

1 NAME

Parse::Eyapp::Treeregexp - Tree transformations

2 SYNOPSIS

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

my $grammar = q{
    %right  '='
    %left   '- ' '+'
    %left   '* ' '/'
    %left   NEG
    %tree
    %{
    use Tail2; # See file examples/Tail2.pm in the distribution
    %}

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

    exp:    %name NUM
            NUM
            | %name WHILE
              'while'  exp '{' block '}'
            | %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 */
    ;

    %%
}; # end grammar

sub TERMINAL::info { $_[0]{attr} }
$Parse::Eyapp::Node::INDENT = 2;

our (@call,$moveinvariant, $condition, $assign, $before, $after);

Parse::Eyapp->new_grammar(
    input=>$grammar,
    classname=>'Rule6',
    firstline=>7,
);
my $parser = Rule6->new();
my $program = "a =1000; c = 1; while (a) { c = c*a; b = 5; a = a-1 }\n";
my $t = $parser->Run(\$program);
my @output = split /\n/, $t->str;
```


pl@nereida:~/LEyapp/examples\$ moveinvariantoutofloopcomplexformula.pl
PROGRAM

```

-----
Before | After
-----|-----
BLOCK( | BLOCK(
  ASSIGN( | ASSIGN(
    TERMINAL[a], | TERMINAL[a],
    NUM( | NUM(
      TERMINAL[1000] | TERMINAL[1000]
    ) | )
  ) # ASSIGN, | ) # ASSIGN,
  ASSIGN( | ASSIGN(
    TERMINAL[c], | TERMINAL[c],
    NUM( | NUM(
      TERMINAL[1] | TERMINAL[1]
    ) | )
  ) # ASSIGN, | ) # ASSIGN,
  WHILE( | ASSIGN(
    VAR( | TERMINAL[b],
      TERMINAL[a] | NUM(
    ), | TERMINAL[5]
    BLOCK( | )
      ASSIGN( | ) # ASSIGN,
      TERMINAL[c], | WHILE(
      TIMES( | VAR(
        VAR( | TERMINAL[a]
          TERMINAL[c] | ),
        ), | BLOCK(
      ), | ASSIGN(
      VAR( | TERMINAL[c],
        TERMINAL[a] | TIMES(
      ) | VAR(
    ) # TIMES | TERMINAL[a]
  ) # ASSIGN, | ) # TIMES
  ASSIGN( | ) # ASSIGN,
    TERMINAL[b], | ASSIGN(
    NUM( | TERMINAL[a],
      TERMINAL[5] | MINUS(
    ) | VAR(
  ) # ASSIGN, | TERMINAL[a]
  ASSIGN( | ),
    TERMINAL[a], | NUM(
    MINUS( | TERMINAL[1]
      VAR( | )
      TERMINAL[a] | ) # MINUS
    ), | ) # ASSIGN
    NUM( | ) # BLOCK
    TERMINAL[1] | ) # WHILE
  ) # MINUS | ) # BLOCK
  ) # ASSIGN
  ) # BLOCK
) # WHILE
) # BLOCK

```

4 The Treeregexp Language

A Treeregexp program is made of the repetition of three kind of primitives: The treeregexp transformations, supporting 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
  | '*'
```

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`.

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.

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. Thus, in the example:

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

the semantic condition `$x->{attr} == 0` states that the value of the number stored in the `TERMINAL` node referenced by `$x` must be zero.

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, in the example

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

the node being visited `$_[0]` can be also referenced using the lexical variable `$TIMES` which is created by the 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 `$x`. The semantic condition states that the attribute of the node associated with `$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 in the SYNOPSIS 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 `$NUM[0]` references the node that matches the first `NUM` term in the formula and `$NUM[1]` the one that matches the second.

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]` as `= $NUM[0]` as

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

Regexp Treeregexes

The previous `constantfold` example used a classic Perl linear regexp to explicit that the root node of the matching subtree must match the Perl regexp. The general syntax for `REGEXP` treeregexes patterns 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 string oriented regexps is 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 suppress 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 suppress 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`. Variable `$W` (or `@W` if there are more than one `REGEXP` and or dot treeregexes) will be used instead if no identifier is specified.

Scalar Treeregexes

A scalar treeregexp is defined writing a Perl scalar inside the treeregexp, like `$x` in `NUM($x)`. A scalar treeregexp 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.

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

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 suffix of `$A->children`.

The following example uses array treereg expressions to move the assignment `b = 5` out of the `while` loop:

```
.. .....
93 my $program = "a =1000; c = 1; while (a) { c = c*a; b = 5; a = a-1 }\n";
94 $parser->YYData->{INPUT} = $program;
95 my $t = $parser->Run;
96 my @output = split /\n/, $t->str;
97
98 my $p = Parse::Eyapp::Treeregexp->new( STRING => q{
99     moveinvariant: BLOCK(
100         @prests,
101         WHILE(VAR($b), BLOCK(@a, ASSIGN($x, NUM($e)), @c)),
102         @possts
103     )
104     => {
105         my $assign = $ASSIGN;
106         $BLOCK[1]->delete($ASSIGN);
107         $BLOCK[0]->insert_before($WHILE, $assign);
108     }
109 },
110 FIRSTLINE => 99,
111 );
112 $p->generate();
```

Star Treeregexp

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

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 together:

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

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() }

```

5 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
```

- *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>
- The tutorial *Parsing Strings and Trees with Parse::Eyapp* (An Introduction to Compiler Construction in seven pages) in <http://nereida.deioc.ull.es/~pl/eyapsimple/>
- perldoc *eyapp*,
- perldoc *treereg*,
- perldoc *vgg*,
- The Syntax Highlight file for vim at http://www.vim.org/scripts/script.php?script_id=2453 and <http://nereida.deioc.ull.es>
- *Analisis Lexico y Sintactico*, (Notes for a course in compiler construction) by Casiano Rodriguez-Leon. Available at <http://nereida.deioc.ull.es/~pl/perlexamples/> Is the more complete and reliable source for Parse::Eyapp. However is in Spanish.
- *Parse::Yapp*,
- Man pages of yacc(1) 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-ocaml yacc/ocaml yacc-tutorial/ocaml yacc-tutorial.html>

6 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

7 CONTRIBUTORS

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

8 AUTHOR

Casiano Rodriguez-Leon (casiano@ull.es)

9 ACKNOWLEDGMENTS

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

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

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

10 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

ACKNOWLEDGMENTS, 8

Array Treeregexp Expressions, 6

AUTHOR, 8

Code Support, 7

CONTRIBUTORS, 8

Dot Treeregexes, 6

Introduction, 2

LICENCE AND COPYRIGHT, 9

NAME, 1

REFERENCES, 8

Referencing the matching nodes, 5

Regexp Treeregexes, 5

Scalar Treeregexes, 5

SEE ALSO, 7

Semantic condition, 4

Star Treeregexp, 6

SYNOPSIS, 1

The treeregexp, 4

The Treeregexp Language, 4

Transformation code, 5

Transformation Families, 6

Treeregexp rules, 4