of the production.

See files Rule9.yp, Transform4.trg and foldand0rule9_4.pl in the examples directory for a more detailed example. File Rule9.yp is very much like the grammar in the SYNOPSIS example but it makes use of the directive `semantic tokens`:

%semantic token '=' '-' '+' '*' '/'

You need to compile the grammar Rule9.yp and the treeregexp Transform4.trg using the commands:

```
eyapp Rule9
treereg -p 'Rule9::' Transform4.trg
```

and the parser must be created inside the client program using the option `yyprefix`:

```
use Rule9;
use Transform4;

my $parser = new Rule9(yyprefix => "Rule9::");
my $t=$parser->YYParse(yylex=>\&Rule9::Lexer,yyerror=>\&Rule9::Error,);
$t->s(@Transform4::all);
```

Now the trees types are prefixed with "Rule9::". After giving as input `a=2*3` to `foldand0rule9_4.pl` we will finally have a transformed tree like this:

```
bless( {
  'children' => [
    bless({'children'=>[],'attr'=>'a','token'=>'VAR'},'Rule9::TERMINAL'),
    bless({'children'=>[],'attr'=>'=','token'=>'='},'Rule9::TERMINAL'),
    bless( {
      'children' => [
        bless({'children'=>[],'attr'=>6,'token'=>'NUM'},'Rule9::TERMINAL')
      ]
    }, 'Rule9::NUM' )
  ]
}, 'Rule9::ASSIGN' );
```

- yylex

Reference to the lexer subroutine

- yyerror

Reference to the error subroutine. The error subroutine receives as first argument the reference to the `Class` parser object. This way it can take advantage of methods like `YYCurval` and $parser-YYExpect> (see below):

```
sub _Error {
  my($token)=$_[0]->YYCurval;
  my($what)= $token ? "input: '$token'" : "end of input";
  my @expected = $_[0]->YYExpect();

  local $" = ', ';
  die "Syntax error near $what. Expected one of these tokens: @expected\n";
}
```

- yydebug

Controls the level of debugging. It works as follows:

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

As an example consider the grammar (file PlusList1.yp in examples/)

```
%%
S:      'c'+  { print "S -> 'c'+\n" }
;
%%
```

When the parser is called with `yyedebug` activated:

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

the output reports about the parser activities:

```
> use_pluslist1.pl
----------------------------------------
In state 0:
Stack:[0]
c
Need token. Got >c<
Shift and go to state 2.
----------------------------------------
In state 2:
Stack:[0,2]
Don't need token.
Reduce using rule 2 (PLUS-1,1): Back to state 0, then go to state 3.
----------------------------------------
In state 3:
Stack:[0,3]
Need token. Got ><
Reduce using rule 3 (S,1): S -> 'c'+
Back to state 0, then go to state 1.
----------------------------------------
In state 1:
```

```
Stack:[0,1]
Shift and go to state 4.
----------------------------------------
In state 4:
Stack:[0,1,4]
Don’t need token.
Accept.
```

The package produced from the grammar has several methods.

The parser object has the following methods that work at parsing time exactly as in *Parse::Yapp*. These methods can be found in the module Parse::Eyapp::Driver. Assume you have in `$parser` the reference to your parser object:

8.2 $parser->YYParse()

It very much works Parse::Yapp::YYParse and as yacc/bison yyparse. It accepts almost the same arguments as `Class-new>` with the exception of `yyprefix` which can be used only with `new`.

8.3 $parser->YYErrok

Works as yacc/bison yyerrok. Modifies the error status so that subsequent error messages will be emitted.

8.4 $parser->YYError

Works as yacc/bison YYERROR. Pretends that a syntax error has been detected.

8.5 $parser->YYNberr

The current number of errors

8.6 $parser->YYAccept

Works as yacc/bison YYACCEPT. The parser finishes returning the current semantic value to indicate success.

8.7 $parser->YYAbort

Works as yacc/bison YYABORT. The parser finishes returning `undef` to indicate failure.

8.8 $parser->YYRecovering

Works as yacc/bison YYRECOVERING. Returns TRUE if the parser is recovering from a syntax error.

8.9 $parser->YYCurtok

Gives the current token

8.10 $parser->YYCurval

Gives the attribute associated with the current token

8.11 $parser->YYExpect

Is a list which contains the tokens the parser expected when the failure occurred

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ \
                              sed -ne ’26,33p’ Postfix.eyp
sub _Error {
  my($token)=$_[0]->YYCurval;
  my($what)= $token ? "input: ’$token’" : "end of input";
  my @expected = $_[0]->YYExpect();

  local $" = ’, ’;
  die "Syntax error near $what. Expected one of these tokens: @expected\n";
}
```

8.12 $parser->YYLexer

Returns a reference to the lexical analyzer

8.13 $parser->YYLhs

Returns the identifier of the left hand side of the current production (the one that is being used for reduction/antiderivation. An example of use can be found in examples/Lhs1.yp:

```
%defaultaction { print $_[0]->YYLhs,"\n" }
```

8.14 $parser->YYRuleindex

Returns the index of the production rule, counting the super rule as rule 0. To know the numbers have a look at the `.output` file. To get a `.output` file use the option `-v` of `eyapp` or the `outputfile` parameter when using method `new_grammar`.

8.15 $parser->YYRightside

Returns an array of strings describing the right hand side of the rule

8.16 $parser->YYIsterm

Returns TRUE if the symbol given as argument is a terminal. Example:

```
  DB<0> x $self->YYIsterm('exp')
 0  ''
  DB<1> x $self->YYIsterm('*')
 0  1
```

An example of combined use of `YYRightside`, `YYRuleindex`, `YYLhs` and `YYIsterm` can be found `examples/Rule3.yp`:

```
sub build_node {
  my $self = shift;
  my @children = @_;
  my @right = $self->YYRightside();
  my $var = $self->YYLhs;
  my $rule = $self->YYRuleindex();

  for(my $i = 0; $i < @right; $i++) {
    $_ = $right[$i];
    if ($self->YYIsterm($_)) {
      $children[$i] = bless { token => $_, attr => $children[$i] },
                                            __PACKAGE__.'::TERMINAL';
    }
  }
  bless {
          children => \@children,
          info => "$var -> @right"
        }, __PACKAGE__."::${var}_$rule"
}
```

8.17 $parser->YYIssemantic

Returns TRUE if the terminal is semantic. Semantics token can be declared using the directive `%semantic token`. The opposite of a Semantic token is a Syntactic token. Syntactic tokens can be declared using the directive `%syntactic token`.

When using the `%tree` directive all the nodes corresponding to syntactic tokens are pruned from the tree. When using the `%tree` directive string tokens (those that appear in the text of the grammar delimited by simple quotes) are considered, by default, syntactic tokens.

When using the `%metatree` directive all the tokens are considered, by default, semantic tokens. Thus, no nodes will be - by default- pruned when construction the code augmented tree. The exception are string tokens used as separators in the definition of lists, like in `S <* ';'>`. If you want the separating string token to appear include an explicit semantic declaration for it (example `%semantic token ';'`).

8.18 $parser->YYName

Returns the name of the current rule

```
  DB<12> x $self->YYName
 0  'exp_11'
```

8.19 $parser->YYPrefix

Return and/or sets the `yyprefix` attribute. This a string that will be concatenated as a prefix to any `Parse::Eyapp::Node` nodes in the syntax tree.

8.20 $parser->YYBypass

Returns TRUE if running under the `%tree bypass` clause

8.21 $parser->YYBypassrule

Returns TRUE if the production being used for reduction was marked to be bypassed.

8.22 $parser->YYFirstline

First line of the input string describing the grammar

8.23 $parser->BeANode

Receives as input a `Class` name. Introduces `Parse::Eyapp::Node` as an ancestor class of `Class`

8.24 $parser->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;
      }
```

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

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

8.25 $parser->YYBuildTS

Similar to `$parser->YYBuildAST` but for translation schemes.

9 `Parse::Eyapp::Parse` objects

The parser for the `Eyapp` language was written and generated using `Parse::Eyapp` and the `eyapp` compiler (actually the first version was bootstrapped using the *yapp* compiler). The grammar describing the `Eyapp` language is in the file `Parse/Eyapp/Parse.yp` in the `Parse::Eyapp` distribution. Therefore `Parse::Eyapp::Parse` objects have all the methods mentioned in the section "Methods Available in the Generated `Class`". A `Parse::Eyapp::Parse` is nothing but a particular kind of `Parse::Eyapp` parser: *the one that parses* `Eyapp` *grammars.*

10 Parse::Eyapp::Node Methods

The `Parse::Eyapp::Node` objects represent the nodes of the syntax tree. All the node classes build by `%tree` and `%metatree` directives inherit from `Parse::Eyapp::Node` and consequently have acces to the methods provided in such module.

10.1 Parse::Eyapp::Node->new

Nodes are usually created using the `%tree` or `%metatree` `Parse::Eyapp` directives. The `Parse::Eyapp::Node` constructor `new` offers an alternative way to create forests.

This class method can be used to build multiple nodes on a row. It receives a string describing the tree and optionally a reference to a subroutine. Such subroutine (called the attribute handler) is in charge to initialize the attributes of the just created nodes. The attribute handler is called with the array of references to the nodes as they appear in the string from left to right.

`Parse::Eyapp::Node->new` returns an array of pointers to the nodes created as they appear in the input string from left to right. In scalar context returns a pointer to the first tree.

The following example (see file `examples/28foldwithnewwithvars.pl`) of a treeregexp transformations creates a new `NUM(TERMINAL)` node using `Parse::Eyapp::Node->new`:

```
my $p = Parse::Eyapp::Treeregexp->new( STRING => q{
  {
    my %Op = (PLUS=>'+', MINUS => '-', TIMES=>'*', DIV => '/');
  }
  constantfold: /TIMES|PLUS|MINUS|DIV/(NUM($x), NUM($y))
     => {
    my $op = $Op{ref($_[0])};

    my $res = Parse::Eyapp::Node->new(
      q{NUM(TERMINAL)},
      sub {
        my ($NUM, $TERMINAL) = @_;
        $TERMINAL->{attr} = eval "$x->{attr} $op $y->{attr}";
        $TERMINAL->{token} = 'NUM';
      },
    );
    $_[0] = $res;
  }
  },
);
```

The string can describe more than one tree like in:

```
my @t = Parse::Eyapp::Node->new(
         'A(C,D) E(F)', sub { my $i = 0; $_->{n} = $i++ for @_ });
```

The following trees will be built:

```
bless( { 'n' => 0,
  'children' => [
    bless( { 'n' => 1, 'children' => [] }, 'C' ),
    bless( { 'n' => 2, 'children' => [] }, 'D' )
  ]
}, 'A' );
bless( { 'n' => 3,
  'children' => [
    bless( { 'n' => 4, 'children' => [] }, 'F' )
  ]
}, 'E' );
```

and `@t` will contain 5 references to the corresponding subtrees A(C,D), C, D, E(F) and F.

