

# 1 NAME

Parse::Eyapp::Node - The nodes of the Syntax Trees

## 2 SYNOPSIS

```
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 EXPRESSION_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}) { # Topicalize
            m{\G\s+}gc;
            $_ eq '' and return('',undef);
            m{\G([0-9]+(?:\.[0-9]+)?)}gc and return('NUM',$1);
            m{\G([A-Za-z][A-Za-z0-9_]*)}gc and return('VAR',$1);
            m{\G(.)}gcs and return($1,$1);
        }
        return('',undef);
    }
}
```

```

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 = $Op{ref($bin)};
            $x->{attr} = eval "$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 transformations

$t->s($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";

```

### 3 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 access to the methods provided in such module.

The examples used in this document can be found in the directory `examples/Node` accompanying the distribution of *Parse::Eyapp*.

#### **Parse::Eyapp::Node->new**

Nodes are usually created from a Eyapp grammar using the `%tree` or `%metatree` 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 of these trees.

The following example (see file `examples/Node/28foldwithnewwithvars.pl`) of a `treeregexp` transformation 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 call to `Parse::Eyapp::Node->new` creates a tree `NUM(TERMINAL)` and decorates the `TERMINAL` leaf with attributes `attr` and `token`. The `constantfold` transformation substitutes all the binary operation trees whose children are numbers for a `NUM(TERMINAL)` tree holding as attribute the number resulting of operating the two numbers.

The input string can describe more than one tree. Different trees are separated by white spaces. Consider the following example (in `examples/Node/builder.pl`):

```
$ cat -n builder.pl
 1 #!/usr/bin/perl -w
 2 use strict;
 3 use Parse::Eyapp::Node;
 4
 5 use Data::Dumper;
 6 $Data::Dumper::Indent = 1;
 7 $Data::Dumper::Purity = 1;
 8
 9 my $string = shift || 'ASSIGN(VAR(TERMINAL), TIMES(NUM(TERMINAL),NUM(TERMINAL)))';
10 my @t = Parse::Eyapp::Node->new(
11     $string,
12     sub { my $i = 0; $_->{n} = $i++ for @_ }
13 );
14
15 print "*****\n";
16 print Dumper(\@t);
```

When feed with input `'A(C,D) E(F)'` the following forest is built:

```
$ builder.pl 'A(C,D) E(F)'
*****
$VAR1 = [
  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' ),
    {}
];
$VAR1->[1] = $VAR1->[0]{'children'}[0];
$VAR1->[2] = $VAR1->[0]{'children'}[1];
$VAR1->[4] = $VAR1->[3]{'children'}[0];

```

Thusm, the forest @t contains 5 subtrees A(C,D), C, D, E(F) and F.

## Directed Acyclic Graphs with Parse::Eyapp::Node->hnew

Parse::Eyapp provides the method Parse::Eyapp::Node->hnew to build *Directed Acyclic Graphs* (DAGs) instead of trees. They are built using *hashed consing*, i.e. *memoizing* the creation of nodes.

The method Parse::Eyapp::Node->hnew 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 in some compiler phases and tasks like detecting *common subexpressions* or during *type checking*. See file Types.eyp in examples/typechecking/Simple-Types-XXX.tar. for a more comprehensive example.

## Expanding Directed Acyclic Graphs with Parse::Eyapp::Node->hexpand

Calls to Parse::Eyapp::Node->hexpand have the syntax

```
$z = Parse::Eyapp::Node->hexpand('CLASS', @children, \&handler)
```

Creates a dag of type 'CLASS' with children @children in a way compatible with hnew. The last optional argument can be a reference to a sub. Such sub will be called after the creation of the DAG with a reference to the root of the DAG as single argument. The following session with the debugger illustrates the use of Parse::Eyapp::Node->hexpand. First we create a DAG using hnew:

```

pl@nereida:~/Lbook/code/Simple-Types/script$ perl -MParse::Eyapp::Node -wde 0
main::(-e:1): 0
DB<1> $x = Parse::Eyapp::Node->hnew('A(C(B),C(B))')
DB<2> x $x
0 A=HASH(0x850c850)
  'children' => ARRAY(0x850ca30)
    0 C=HASH(0x850c928)
      'children' => ARRAY(0x850c9e8)
        0 B=HASH(0x850c9a0)
          'children' => ARRAY(0x83268c8)
            empty array
    1 C=HASH(0x850c928)
      -> REUSED_ADDRESS

