

# 1 NAME

Parse::Eyapp::translationschemestut - Introduction to Translation Schemes in Eyapp

## 2 INTRODUCTION

A *translation scheme* scheme is a context free grammar where the right hand sides of the productions have been augmented with semantic actions (i.e. with chunks of Perl code):

```
A -> alpha { action(@_) } beta
```

The analyzer generated by `Parse::Eyapp` executes `action()` after all the semantic actions associated with `alpha` have been executed and before the execution of any of the semantic actions associated with `beta`.

In a translation scheme the embedded actions modify the attributes associated with the symbols of the grammar.

```
A -> alpha { action(@_) } beta
```

*each symbol on the right hand side of a production rule has an associated scalar attribute.* In ordinary `eyapp` programs the attributes of the symbol to the left of `action` are passed as arguments to `action` (in the example, those of `alpha`). These arguments are preceded by a reference to the syntax analyzer object. There is no way inside an ordinary `eyapp` program for an intermediate `action` to access the attributes of the symbols on its right, i.e. those associated with the symbols of `beta`. This restriction is lifted if you use the `%metatree` directive.

Eyapp allows through the `%metatree` directive the creation of *Translation Schemes* where the actions have access to almost any node of the syntax tree.

When using the `%metatree` directive semantic actions aren't immediately executed. Instead they are inserted as nodes of the syntax tree. The main difference with ordinary nodes being that the attribute of such a `CODE` node is a reference to the anonymous subroutine representing the semantic action. The tree is later traversed in depth-first order using the `$t->translation_scheme` method: each time a `CODE` node is visited the action is executed.

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

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

my $ts = q{
  %token FLOAT INTEGER NAME

  %{
    our %s;
  %}

  %metatree

  %%
  D1: D <* ';' '>'
  ;

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

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

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

```

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

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

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

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

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

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

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

$t->translation_scheme;

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

```

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

### 3 EXECUTION STAGES OF A TRANSLATION SCHEME

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

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

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

This stage is called *Class Construction Time*

2. A parser conforming to the generated grammar is built

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

This stage is called *Parser Construction Time*

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

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

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

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

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

This combination of bottom-up parsing with depth first traversing leads to a semantic behavior similar to recursive top-down parsers but with two advantages:

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

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

## 4 THE %begin DIRECTIVE

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

```
pl@nereida:~/LEyapp/examples$ cat -n trans_scheme_default_action.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use Data::Dumper;
 4  use Parse::Eyapp;
 5  use IO::Interactive qw(is_interactive);
 6
 7  my $translationscheme = q{
 8  %{
 9  # head code is available at tree construction time
10  use Data::Dumper;
11  our %sym; # symbol table
12  %}
13
14  %defaultaction {
15      $lhs->{n} = eval " $left->{n} $_[2]->{attr} $right->{n} "
16  }
17
18  %metatree
19
20  %right    '=',
21  %left    '-','+',
22  %left    '*','/'
23
24  %%
25  line:      %name EXP
26              exp <+ ';'> /* Expressions separated by semicolons */
```

```

27         { $lhs->{n} = $_[1]->Last_child->{n} }
28 ;
29
30 exp:
31     %name PLUS
32     exp.left '+' exp.right
33 | %name MINUS
34     exp.left '-' exp.right
35 | %name TIMES
36     exp.left '*' exp.right
37 | %name DIV
38     exp.left '/' exp.right
39 | %name NUM
40     $NUM
41     { $lhs->{n} = $NUM->{attr} }
42 | '(' $exp ')' %begin { $exp }
43 | %name VAR
44     $VAR
45     { $lhs->{n} = $sym{$VAR->{attr}}->{n} }
46 | %name ASSIGN
47     $VAR '=' $exp
48     { $lhs->{n} = $sym{$VAR->{attr}}->{n} = $exp->{n} }
49
50 ;
51
52 %%
53 # tail code is available at tree construction time
54 sub _Error {
55     die "Syntax error.\n";
56 }
57
58 sub _Lexer {
59     my($parser)=shift;
60
61     for ($parser->YYData->{INPUT}) {
62         s/^\s+//;
63         $_ or return('','undef');
64         s/^(([0-9]+(?:\.[0-9]+)?)// and return('NUM',$1);
65         s/^(([A-Za-z][A-Za-z0-9_]*)// and return('VAR',$1);
66         s/^(.)// and return($1,$1);
67     }
68     return('','undef');
69 }
70
71 sub Run {
72     my($self)=shift;
73     return $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error );
74 }
75 }; # end translation scheme
76
77 sub TERMINAL::info { $_[0]->attr }
78
79 my $p = Parse::Eyapp->new_grammar(
80     input=>$translationscheme,
81     classname=>'main',
82     firstline => 6,
83     outputfile => 'main.pm');
84 die $p->qttables() if $p->Warnings;
85 my $parser = main->new();
86 print "Write a sequence of arithmetic expressions: " if is_interactive();

```

```

87 $parser->YYData->{INPUT} = <>;
88 my $t = $parser->Run() or die "Syntax Error analyzing input";
89 $t->translation_scheme;
90
91 $Parse::Eyapp::Node::INDENT = 2;
92 my $treestring = $t->str;
93
94 $Data::Dumper::Indent = 1;
95 $Data::Dumper::Terse = 1;
96 $Data::Dumper::Deepcopy = 1;
97 our %sym;
98 my $symboltable = Dumper(\%sym);
99
100 print <<"EOR";
101 *****Tree*****
102 $treestring
103 *****Symbol table*****
104 $symboltable
105 *****Result*****
106 $t->{n}
107
108 EOR

```

When executed with input `a=2*3;b=a*a` the program produces an output similar to this:

```

pl@nereida:~/LEyapp/examples$ trans_scheme_default_action.pl
Write a sequence of arithmetic expressions: a=2*3;b=a*a
*****Tree*****

```

```

EXP(
  _PLUS_LIST(
    ASSIGN(
      TERMINAL[a],
      TERMINAL[=],
      TIMES(
        NUM(TERMINAL[2], CODE),
        TERMINAL[*],
        NUM(TERMINAL[3], CODE),
        CODE
      ) # TIMES,
      CODE
    ) # ASSIGN,
    ASSIGN(
      TERMINAL[b],
      TERMINAL[=],
      TIMES(
        VAR(TERMINAL[a], CODE),
        TERMINAL[*],
        VAR(TERMINAL[a], CODE),
        CODE
      ) # TIMES,
      CODE
    ) # ASSIGN
  ) # _PLUS_LIST,
  CODE
) # EXP
*****Symbol table*****
{
  'a' => {
    'n' => 6
  },

```

```
'b' => {
  'n' => 36
}
```

```
*****Result*****
36
```

## 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](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](mailto: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](mailto: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, 7

AUTHOR, 7

CONTRIBUTORS, 7

EXECUTION STAGES OF A TRANSLATION SCHEME,  
2

INTRODUCTION, 1

LICENCE AND COPYRIGHT, 7

NAME, 1

REFERENCES, 7

SEE ALSO, 6

THE %begin DIRECTIVE, 3