10.2 `Parse::Eyapp::Node->hnew`

`Parse::Eyapp` provides the method `Parse::Eyapp::Node->hnew` to build DAGs instead of trees. It is built using *hashed consing*. It works very much like `Parse::Eyapp::Node->new` but if one of the implied trees was previously built, `hnew` returns a reference to the existing one. See the following debugger session where several DAGs describing *type expressions* are built:

```
  DB<2> x $a = Parse::Eyapp::Node->hnew('F(X_3(A_3(A_5(INT)), CHAR, A_5(INT)),CHAR)')
 0  F=HASH(0x85f6a20)
    'children' => ARRAY(0x85e92e4)
    |- 0  X_3=HASH(0x83f55fc)
    |     'children' => ARRAY(0x83f5608)
    |     |- 0  A_3=HASH(0x85a0488)
    |     |     'children' => ARRAY(0x859fad4)
    |     |        0  A_5=HASH(0x85e5d3c)
    |     |           'children' => ARRAY(0x83f4120)
    |     |              0  INT=HASH(0x83f5200)
    |     |                 'children' => ARRAY(0x852ccb4)
    |     |                      empty array
    |     |- 1  CHAR=HASH(0x8513564)
    |     |     'children' => ARRAY(0x852cad4)
    |     |          empty array
    |     `- 2  A_5=HASH(0x85e5d3c)
    |           -> REUSED_ADDRESS
    `- 1  CHAR=HASH(0x8513564)
          -> REUSED_ADDRESS
  DB<3> x $a->str
 0  'F(X_3(A_3(A_5(INT)),CHAR,A_5(INT)),CHAR)'
```

The second occurrence of `A_5(INT)` is labelled `REUSED_ADDRESS`. The same occurs with the second instance of `CHAR`. `Parse::Eyapp::Node->hnew` can be more convenient than `new` when dealing with optimizations like *common subexpressions* or during *type checking*.

10.3 $node->type

Returns the type of the node. It can be called as a sub when `$node` is not a `Parse::Eyapp::Node` like this:

```
Parse::Eyapp::Node::type($scalar)
```

This is the case when visiting `CODE` nodes.
The following session with the debugger illustrates how it works:

```
  > perl -MParse::Eyapp::Node -de0
  DB<1> @t = Parse::Eyapp::Node->new("A(B,C)") # Creates a tree
  DB<2> x map { $_->type } @t # Get the types of the three nodes
  0  'A'
  1  'B'
  2  'C'
  DB<3> x Parse::Eyapp::Node::type(sub {})
  0  'CODE'
  DB<4> x Parse::Eyapp::Node::type("hola")
  0  'Parse::Eyapp::Node::STRING'
  DB<5> x Parse::Eyapp::Node::type({ a=> 1})
  0  'HASH'
  DB<6> x Parse::Eyapp::Node::type([ a, 1 ])
  0  'ARRAY'
```

As it is shown in the example it can be called as a subroutine with a (code/hash/array) reference or an ordinary scalar.

The words HASH, CODE, ARRAY and STRING are reserved for ordinary Perl references. Avoid naming a node with one of those words.

10.4 $node->child

Setter-getter to modify a specific child of a node. It is called like:

```
$node->child($i)
```

Returns the child with index $i. Returns `undef` if the child does not exists. It has two obligatory parameters: the node (since it is a method) and the index of the child. Sets teh new value if called

```
$node->child($i, $tree)
```

The method will croak if the obligatory parameters are not provided. Follows an example of use inside a Treereg program (see file `examples/TSwithtreetransformations2.eyp`:

```
my $transform = Parse::Eyapp::Treeregexp->new( STRING => q{
   commutative_add: PLUS($x, ., $y, .) # 1st . = '+' 2nd . = CODE
     => { my $t = $x; $_[0]->child(0, $y); $_[0]->child(2, $t)}
}
```

10.5 Child Access Through `%tree alias`

Remember that when the `Eyapp` program runs under the `%tree alias` directive The dot and dollar notations can be used to generate named getter-setters to access the children:

```
%tree bypass alias
....
%%
exp: %name PLUS
       exp.left '+' exp.right
....
%%
.... # and later
print $exp->left->str;
```

Here methods with names `left` and `right` will be created to access the corresponding children associated with the two instances of `exp` in the right hand side of the production rule.