```

We obtain the REUSED\_ADDRESS for the second child since the C(B) subtree appears twice. Now, suppose we want to expand the existing tree/DAG C(B) to A(C(B)). We can do that using `hexpand`:

```

DB<3> $y = Parse::Eyapp::Node->hexpand('A', $x->child(0))
DB<4> x $y
0 A=HASH(0x8592558)
  'children' => ARRAY(0x832613c)
    0 C=HASH(0x850c928)
      'children' => ARRAY(0x850c9e8)
        0 B=HASH(0x850c9a0)
          'children' => ARRAY(0x83268c8)
            empty array

```

We get new memory for C<\$y>: C<HASH(0x8592558)> is anew address. Assume we want to expand the tree/DAG C<C(B)> to C<A(C(B),C(B))>. We can do it this way:

```

DB<5> $z = Parse::Eyapp::Node->hexpand('A', $x->children)
DB<6> x $z
0 A=HASH(0x850c850)
  'children' => ARRAY(0x850ca30)
    0 C=HASH(0x850c928)
      'children' => ARRAY(0x850c9e8)
        0 B=HASH(0x850c9a0)
          'children' => ARRAY(0x83268c8)
            empty array
    1 C=HASH(0x850c928)
      -> REUSED_ADDRESS

```

Notice that the address c<0x850c850> for \$z is the same than the address for \$x. No new memory has been allocated for \$z.

The following command illustrates the use of `hexpand` with a handler:

```

DB<7> $z = Parse::Eyapp::Node->hexpand('A', $x->children, sub { $_[0]->{t} = "X" })
DB<8> x $z
0 A=HASH(0x850c850)
  'children' => ARRAY(0x850ca30)
    0 C=HASH(0x850c928)
      'children' => ARRAY(0x850c9e8)
        0 B=HASH(0x850c9a0)
          'children' => ARRAY(0x83268c8)
            empty array
    1 C=HASH(0x850c928)
      -> REUSED_ADDRESS
't' => 'X'

```

## \$node->type

Returns (or sets) the type (class) of the node. It can be called as a subroutine 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 AST node with one of those words.

To be used as a setter, be sure *Parse::Eyapp::Driver* is loaded:

```
$ perl -MParse::Eyapp::Driver -MParse::Eyapp::Node -wde0
main::(-e:1): 0
DB<1> x $t = Parse::Eyapp::Node->new("A(B,C)") # Creates a tree
0 A=HASH(0x8557bdc)
  'children' => ARRAY(0x8557c90)
    0 B=HASH(0x8557cf0)
      'children' => ARRAY(0x8325804)
        empty array
    1 C=HASH(0x8557c6c)
      'children' => ARRAY(0x8557d5c)
        empty array
DB<2> x $t->type('FUN') # Change the type of $t to 'FUN'
0 'FUN'
DB<3> x $t
0 FUN=HASH(0x8557bdc)
  'children' => ARRAY(0x8557c90)
    0 B=HASH(0x8557cf0)
      'children' => ARRAY(0x8325804)
        empty array
    1 C=HASH(0x8557c6c)
      'children' => ARRAY(0x8557d5c)
        empty array
DB<4> x $t->isa('Parse::Eyapp::Node')
0 1
```

## **\$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 exist. It has two obligatory parameters: the node (since it is a method) and the index of the child. Sets the new value if called

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

The method will croak if the obligatory parameters are not provided.

In the files `examples/Node/TSwithtreetransformations2.ey` and `examples/node/usetwithtreetransformations2` you can find a somewhat complicated example of call to `child` as a setter. It is inside a transformation that swaps the children of a PLUS node (remember that the tree is a concrete tree including code since it is a translation scheme built under the directive `%metatree`):

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

    commutative_add: PLUS($x, ., $y, .) # 1st dot correspond to '+' 2nd dot to CODE
    => { my $t = $x; $_[0]->child(0, $y); $_[0]->child(2, $t)}

    .....
}
```

## 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:

```
examples/Node$ cat -n alias_and_yyprefix.pl
 1  #!/usr/local/bin/perl
 2  use warnings;
 3  use strict;
 4  use Parse::Eyapp;
 5
 6  my $grammar = q{
 7      %prefix R::S::
 8
 9      %right  '='
10     %left   '- ' '+'
11     %left   '* ' '/'
12     %left   NEG
13     %tree  bypass alias
14
15     %%
16     line: $exp { $_[1] }
17     ;
18
19     exp:
20         %name NUM
21             $NUM
22     | %name VAR
23         $VAR
24     | %name ASSIGN
25         $VAR '=' $exp
26     | %name PLUS
27         exp.left '+' exp.right
28     | %name MINUS
29         exp.left '-' exp.right
30     | %name TIMES
31         exp.left '*' exp.right
32     | %name DIV
33         exp.left '/' exp.right
34     | %no bypass UMINUS
35         '- ' $exp %prec NEG
36     | '(' exp ')' { $_[2] } /* Let us simplify a bit the tree */
37     ;
38
39     %%
40
    .....
```