10.6 $node->children

Returns the array of children of the node. When the tree is a translation scheme the CODE references are also included. See `examples/TSPostfix3.eyp` for an example of use inside a Translation Scheme:

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$\
                        sed -ne '31,34p' TSPostfix3.eyp
line: %name PROG
       exp <%name EXP + ';'>
         { @{$lhs->{t}} = map { $_->{t}} ($_[1]->children()); }
```

The tree in a Translation Scheme contains the references to the `CODE` implementing the semantic actions. For example, the syntax tree built by the parser for the input `a=-b*3` in `TSPostfix3.eyp` is:

```
PROG(EXP(
    ASSIGN(
      TERMINAL[a],
      TERMINAL[=],
      TIMES(
        NEG(TERMINAL[-], VAR(TERMINAL[b], CODE), CODE),
        TERMINAL[*],
        NUM(TERMINAL[3], CODE),
        CODE
      ) # TIMES,
      CODE
    ) # ASSIGN
  ) # EXP,
  CODE
) # PROG
```

`$node->children` can also be used as a setter.

10.7 $node->Children

Returns the array of children of the node. When dealing with a translation scheme, the $node->Children method (first in uppercase) returns the non CODE children of the node.

10.8 $node->last_child

Return the last child of the node. When dealing with translation schemes, the last can be a `CODE` node.

10.9 $node->Last_child

The `$node->Last_child` method returns the last non CODE child of the node. See an example:

```
line:       %name EXP
              exp <+ ';'> /* Expressions separated by semicolons */
          { $lhs->{n} = $_[1]->Last_child->{n} }
;
```

10.10 $node->descendant

The `descendant` method returns the descendant of a node given its *coordinates.* The coordinates of a node `$s` relative to a tree `$t` to which it belongs is a string of numbers separated by dots like `".1.3.2"` which denotes the *child path* from `$t` to `$s`, i.e. `$s == $t->child(1)->child(3)->child(2)`.

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]
```

10.11 $node->str

The `str` method returns a string representation of the tree. The *str* method traverses the syntax tree dumping the type of the node being visited in a string. If the node being visited has a method `info` it will be executed and its result inserted between `$DELIMITER`s into 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 being built by `str`.

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 = ','; # Separator between nodes, by default a comma
$DELIMITER = '['; # The string returned by C<info> will be enclosed
$FOOTNOTE_HEADER = "\n---------------------------\n";
$FOOTNOTE_SEP = ")\n";
$FOOTNOTE_LEFT = '^{'; # Left delimiter for a footnote number
$FOOTNOTE_RIGHT = '}'; # Right delimiter for a footnote number
$LINESEP = 4; # When indent=2 the enclosing parenthesis will be
              # commented if more than $LINESEP apart
```

The following list defines the `$DELIMITER`s you can choose for attribute representation:

```
'[' => ']', '{' => '}', '(' => ')', '<' => '>'
```

If the node being visited has a method `footnote`, the string returned by the method will be concatenated at the end of the string as a footnote. The variables `$FOOTNOTE_LEFT` and `$FOOTNOTE_RIGHT` govern the displaying of footnote numbers.

Follows an example of output using `footnotes`.

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/script> \
                                        usetypes.pl prueba24.c
PROGRAM^{0}(FUNCTION[f]^{1}(RETURNINT(TIMES(INUM(TERMINAL[2:2]),VAR(TERMINAL[a:2])))))
---------------------------
0)
Types:
$VAR1 = {
  'CHAR' => bless( {
    'children' => []
  }, 'CHAR' ),
  'VOID' => bless( {
    'children' => []
  }, 'VOID' ),
  'INT' => bless( {
    'children' => []
  }, 'INT' ),
  'F(X_1(INT),INT)' => bless( {
    'children' => [
      bless( {
        'children' => [
          $VAR1->{'INT'}
        ]
      }, 'X_1' ),
      $VAR1->{'INT'}
    ]
  }, 'F' )
};
Symbol Table:
$VAR1 = {
  'f' => {
    'type' => 'F(X_1(INT),INT)',
    'line' => 1
  }
};

---------------------------
1)
$VAR1 = {
  'a' => {
    'type' => 'INT',
    'param' => 1,
    'line' => 1
  }
};
```

The first footnote was due to a call to `PROGRAM:footnote`. The `footnote` method for the `PROGRAM` node was defined as:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> \
                             sed -n -e '691,696p' Types.eyp | cat -n
     1  sub PROGRAM::footnote {
     2    return "Types:\n"
     3           .Dumper($_[0]->{types}).
     4           "Symbol Table:\n"
     5           .Dumper($_[0]->{symboltable})
     6  }
```

The second footnote was produced by the existence of a `FUNCTION::footnote` method:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> \
                           sed -n -e '702,704p' Types.eyp | cat -n
1  sub FUNCTION::footnote {
2    return Dumper($_[0]->{symboltable})
3  }
```

The source program for the example was:

```
int f(int a) {
  return 2*a;
}
```

10.12 $node->delete($child)

The `delete` method is used to delete the specified child of `$node`. The child to delete can be specified using the index or a reference. It returns the deleted child.

Throws an exception if the object can't do `children` or has no `children`. See also the `delete` method of treeregexes (`Parse::Eyapp:YATW` objects) to delete the node being visited.

The following example moves out of a loop an assignment statement assuming is an invariant of the loop. To do it uses the `delete` and `insert_before` methods:

```
nereida:~/src/perl/YappWithDefaultAction/examples> \
            sed -ne '98,113p' moveinvariantoutofloopcomplexformula.pl
my $p = Parse::Eyapp::Treeregexp->new( STRING => q{
  moveinvariant: BLOCK(
                   @prests,
                   WHILE(VAR($b), BLOCK(@a, ASSIGN($x, NUM($e)), @c)),
                   @possts
                 )
    => {
         my $assign = $ASSIGN;
         $BLOCK[1]->delete($ASSIGN);
         $BLOCK[0]->insert_before($WHILE, $assign);
       }
  },
  FIRSTLINE => 99,
);
$p->generate();
$moveinvariant->s($t);
```

The example below deletes CODE nodes from the tree build for a translation scheme:

```
my $transform = Parse::Eyapp::Treeregexp->new(
  STRING=>q{delete_code: CODE => { Parse::Eyapp::Node::delete($CODE) }},
)
```

Observe how delete is called as a subroutine.

10.13 $node->unshift($newchild)

Inserts `$newchild` at the beginning of the list of children of `$node`. See also the `unshift` method for `Parse::Eyapp:YATW` treeregexp transformation objects

10.14 $node->push($newchild)

Inserts `$newchild` at the end of the list of children of `$node`.

10.15 $node->insert_before($position, $new_child)

Inserts `$newchild` before `$position` in the list of children of `$node`. Variable `$position` can be an index or a reference.

The method throws an exception if `$position` is an index and is not in range. Also if `$node` has no children.

The method throws a warning if `$position` is a reference and does not define an actual child. In such case `$new_child` is not inserted.

See also the insert_before method for `Parse::Eyapp:YATW` treeregexp transformation objects

10.16 $node->insert_after($position, $new_child)

Inserts `$newchild` after `$position` in the list of children of `$node`. Variable `$position` can be an index or a reference.

The method throws an exception if `$position` is an index and is not in the range of `$node`-children>.

The method throws a warning if `$position` is a reference and does not exists in the list of children. In such case `$new_child` is not inserted.

10.17 $node->translation_scheme

Traverses $node. Each time a CODE node is visited the subroutine referenced is called with arguments the node and its children. Usually the code will decorate the nodes with new attributes or will update existing ones. Obviously this method does nothing for an ordinary AST. It is used after compiling an Eyapp program that makes use of the `%metatree` directive.

10.18 $node->bud

Bottom-up decorator. The tree is traversed bottom-up. The set of transformations is applied to each node in the order supplied by the user. As soon as one succeeds no more transformations are applied. For an example see the files `examples/Types.eyp` and `examples/Trans.trg`. The code below shows an extract of the type-checking phase:

```
nereida:~/src/perl/YappWithDefaultAction/examples> \
                        sed -ne '600,611p' Types.eyp
 my @typecheck = (
   our $inum,
   our $charconstant,
   our $bin,
   our $arrays,
   our $assign,
   our $control,
   our $functioncall,
   our $statements,
 );

 $t->bud(@typecheck);

nereida:~/src/perl/YappWithDefaultAction/examples> \
                        sed -ne '183,192p' Trans.trg
control: /IF|IFELSE|WHILE/:con($bool)
  => {
    $bool = char2int($con, 0) if $bool->{t} == $CHAR;
      type_error("Condition must have integer type!", $bool->line)
    unless $bool->{t} == $INT;

    $con->{t} = $VOID;

    return 1;
  }
```

11 TRANSFORMATIONS: Parse::Eyapp:YATW

Parse::Eyapp:YATW objects represent tree transformations.

11.1 Parse::Eyapp::Node->new

Builds a treeregexp transformation object. Though usually you build a transformation by means of Treeregexp programs you can directly invoke the method to build a tree transformation. A transformation object can be built from a function that conforms to the YATW tree transformation call protocol.

Follows an example (file `examples/12ts_simplify_with_s.pl`):

```
nereida:~/src/perl/YappWithDefaultAction/examples> \
       sed -ne '68,$p' 12ts_simplify_with_s.pl | cat -n
 1  sub is_code {
 2    my $self = shift; # tree
 3
 4    # $_[0] is the father, $_[1] the index
 5    if ((ref($self) eq 'CODE')) {
 6      splice(@{$_[0]->{children}}, $_[1], 1);
 7      return 1;
 8    }
 9    return 0;
10  }
11
12  Parse::Eyapp->new_grammar(
13    input=>$translationscheme,
14    classname=>'Calc',
15    firstline =>7,
16  );
17  my $parser = Calc->new();                # Create the parser
18
19  $parser->YYData->{INPUT} = "2*-3\n";  print "2*-3\n"; # Set the input
20  my $t = $parser->Run;                    # Parse it
21  print $t->str."\n";
22  my $p = Parse::Eyapp::YATW->new(PATTERN => \&is_code);
23  $p->s($t);
24  { no warnings; # make attr info available only for this display
25    local *TERMINAL::info = sub { $_[0]{attr} };
26    print $t->str."\n";
27  }
```

After the `Parse::Eyapp::YATW` object `$p` is built at line 22 the call to method `$p->s($t)` applies the transformation `is_code` using a bottom-up traversing of the tree `$t`. The effect achieved is the elimination of `CODE` references in the translation scheme tree. When executed the former code produces:

```
nereida:~/src/perl/YappWithDefaultAction/examples> 12ts_simplify_with_s.pl
2*-3
EXP(TIMES(NUM(TERMINAL,CODE),TERMINAL,UMINUS(TERMINAL,NUM(TERMINAL,CODE),CODE),CODE),CODE)
EXP(TIMES(NUM(TERMINAL[2]),TERMINAL[*],UMINUS(TERMINAL[-],NUM(TERMINAL[3]))))
```