```

76  }; # end grammar
77
78
79  Parse::Eyapp->new_grammar(
80    input=>$grammar,
81    classname=>'Alias',
82    firstline =>7,
83    outputfile => 'main',
84  );
85  my $parser = Alias->new();
86  $parser->YYData->{INPUT} = "a = -(2*3+5-1)\n";
87  my $t = $parser->Run;
88  $Parse::Eyapp::Node::INDEXT=0;
89  print $t->VAR->str."\n";           # a
90  print "*****\n";
91  print $t->exp->exp->left->str."\n"; # 2*3+5
92  print "*****\n";
93  print $t->exp->exp->right->str."\n"; # 1

```

Here methods with names `left` and `right` will be created inside the class `R::S` (see the use of the `%prefix` directive in line 7) to access the corresponding children associated with the two instances of `exp` in the right hand side of the production rule. when executed, the former program produces this output:

```

examples/Node$ alias_and_yyprefix.pl
R::S::TERMINAL
*****
R::S::PLUS(R::S::TIMES(R::S::NUM,R::S::NUM),R::S::NUM)
*****
R::S::NUM

```

## \$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/Node/TSPostfix3.eyy` for an example of use inside a Translation Scheme:

```

examples/Node$ cat TSPostfix3.eyy
..... # precedence declarations

%metatree

%defaultaction {
  if (@_==2) { # NUM and VAR
    $lhs->{t} = $_[1]->{attr};
    return
  }
  if (@_==4) { # binary operations
    $lhs->{t} = "$_[1]->{t} $_[3]->{t} $_[2]->{attr}";
    return
  }
  die "Fatal Error. Unexpected input. Numargs = ".scalar(@_)." \n".Parse::Eyapp::Node->str(@_);
}

%%
line: %name PROG
      exp <%name EXP + ';'>
        { @{$lhs->{t}} = map { $_->{t}} ($_[1]->children()); }

;

exp:      %name NUM NUM
        | %name VAR VAR
        | %name ASSIGN VAR '=' exp { $lhs->{t} = "$_[1]->{attr} $_[3]->{t} ="; }

```

```

| %name PLUS   exp '+' exp
| %name MINUS  exp '-' exp
| %name TIMES  exp '*' exp
| %name DIV    exp '/' exp
| %name NEG    '-' exp %prec NEG { $_[0]->{t} = "$_[2]->{t} NEG" }
| '(' exp ')' %begin { $_[2] }
;

%%

.....

```

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.eyy` 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.

### \$node->Children

Returns the array of children of the node. When dealing with a translation scheme, the `$node->Children` method (Notice the case difference with `$node->children`, first in uppercase) returns the non CODE children of the node. The following execution with the debugger of the example in `examples/Node/ts_with_ast.pl` illustrates the difference:

```

examples/Node$ perl -wd ts_with_ast.pl
main:(ts_with_ast.pl:6):      my $translationscheme = q{
main:(ts_with_ast.pl:7):      %{

```

The `$translationscheme` variable contains the code of a small calculator:

```

%metatree

%left  '-' '+'
%left  '*'
%left  NEG

%%
line:   %name EXP
        $exp { $lhs->{n} = $exp->{n} }
;

exp:
    %name PLUS
        exp.left '+' exp.right
        { $lhs->{n} .= $left->{n} + $right->{n} }
| %name TIMES
    exp.left '*' exp.right

```

```

        { $lhs->{n} = $left->{n} * $right->{n} }
|   %name NUM   $NUM
        { $lhs->{n} = $NUM->{attr} }
|   '( ' $exp ') ' %begin { $exp }
|   exp.left '-' exp.right
        { $lhs->{n} = $left->{n} - $right->{n} }

|   '-' $exp %prec NEG
        { $lhs->{n} = -$exp->{n} }
;

```

We run the program with input 2+(3) and stop it at line 88, just after the augmented AST (CODE node included) has been built:

```

DB<1> c 88
main::(ts_with_ast.pl:88):      $t->translation_scheme;

```

Now, let us see the difference between the methods `children` and `Children`:

```

DB<2> @a = $t->children; @b = $t->Children
DB<3> print Parse::Eyapp::Node::str($_)."\n" for @a
PLUS(NUM(TERMINAL, CODE), TERMINAL, NUM(TERMINAL, CODE), CODE)
CODE
DB<4> print $_->str."\n" for @b
PLUS(NUM(TERMINAL, CODE), TERMINAL, NUM(TERMINAL, CODE), CODE)
DB<5>

```

## `$node->last_child`

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

## `$node->Last_child`

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

```

examples/Node$ 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  %prefix Calc::
15
16  %defaultaction {
17      $lhs->{n} = eval " $left->{n} $_[2]->{attr} $right->{n} "
18  }
19
20  %metatree
21
22  %right    '='
23  %left    '- ' '+ '
24  %left    '* ' '/'
25
26  %%

```

```

27 line:      %name EXP
28           exp <+ ';'> /* Expressions separated by semicolons */
29           { $lhs->{n} = $_[1]->Last_child->{n} }
30 ;
31
32 exp:
33         %name PLUS
34         exp.left '+' exp.right
35     |   %name MINUS
36         exp.left '-' exp.right
37     |   %name TIMES
38         exp.left '*' exp.right
39     |   %name DIV
40         exp.left '/' exp.right
41     |   %name NUM
42         $NUM
43         { $lhs->{n} = $NUM->{attr} }
44     |   '(' $exp ')' %begin { $exp }
45     |   %name VAR
46         $VAR
47         { $lhs->{n} = $sym{$VAR->{attr}}->{n} }
48     |   %name ASSIGN
49         $VAR '=' $exp
50         { $lhs->{n} = $sym{$VAR->{attr}}->{n} = $exp->{n} }
51
52 ;
53
54 %%
55 # tail code is available at tree construction time
.....
77 }; # end translation scheme
78
.....

```

The node associated with `$_[1]` in

```

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

```

is associated with the whole expression

```
exp <+ ';'>
```

and is a `Calc::_PLUS_LIST` node. When feed with input `a=3;b=4` the children are the two `Calc::ASSIGN` subtrees associated with `a=3` and `b=4` and the `CODE` associated with the semantic action:

```
{ $lhs->{n} = $_[1]->Last_child->{n} }
```

Using `Last_child` we are avoiding the last `CODE` child and setting the `n(umeric)` attribute of the `EXP` node to the one associated with `b=4` (i.e. 4).

```
examples/Node$ trans_scheme_default_action.pl
Write a sequence of arithmetic expressions: a=3;b=4
*****Tree*****
```

```
Calc::EXP(
  Calc::_PLUS_LIST(
    Calc::ASSIGN(
      Calc::TERMINAL,
      Calc::TERMINAL,
      Calc::NUM(
        Calc::TERMINAL,
```

```

        CODE
    ),
    CODE
) # Calc::ASSIGN,
Calc::ASSIGN(
    Calc::TERMINAL,
    Calc::TERMINAL,
    Calc::NUM(
        Calc::TERMINAL,
        CODE
    ),
    CODE
) # Calc::ASSIGN
) # Calc::_PLUS_LIST,
CODE
) # Calc::EXP
*****Symbol table*****
{
    'a' => { 'n' => '3' },
    'b' => { 'n' => '4' }
}

*****Result*****
4

```

## \$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]
)

```

## \$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. To be specific the value returned by the function referenced by `$CLASS_HANDLER` will be dumped. The default value for such function is to return the type of the node. If the node being visited has a method `info` it will be executed and its result inserted between `$DELIMITERS` 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 suppress
$INDENT = 0; # -1 compact, no info, no footnotes
          # 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
$CLASS_HANDLER = sub { type($_[0]) }; # What to print to identify the node

```

Footnotes and attribute info will not be inserted when \$INDENT is -1. A compact representation will be obtained. Such representation can be feed to `new` or `hnew` to obtain a copy of the tree. See the following session with the debugger:

```

pl@nereida:~/LEyapp$ perl -MParse::Eyapp::Node -wde 0
main::(-e:1): 0
DB<1> $x = Parse::Eyapp::Node->new('A(B(C,D),D)', sub { $_->{order} = $i++ for @_; })
DB<2> *A::info = *B::info = *C::info = *D::info = sub { shift()->{order} }
DB<3> p $x->str
A[0](B[1](C[2],D[3]),D[4])
DB<4> $Parse::Eyapp::Node::INDENT=-1
DB<5> p $x->str
A(B(C,D),D)
DB<6> x Parse::Eyapp::Node->hnew($x->str)
0 A=HASH(0x8574704)
  'children' => ARRAY(0x85745d8)
    0 B=HASH(0x857468c)
      'children' => ARRAY(0x8574608)
        0 C=HASH(0x85745b4)
          'children' => ARRAY(0x8509670)
            empty array
          1 D=HASH(0x8574638)
            'children' => ARRAY(0x857450c)
              empty array
            1 D=HASH(0x8574638)
              -> REUSED_ADDRESS
          1 B=HASH(0x857468c)
            -> REUSED_ADDRESS
        2 C=HASH(0x85745b4)
          -> REUSED_ADDRESS
        3 D=HASH(0x8574638)
          -> REUSED_ADDRESS
        4 D=HASH(0x8574638)
          -> REUSED_ADDRESS

```

The following list defines the \$DELIMITERS 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(TERM[2:2]),VAR(TERM[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:

```

 1 int f(int a) {
 2   return 2*a;
 3 }

```

## \$node->equal

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

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

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

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

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

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

```
examples/Node$ cat -n equal.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use Parse::Eyapp::Node;
 4
 5  my $string1 = shift || 'ASSIGN(VAR(TERMINAL))';
 6  my $string2 = shift || 'ASSIGN(VAR(TERMINAL))';
 7  my $t1 = Parse::Eyapp::Node->new($string1, sub { my $i = 0; $_->{n} = $i++ for @_ });
 8  my $t2 = Parse::Eyapp::Node->new($string2);
 9
10  # Without attributes
11  if ($t1->equal($t2)) {
12    print "\nNot considering attributes: Equal\n";
13  }
14  else {
15    print "\nNot considering attributes: Not Equal\n";
16  }
17
18  # Equality with attributes
19  if ($t1->equal($t2, n => sub { return $_[0] == $_[1] }))) {
20    print "\nConsidering attributes: Equal\n";
21  }
22  else {
23    print "\nConsidering attributes: Not Equal\n";
24  }
```

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

```
examples/Node$ equal.pl
Not considering attributes: Equal
Considering attributes: Not Equal
```

## Using equal During Testing

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

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

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

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

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

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

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

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

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

```
examples/Node$ cat -n testequal.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use Parse::Eyapp::Node;
 4  use Data::Dumper;
 5  use Data::Compare;
 6
 7  my $debugging = 0;
 8
 9  my $handler = sub {
10    print Dumper($_[0], $_[1]) if $debugging;
11    Compare($_[0], $_[1])
12  };
13
14  my $t1 = bless( {
15                'types' => {
16                    'CHAR' => bless( { 'children' => [] }, 'CHAR' ),
17                    'VOID' => bless( { 'children' => [] }, 'VOID' ),
```

```

18         'INT' => bless( { 'children' => [] }, 'INT' ),
19         'F(X_0(),INT)' => bless( {
20             'children' => [
21                 bless( { 'children' => [] }, 'X_0' ),
22                 bless( { 'children' => [] }, 'INT' ) ]
23             }, 'F' )
24     },
25     'symboltable' => { 'f' => { 'type' => 'F(X_0(),INT)', 'line' => 1 } },
26     'lines' => 2,
27     'children' => [
28         bless( {
29             'symboltable' => {},
30             'fatherblock' => {},
31             'children' => [],
32             'depth' => 1,
33             'parameters' => [],
34             'function_name' => [ 'f', 1 ],
35             'symboltableLabel' => {},
36             'line' => 1
37         }, 'FUNCTION' )
38     ],
39     'depth' => 0,
40     'line' => 1
41 }, 'PROGRAM' );
42 $t1->{'children'}[0]['fatherblock'] = $t1;
43
44 # Tree similar to $t1 but without some attributes (line, depth, etc.)
45 my $t2 = bless( {
46     'types' => {
47         'CHAR' => bless( { 'children' => [] }, 'CHAR' ),
48         'VOID' => bless( { 'children' => [] }, 'VOID' ),
49         'INT' => bless( { 'children' => [] }, 'INT' ),
50         'F(X_0(),INT)' => bless( {
51             'children' => [
52                 bless( { 'children' => [] }, 'X_0' ),
53                 bless( { 'children' => [] }, 'INT' ) ]
54             }, 'F' )
55     },
56     'symboltable' => { 'f' => { 'type' => 'F(X_0(),INT)', 'line' => 1 } },
57     'children' => [
58         bless( {
59             'symboltable' => {},
60             'fatherblock' => {},
61             'children' => [],
62             'parameters' => [],
63             'function_name' => [ 'f', 1 ],
64         }, 'FUNCTION' )
65     ],
66     }, 'PROGRAM' );
67 $t2->{'children'}[0]['fatherblock'] = $t2;
68
69 # Tree similar to $t1 but without some attributes (line, depth, etc.)
70 # and without the symboltable and types attributes used in the comparison
71 my $t3 = bless( {
72     'types' => {
73         'CHAR' => bless( { 'children' => [] }, 'CHAR' ),
74         'VOID' => bless( { 'children' => [] }, 'VOID' ),
75         'INT' => bless( { 'children' => [] }, 'INT' ),
76         'F(X_0(),INT)' => bless( {
77             'children' => [

```

```

78             bless( { 'children' => [] }, 'X_0' ),
79             bless( { 'children' => [] }, 'INT' ) ]
80         }, 'F' )
81     },
82     'children' => [
83         bless( {
84             'symboltable' => {},
85             'fatherblock' => {},
86             'children' => [],
87             'parameters' => [],
88             'function_name' => [ 'f', 1 ],
89             }, 'FUNCTION' )
90     ],
91     }, 'PROGRAM' );
92
93 $t3->{'children'}[0]{'fatherblock'} = $t2;
94
95 # Without attributes
96 if (Parse::Eyapp::Node::equal($t1, $t2)) {
97     print "\nNot considering attributes: Equal\n";
98 }
99 else {
100     print "\nNot considering attributes: Not Equal\n";
101 }
102
103 # Equality with attributes
104 if (Parse::Eyapp::Node::equal(
105     $t1, $t2,
106     symboltable => $handler,
107     types => $handler,
108 )
109 ) {
110     print "\nConsidering attributes: Equal\n";
111 }
112 else {
113     print "\nConsidering attributes: Not Equal\n";
114 }
115
116 # Equality with attributes
117 if (Parse::Eyapp::Node::equal(
118     $t1, $t3,
119     symboltable => $handler,
120     types => $handler,
121 )
122 ) {
123     print "\nConsidering attributes: Equal\n";
124 }
125 else {
126     print "\nConsidering attributes: Not Equal\n";
127 }

```

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

## `$node->delete`

The `$node->delete($child)` 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, it uses the `delete` and `insert_before` methods:

```
neraida:~/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.

### **`$node->unshift($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

### **`$node->push($newchild)`**

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

### **`$node->insert_before($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

### **`$node->insert_after($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.

## **\$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. (See `examples/Node/TSPostfix3.eypp` for an example).

## **\$node->bud(@transformations)**

Bottom-up decorator. The tree is traversed bottom-up. The set of transformations in `@transformations` is applied to each node in the tree referenced by `$node` in the order supplied by the user. *As soon as one succeeds no more transformations are applied.*

For an example see the files `lib/Simple/Types.eypp` and `lib/Simple/Trans.trg` in `examples/typechecking/Simple-Typing` shows an extract of the type-checking phase of a toy-example compiler:

```
examples/typechecking/Simple-Types-0.4/lib/Simple$ sed -ne '600,613p' Types.eypp
my @typecheck = (      # Check typing transformations for
  our $inum,          # - Numerical constants
  our $charconstant, # - Character constants
  our $bin,           # - Binary Operations
  our $arrays,        # - Arrays
  our $assign,        # - Assignments
  our $control,       # - Flow control sentences
  our $functioncall, # - Function calls
  our $statements,    # - Those nodes with void type
                    # (STATEMENTS, PROGRAM, etc.)
  our $returntype,    # - Return
);

$t->bud(@typecheck);
```

You can find another example of use of `bud` in the file `examples/ParsingStringsAndTrees/infix2pir.pl`

## **4 Parse::Eyapp:YATW Methods**

`Parse::Eyapp:YATW` objects represent tree transformations. They carry the information of what nodes match and how to modify them.

### **Parse::Eyapp:YATW->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 (see the section `The YATW Tree Transformation Call Protocol`). Follows an example (file `examples/12ts_simplify_with_s.pl`):

```
neraida:~/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    # After the shift $_[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 achieved effect 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), 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
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use Rule6;
 4  use Parse::Eyapp::YATW;
 5
 6  my %BinaryOperation = (PLUS=>'+', MINUS => '-', TIMES=>'*', DIV => '/');
 7
 8  sub set_terminfo {
 9    no warnings;
10    *TERMINAL::info = sub { $_[0]{attr} };
11  }
12  sub is_foldable {
13    my ($op, $left, $right);
14    return 0 unless defined($op = $BinaryOperation{ref($_[0])});
15    return 0 unless ($left = $_[0]->child(0), $left->isa('NUM'));
16    return 0 unless ($right = $_[0]->child(1), $right->isa('NUM'));
17
18    my $leftnum = $left->child(0)->{attr};
19    my $rightnum = $right->child(0)->{attr};
20    $left->child(0)->{attr} = eval "$leftnum $op $rightnum";
21    $_[0] = $left;
22  }
23
24  my $parser = new Rule6();
25  $parser->YYData->{INPUT} = "2*3";
26  my $t = $parser->Run;
27  &set_terminfo;
28  print "\n***** Before *****\n";
29  print $t->str;
30  my $p = Parse::Eyapp::YATW->new(PATTERN => \&is_foldable);
31  $p->s($t);
32  print "\n***** After *****\n";
33  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])