The file `foldrule6.pl` in the `examples/` distribution directory gives you another example:

```
nereida:~/src/perl/YappWithDefaultAction/examples> cat -n foldrule6.pl
#!/usr/bin/perl -w
use strict;
use Rule6;
use Parse::Eyapp::YATW;

my %BinaryOperation = (PLUS=>'+', MINUS => '-', TIMES=>'*', DIV => '/');

sub set_terminfo {
  no warnings;
  *TERMINAL::info = sub { $_[0]{attr} };
}
sub is_foldable {
  my ($op, $left, $right);
  return 0 unless defined($op = $BinaryOperation{ref($_[0])});
  return 0 unless ($left = $_[0]->child(0), $left->isa('NUM'));
  return 0 unless ($right = $_[0]->child(1), $right->isa('NUM'));

  my $leftnum = $left->child(0)->{attr};
  my $rightnum = $right->child(0)->{attr};
  $left->child(0)->{attr} = eval "$leftnum $op $rightnum";
  $_[0] = $left;
}

my $parser = new Rule6();
$parser->YYData->{INPUT} = "2*3";
my $t = $parser->Run;
&set_terminfo;
print "\n***** Before ******\n";
print $t->str;
my $p = Parse::Eyapp::YATW->new(PATTERN => \&is_foldable);
$p->s($t);
print "\n***** After ******\n";
print $t->str."\n";
```

when executed produces:

```
nereida:~/src/perl/YappWithDefaultAction/examples> foldrule6.pl

***** Before ******
TIMES(NUM(TERMINAL[2]),NUM(TERMINAL[3]))
***** After ******
NUM(TERMINAL[6])
```

11.2 The YATW Tree Transformation Call Protocol

For a subroutine `pattern_sub` to work as a YATW tree transformation - as sub `is_code` above - has to conform to the following call description:

```
pattern_sub(
    $_[0],  # Node being visited
    $_[1],  # Father of this node
    $index, # Index of this node in @Father->children
    $self,  # The YATW pattern object
);
```

The `pattern_sub` must return TRUE if matched and FALSE otherwise.

The protocol may change in the near future. Avoid using other information than the fact that the first argument is the node being visited.

11.3 Parse::Eyapp::YATW->buildpatterns

Works as Parse::Eyapp->new but receives an array of subs conforming to the YATW Tree Transformation Call Protocol.

```
our @all = Parse::Eyapp::YATW->buildpatt(\&delete_code, \&delete_tokens);
```

11.4 $yatw->delete

The root of the tree that is currently matched by the YATW transformation `$yatw` will be deleted from the tree as soon as is safe. That usually means when the processing of their siblings is finished. The following example (taken from file `examples/13ts_simplify_with_delete.pl` in the Parse::Eyapp distribution) illustrates how to eliminate CODE and syntactic terminals from the syntax tree:

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ \
       sed -ne '62,$p' 13ts_simplify_with_delete.pl | cat -n
sub not_useful {
  my $self = shift; # node
  my $pat = $_[2];  # get the YATW object

  (ref($self) eq 'CODE') or ((ref($self) eq 'TERMINAL') and ($self->{token} eq $self->{attr}))
    or do { return 0 };
  $pat->delete();
  return 1;
}

Parse::Eyapp->new_grammar(
  input=>$translationscheme,
  classname=>'Calc',
  firstline =>7,
);
my $parser = Calc->new();                # Create the parser

$parser->YYData->{INPUT} = "2*3\n"; print $parser->YYData->{INPUT};
my $t = $parser->Run;                    # Parse it
print $t->str."\n";                      # Show the tree
my $p = Parse::Eyapp::YATW->new(PATTERN => \&not_useful);
$p->s($t);                               # Delete nodes
print $t->str."\n";                      # Show the tree
```

when executed we get the following output:

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ 13ts_simplify_with_delete.pl
2*3
EXP(TIMES(NUM(TERMINAL,CODE),TERMINAL,NUM(TERMINAL,CODE),CODE))
EXP(TIMES(NUM(TERMINAL),NUM(TERMINAL))
```

11.5 $yatw->unshift($b)

Safely unshifts the node `$b` in the list of children of the father of the node that matched with `$yatw` (i.e in $_[0]). The following example (file `examples/26delete_with_trreereg.pl` shows a YATW transformation `insert_child` that illustrates the use of `unshift`:

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ \
        sed -ne '70,$p' 26delete_with_trreereg.pl | cat -n
my $transform = Parse::Eyapp::Treeregexp->new( STRING => q{

    delete_code : CODE => { $delete_code->delete() }

    {
      sub not_semantic {
        my $self = shift;
        return  1 if ((ref($self) eq 'TERMINAL') and ($self->{token} eq $self->{attr}));
        return 0;
      }
    }

    delete_tokens : TERMINAL and { not_semantic($TERMINAL) } => {
      $delete_tokens->delete();
    }

    insert_child : TIMES(NUM(TERMINAL), NUM(TERMINAL)) => {
      my $b = Parse::Eyapp::Node->new( 'UMINUS(TERMINAL)',
                sub { $_[1]->{attr} = '4.5' });

      $insert_child->unshift($b);
    }
  },
)->generate();

Parse::Eyapp->new_grammar(
  input=>$translationscheme,
  classname=>'Calc',
  firstline =>7,
);
my $parser = Calc->new();                # Create the parser

$parser->YYData->{INPUT} = "2*3\n"; print $parser->YYData->{INPUT}; # Set the input
my $t = $parser->Run;                # Parse it
print $t->str."\n";                        # Show the tree
# Get the AST
our ($delete_tokens, $delete_code);
$t->s($delete_tokens, $delete_code);
print $t->str."\n";                        # Show the tree
our $insert_child;
$insert_child->s($t);
print $t->str."\n";                        # Show the tree
```

When is executed the program produces the following output:

```
pl@nereida:~/src/perl/YappWithDefaultAction/examples$ 26delete_with_trreereg.pl
2*3
EXP(TIMES(NUM(TERMINAL,CODE),TERMINAL,NUM(TERMINAL,CODE),CODE))
EXP(TIMES(NUM(TERMINAL),NUM(TERMINAL)))
EXP(UMINUS(TERMINAL),TIMES(NUM(TERMINAL),NUM(TERMINAL)))
```

Don't try to take advantage that the transformation sub receives in $_[1] a reference to the father (see the section YATW Tree Transformation protocol) and do something like:

```
unshift $_[1]->{children}, $b
```

it is unsafe.

11.6 $yatw->insert_before($node)

Safely inserts $node in the list of children node before the node $_[0] that matched with `$yatw`.

The following example (file `t/33moveinvariantoutofloop.t`) illustrates its use:

```
my $p = Parse::Eyapp::Treeregexp->new( STRING => q{
  moveinvariant: WHILE(VAR($b), BLOCK(@a, ASSIGN($x, $e), @c))
       and { is_invariant($ASSIGN, $WHILE) } => {
         my $assign = $ASSIGN;
         $BLOCK->delete($ASSIGN);
         $moveinvariant->insert_before($assign);
       }
  },
);
```

Here the `$assign` tree will be moved before the node `$WHILE` in the list of siblings of `$WHILE` (presumably the father is a `BLOCK` node).

12 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::Matc`
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`)

12.1 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
#!/usr/bin/perl -w
use strict;
use Rule6;
use Parse::Eyapp::Treeregexp;
use Parse::Eyapp::Node;

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

my $severity = shift || 0;
my $parser = new Rule6();
$parser->YYData->{INPUT} = shift || '0*2';
my $t = $parser->Run;

my $transform = Parse::Eyapp::Treeregexp->new(
  STRING => q{
    zero_times_whatever: TIMES(NUM($x)) and { $x->{attr} == 0 } => { $_[0] = $NUM }
  },
  SEVERITY => $severity,
  FIRSTLINE => 15,
)->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
```

13 Tree Substitution: The `s` methods

Both `Parse::Eyapp:Node` and `Parse::Eyapp::YATW` objects (i.e. nodes and tree transformations) are provided with a `s` method.

In the case of a `Parse::Eyapp::YATW` object the method `s` applies the tree transformation using a single bottom-up traversing: the transformation is applied to the children subtrees and -if the current node matches - to the current nodes.

For `Parse::Eyapp:Node` nodes the set of transformations is applied to each node until no transformation matches any more. The example in the SYNOPSIS section illustrates the use:

```
# 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
```

```
19
20  $t->s($uminus); # Transform UMINUS nodes
21  $t->s(@all);    # constant folding and mult. by zero
```

The call at line 20 can be substituted by `$uminus->s($t)` without changes.

14 Parse::Eyapp::Scope

A scope manager helps to compute the mapping function that maps the uses (instances) of objects to its definitions. For instance, to asociate each ocurrence of an identifier with the declaration that applies to it. Another example is to associate each occurrence of a `CONTINUE` or `BREAK` node with the shallowest `LOOP` that encloses it. Or to associate a `GOTO` node to jump to, that is, with the statement node associated with the label.

The compiler writer must mark at the appropriate time (new block for identifier scope analysis, new loop for loop scope analysis, etc.) the *beginning of a new scope* calling the method begin_scope. From that point on any *ocurring instance* of an object (for example, variables in expressiones for identifier scope analysis, breaks and continues for loop scope analysis, etc.) must be declared calling the method scope_instance. The programmer must also mark the *end of the current scope* at the appropriate time.

14.1 $scope->end_scope

There are two ways of calling `$scope->end_scope`. The first one is for Scope Analysis Problems where a symbol table is needed (for example in variable scope analysis).

14.1.1 $scope->end_scope with first Arg a Symbol Table

For each *ocurring instance* of an object `$x` that occurred since the last call to begin_scope the call to

```
$scope->end_scope(\%symboltable, $definition_node, 'attr1', 'attr2', ... )
```

decorates the *ocurring instance* `$x` with several attributes:

- An entry `$x->{SCOPE_NAME}` is built that will reference `$definition_node`.
- An entry `$x->{ENTRY_NAME}` is built. That entry references `$symboltable{$x->key}` (to have a faster access from the instance to the attributes of the object).
- For each aditional arguments `attr#k` an entry `$x->{attr#k}` will be built. That entry references `$symboltable{$x->key}{attr#k}`. Therefore the entry for `$x` in the symbol table must already have a field named `attr#k`.

In a list context `$scope>end_scope` returns two references. The first one is a reference to a list of node instances that weren't defined in the current scope. The second is a reference to a list of nodes that were defined in this scope. In a scalar context returns the first of these two. An instance `$x` is *defined* if, and only if, `exists $symboltable{$_->key}`.