```

## The YATW Tree Transformation Call Protocol

For a subroutine `pattern_sub` to work as a YATW tree transformation - as subroutines `is_foldable` and `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.

## Parse::Eyapp::YATW->buildpatterns

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

```

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

```

## \$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
 1 sub not_useful {
 2     my $self = shift; # node
 3     my $pat = $_[2]; # get the YATW object
 4
 5     (ref($self) eq 'CODE') or ((ref($self) eq 'TERMINAL') and ($self->{token} eq $self->{attr}))
 6     or do { return 0 };
 7     $pat->delete();
 8     return 1;
 9 }
10
11 Parse::Eyapp->new_grammar(
12     input=>$translationscheme,
13     classname=>'Calc',
14     firstline =>7,
15 );
16 my $parser = Calc->new();           # Create the parser
17
18 $parser->YYData->{INPUT} = "2*3\n"; print $parser->YYData->{INPUT};
19 my $t = $parser->Run;               # Parse it
20 print $t->str."\n";                # Show the tree
21 my $p = Parse::Eyapp::YATW->new(PATTERN => \&not_useful);
22 $p->s($t);                          # Delete nodes
23 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[2],CODE),TERMINAL[*],NUM(TERMINAL[3],CODE),CODE))
EXP(TIMES(NUM(TERMINAL[2]),NUM(TERMINAL[3])))

```

## \$yatw->unshift

The call `$yatw->unshift($b)` safely unshifts (inserts at the beginning) the node `$b` in the list of its siblings of the node that matched (i.e in the list of siblings of `$_[0]`). The following example shows a YATW transformation `insert_child` that illustrates the use of `unshift` (file `examples/26delete_with_trreereg.pl`):

```

pl@nereida:~/src/perl/YappWithDefaultAction/examples$ \
    sed -ne '70,$p' 26delete_with_trreereg.pl | cat -n
 1 my $transform = Parse::Eyapp::Treeregexp->new( STRING => q{
 2
 3     delete_code : CODE => { $delete_code->delete() }
 4
 5     {
 6         sub not_semantic {
 7             my $self = shift;
 8             return 1 if ((ref($self) eq 'TERMINAL') and ($self->{token} eq $self->{attr}));
 9             return 0;
10        }
11    }
12
13    delete_tokens : TERMINAL and { not_semantic($TERMINAL) } => {
14        $delete_tokens->delete();
15    }
16
17    insert_child : TIMES(NUM(TERMINAL), NUM(TERMINAL)) => {
18        my $b = Parse::Eyapp::Node->new( 'UMINUS(TERMINAL)',
19            sub { $_[1]->{attr} = '4.5' }); # The new node will be a sibling of TIMES
20
21        $insert_child->unshift($b);
22    }
23 },
24 )->generate();
25
26 Parse::Eyapp->new_grammar(
27     input=>$translationscheme,
28     classname=>'Calc',
29     firstline =>7,
30 );
31 my $parser = Calc->new();           # Create the parser
32
33 $parser->YYData->{INPUT} = "2*3\n"; print $parser->YYData->{INPUT}; # Set the input
34 my $t = $parser->Run;               # Parse it
35 print $t->str."\n";                 # Show the tree
36 # Get the AST
37 our ($delete_tokens, $delete_code);
38 $t->s($delete_tokens, $delete_code);
39 print $t->str."\n";                 # Show the tree
40 our $insert_child;
41 $insert_child->s($t);
42 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[2],CODE),TERMINAL[*],NUM(TERMINAL[3],CODE),CODE))

```

```
EXP(TIMES(NUM(TERMINAL[2]),NUM(TERMINAL[3])))
EXP(UMINUS(TERMINAL[4.5]),TIMES(NUM(TERMINAL[2]),NUM(TERMINAL[3])))
```

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

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

it is unsafe.

## `$yatw->insert_before`

A call to `$yatw->insert_before($node)` safely inserts `$node` in the list of siblings of `$_[0]` just before `$_[0]` (i.e. the node that matched with `$yatw`). The following example (see file `examples/YATW/moveinvariantoutofloopcomplex`) 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($x, $e)` subtree - if is loop invariant - will be moved to the list of siblings of `$WHILE` just before the `$WHILE`. Thus a program like

```
"a =1000; c = 1; while (a) { c = c*a; b = 5; a = a-1 }\n"
```

is transformed in s.t. like:

```
"a =1000; c = 1; b = 5; while (a) { c = c*a; a = a-1 }\n"
```

## 5 TREE MATCHING AND TREE SUBSTITUTION

See the documentation in `Parse::Eyapp::treematchingtut`

## 6 SEE ALSO

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

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

- The tutorial *Parsing Strings and Trees with Parse::Eyapp* (An Introduction to Compiler Construction in seven pages) in <http://nereida.deioc.ull.es/~pl/eyasimple/>
- *Parse::Eyapp*, *Parse::Eyapp::eyapplanguageref*, *Parse::Eyapp::debuggingtut*, *Parse::Eyapp::defaultactionsintro*, *Parse::Eyapp::translationschemestut*, *Parse::Eyapp::Driver*, *Parse::Eyapp::Node*, *Parse::Eyapp::YATW*, *Parse::Eyapp::Treeregexp*, *Parse::Eyapp::Scope*, *Parse::Eyapp::Base*, *Parse::Eyapp::datagenerationtut*
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/languageintro.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/debuggingtut.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/eyapplanguageref.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/Treeregexp.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/Node.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/YATW.pdf>

- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/Eyapp.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/Base.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/translationschemestut.pdf>
- The pdf file in <http://nereida.deioc.ull.es/~pl/perlexamples/treematchingtut.pdf>
- perldoc *eyapp*,
- perldoc *treereg*,
- perldoc *vgg*,
- The Syntax Highlight file for vim at [http://www.vim.org/scripts/script.php?script\\_id=2453](http://www.vim.org/scripts/script.php?script_id=2453) and <http://nereida.deioc.ull.es>
- *Analisis Lexico y Sintactico*, (Notes for a course in compiler construction) by Casiano Rodriguez-Leon. Available at <http://nereida.deioc.ull.es/~pl/perlexamples/> Is the more complete and reliable source for Parse::Eyapp. However is in Spanish.
- *Parse::Yapp*,
- Man pages of yacc(1) and bison(1), <http://www.delorie.com/gnu/docs/bison/bison.html>
- *Language::AttributeGrammar*
- *Parse::RecDescent*.
- *HOP::Parser*
- *HOP::Lexer*
- ocaml yacc tutorial at <http://plus.kaist.ac.kr/~shoh/ocaml/ocamllex-ocamyacc/ocamyacc-tutorial/ocamyacc-tutorial.html>

## 7 REFERENCES

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

## 8 CONTRIBUTORS

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

## 9 AUTHOR

Casiano Rodriguez-Leon ([casiano@ull.es](mailto:casiano@ull.es))

## 10 ACKNOWLEDGMENTS

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

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

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

## 11 LICENCE AND COPYRIGHT

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

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

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

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

# Index

`$node->Children`, 9  
`$node->Last child`, 10  
transformations), 20  
`$node->child`, 6  
`$node->children`, 8  
`$node->delete`, 18  
`$node->descendant`, 12  
`$node->equal`, 15  
`$node->insert after($position, $new child)`, 19  
`$node->insert before($position, $new child)`, 19  
`$node->last child`, 10  
`$node->push($newchild)`, 19  
`$node->str`, 12  
`$node->translation scheme`, 20  
`$node->type`, 5  
`$node->unshift($newchild)`, 19  
`$ytw->delete`, 22  
`$ytw->insert before`, 24  
`$ytw->unshift`, 23

ACKNOWLEDGMENTS, 26

AUTHOR, 25

Child Access Through `%tree` alias, 7

CONTRIBUTORS, 25

Directed Acyclic Graphs with `Parse::Eyapp::Node->hnew`, 4

Expanding Directed Acyclic Graphs with `Parse::Eyapp::Node->hexpand`, 4

LICENCE AND COPYRIGHT, 26

METHODS, 2

NAME, 1

`Parse::Eyapp::Node->new`, 2

`Parse::Eyapp::YATW->buildpatterns`, 22

`Parse::Eyapp::YATW->new`, 20

`Parse::Eyapp::YATW` Methods, 20

REFERENCES, 25

SEE ALSO, 24

SYNOPSIS, 1

The YATW Tree Transformation Call Protocol, 22

TREE MATCHING AND TREE SUBSTITUTION,  
24

Using `equal` During Testing, 16