14.1.2 $scope->end_scope for Simple Scope Analysis

Some scope analysis problems do not require the existence of a symbol table (for instance, the problem of associating a `RETURN` node with the `FUNCTION` that encloses it). For such kind of problems `$scope>end_scope` provides a second form of call. The second way to call `$scope>end_scope` is

```
$declared = $scopemanager->end_scope($definition_node);
```

The only argument is the reference to the node that controls/defines the scope. The method returns a reference to the declared nodes. Any node instanced with `scope_instance` since the last call to `begin_scope` is considered *declared.*

14.2 $scope->begin_scope

Marks the beginning of an scope. Example (file `examples/Types.eyp`):

```
loopPrefix:
    $WHILE '(' expression ')'
      {
        $loops->begin_scope;
        $_[3]->{line} = $WHILE->[1];
        $_[3]
      }
```

14.3 $scope->scope_instance

Declares the node argument to be an occurring instance of the scope:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code> \
    sed -ne '375,380p' Simple6.eyp | cat -n
 1      $Variable '=' binary
 2        {
 3          my $parser = shift;
 4          $ids->scope_instance($Variable);
 5          $parser->YYBuildAST(@_);
 6        }
```

14.4 Parse::Eyapp::Scope->new

`Parse::Eyapp::Scope->new` returns a scope managment object. The scope mapping function is implemented by `Parse::Eyapp::Scope` through a set of attributes that are added to the nodes involved in the scope analysis. Some of these attributes can be specified using the parameters of `Parse::Eyapp::Scope->new`. The arguments of `new` are:

- `SCOPE_NAME` is the name choosed for the attribute of the *node instance* which will held the reference to the *definition node*. If is not specified it will take the value `"scope"`.
- `ENTRY_NAME` is the name of the attribute of the *node instance* which will held the reference to the symbol table entry. By default takes the value `"entry"`.
- `SCOPE_DEPTH` is the name for an attribute of the es el nombre del atributo del nodo ámbito (nodo bloque) y contiene la profundidad de anidamiento del ámbito. Es opcional. Si no se especifica no será guardado.

15 ENVIRONMENT

Remember to set the environment variable `PERL5LIB` if you decide to install `Parse::Eyapp` at a location other than the standard. For example, on a bash or sh:

```
export PERL5LIB-/home/user/wherever_it_is/lib/:$PERL5LIB
```

on a `csh` or <tcsh>

```
setenv PERL5LIB /home/user/wherever_it_is/lib/:$PERL5LIB
```

Be sure the scripts `eyapp` and `treereg` are in the execution PATH.

16 DEPENDENCIES

This distribution depends on the following modules:

- *List::MoreUtils*
- *List::Util*
- *Data::Dumper*

- *Pod::Usage*

It seems that *List::Util* is in the core of Perl distributions since version 5.73:

```
> perl -MModule::CoreList -e 'print Module::CoreList->first_release("List::Util")'
5.007003
```

and *Data::Dumper* is also in the core since 5.5:

```
> perl -MModule::CoreList -e 'print Module::CoreList->first_release("Data::Dumper")'
5.005
```

and *Pod::Usage* is also in the core since 5.6:

```
> perl -MModule::CoreList -e 'print Module::CoreList->first_release("Pod::Usage")'
5.006
```

I also recommend the following modules:

- *Test::Pod*
- *Test::Warn*
- *Test::Exception*

The dependence on *Test::Warn*, *Test::Pod* and *Test::Exception* is merely for the execution of tests. If the modules aren't installed the tests depending on them will be skipped.

17 INSTALLATION

To install it, follow the traditional mantra:

```
perl Makefile.PL
make
make test
make install
```

Also:

- Make a local copy of the `examples/` directory in this distribution
- Probably it will be also a good idea to make a copy of the tests in the `t/` directory. They also illustrate the use of Eyapp
- Print `eyapptut.pdf` and `Eyapp.pdf`.
- Consider also saving the HTML files
- Read eyapptut.pdf first

18 BUGS AND LIMITATIONS

- This distribution is an alpha version. I plan to make a release in CPAN by February 2007. Hopefully, at that time the interface will freeze or -at least- changes in the API will be minor. In the meanwhile it will be likely to change.
- The way Parse::Eyapp parses Perl code is verbatim the way it does Parse::Yapp 1.05. I quote Francois Desarmenien *Parse::Yapp* documentation:

  "Be aware that matching braces in Perl is much more difficult than in C: inside strings they don't need to match. While in C it is very easy to detect the beginning of a string construct, or a single character, it is much more difficult in Perl, as there are so many ways of writing such literals. So there is no check for that today. If you need a brace in a double-quoted string, just quote it (`\{` or `\}`). For single-quoted strings, you will need to make a comment matching it *in th right order*. Sorry for the inconvenience.

```
    {
        "{ My string block }".
        "\{ My other string block \}".
        qq/ My unmatched brace \} /.
        # Force the match: {
        q/ for my closing brace } /
        q/ My opening brace { /
        # must be closed: }
    }
```

All of these constructs should work."

I wrote an alternative *exact solution* but resulted in much slower code. Therefore, until something better is found, I rather prefer for Parse::Eyapp to live with this limitation.

The same limitation may appear inside header code (code between `%{` and `%}`)

- English is not my native language. I know for sure this text has lexical, syntactic and semantic errors. I'll be most gratefull to know about any typos, grammar mistakes, ways to rewrite paragraphs and misconceptions you have found.
- I am sure there are unknown bugs. Please report problems to Casiano Rodriguez-Leon (casiano@cpan.org).

19 SEE ALSO

- *Parse::eyapptut*
- 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*.

20 REFERENCES

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

21 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.

22 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.

23 LICENCE AND COPYRIGHT

Index