Parsing Strings and Trees with `Parse::Eyapp`
(An Introduction to Compiler Construction)

Casiano Rodriguez-Leon
Dpto. Estadística, I.O. y Computación
Universidad de La Laguna
La Laguna, 38271, Spain
casiano@ull.es

Abstract

Parse::Eyapp (Extended yapp) is a collection of modules that extends Francois Desarmenien Parse::Yapp 1.05. Eyapp extends yacc/yapp syntax with functionalities like named attributes, EBNF-like expressions, modifiable default action, automatic syntax tree building, semi-automatic abstract syntax tree building, translation schemes, tree regular expressions, tree transformations, scope analysis support, and directed acyclic graphs among others. This article teaches you the basics of Compiler Construction and Parse::Eyapp by building a translator from infix expressions to Parrot Intermediate Representation.

1 Introduction

Almost any reader of this Journal knows what *Parsing* is about. One of the strengths of Perl is its excellence for text analysis. Additionally to its embedded regular expression capacities, modules like `Parse::RecDescent` [1] and `Parse::Yapp` [2] make easier the task of text understanding and text transformation. This is in clear contrast with the absence of Perl 5 generic tools[1] giving support for the subsequent stages of text processing. The exception being the module `Language::AttributeGrammar` [3]. Parrot does well in this chapter, having the Parrot Grammar Engine (PGE) [4] and the Tree Grammar Engine (TGE) [5].

`Parse::Eyapp` (Extended yapp) is a collection of modules that extends Francois Desarmenien `Parse::Yapp 1.05`: Any `yapp` program runs without changes with `eyapp`. Additionally `Parse::Eyapp` provides new functionalities like named attributes, EBNF-like expressions, modifiable default actions, abstract syntax tree building and translation schemes. It also provides a language for tree transformations. This article introduces the basics of translator construction with `Parse::Eyapp` through an example that compiles infix expressions into Parrot Intermediate Representation (PIR)[6]. The input to the program will be a (semicolon separated) list of infix expressions, like:

```
1  b = 5;
2  a = b+2;
3  a = 2*(a+b)*(2-4/2); # is zero
4  print a;
5  d = (a = a+1)*4-b;
6  c = a*b+d;
7  print c;
8  print d
```

[1]There are however very good specific ones, for example, for `XML` and `HTML` support and symbolic mathematics

and the output is an equivalent PIR:

```
.sub 'main' :main
   .local num a, b, c, d
   b = 5
   a = b + 2
   a = 0 # expression at line 3
   print "a = "     # above was
   print a    # reduced to zero
   print "\n" # at compile time
   a = a + 1
   $N5 = a * 4
   d = $N5 - b
   $N7 = a * b
   c = $N7 + d
   print "c = "
   print c
   print "\n"
   print "d = "
   print d
   print "\n"
.end
```

2 The Phases of a Translator

The code below (file `examples/infix2pir.pl`) displays the stages of the translator: *Lexical and syntax analysis, tree transformations and decorations, address assignments, code generation and peephole optimization*. The simplicity of the considered language (no types, no control structures) permits the skipping of *context handling* (also called *semantic analysis*). Context handling includes jobs like *type checking, live analysis*, etc. Don't get overflowed for so much terminology: The incoming sections will explain in more detail each of these phases.

```
my $parser = Infix->new();
```

```
# Set input
$parser->YYData->{INPUT}
  = slurp_file($filename, 'inf');

# Lexical and syntax analysis
my $t = $parser->YYParse(
 yylex => \&Infix::Lex,
 yyerror => \&Infix::Err);

# Tree transformations
$t->s(our @algebra);

# Address assignment
our $reg_assign;
$reg_assign->s($t);

# Code generation
$t->bud(our @translation);
my $dec = build_dec();

peephole_optimization($t->{tr});

output_code(\$t->{tr}, \$dec);
```

The compiler uses the parser for infix expressions that was generated from the Eyapp grammar `Infix.eyp` (see section 4) using the command:

```
$ eyapp Infix.eyp
$ ls -tr | tail -1
Infix.pm
```

It also uses the module containing different families of tree transformations that are described in the `I2PIR.trg` file (explained in sections 5 and 7):

```
$ treereg -m main I2PIR.trg
$ ls -tr | tail -1
I2PIR.pm
$ head -1 I2PIR.pm
package main;
```

The option `-m main` tells `treereg` to place the transformations inside the `main` namespace.

3 Lexical Analysis

Lexical Analysis decomposes the input stream in a sequence of lexical units called *tokens*. Associated with each token is its *attribute* which carries the corresponding information. In the code example below the attribute associated with token `NUM` is its numerical value and the attribute associated with token `VAR` is the actual string. Each time the *parser* requires a new token, the lexer returns the couple `(token, attribute)` that matched. Some tokens - like `PRINT` - do not carry any special information. In such cases, just to keep the protocol simple, the lexer returns the couple `(token, token)`. Using Eyapp terminology such tokens are called *syntactic tokens*. On the other side, *Semantic tokens* are those tokens - like `VAR` or `NUM` - whose attributes transport useful information. When the end of input is reached the lexer returns the couple `('', undef)`.

```
sub Lex {
  my($parser)=shift;

  for ($parser->YYData->{INPUT}) {
    m{\G[ \t]*}gc;
    m{\G\n}gc
      and $lineno++;
    m{\G([0-9]+(?:\.[0-9]+)?)}gc
      and return('NUM',$1);
    m{\Gprint}gc
      and return('PRINT', 'PRINT');
    m{\G([A-Za-z][A-Za-z0-9_]*)}gc
      and return('VAR',$1);
    m{\G(.)}gc
      and return($1,$1);
    return('',undef); # End of input
  }
}
```

Lexical analyzers can have a non negligible impact in the overall performance. Ways to speed up this stage can be found in the works of Simoes [7] and Tambouras [8].

4 Syntax Analysis

The code below shows the body of the grammar (file `Infix.eyp`). Eyapp syntax very much resembles the syntax of old cherished `yacc` [9]. An Eyapp program has three parts: *head*, *body* and *tail*. Each part is separated from the former by the symbol %%. The head section contains declarations, code support and directives. The grammar rules describing the language - and the semantic actions that indicate how evaluate the attributes associated with the symbols - reside in the body section. The tail section includes Perl code that gives support to the semantic actions. Commonly the lexical analyzer and error diagnostic subroutines go there.

```
%right  '='         # Head section
%left   '-' '+'
%left   '*' '/'
%left   NEG
%tree

%%
line:             # Body section
  sts <%name EXPS + ';'>
;
sts:
    %name PRINT
    PRINT leftvalue
  | exp
;
exp:
    %name NUM    NUM
  | %name VAR    VAR
  | %name ASSIGN leftvalue '=' exp
  | %name PLUS   exp '+' exp
  | %name MINUS  exp '-' exp
  | %name TIMES  exp '*' exp
  | %name DIV    exp '/' exp
  | %name NEG
          '-' exp %prec NEG
  |            '(' exp ')'
;
leftvalue : %name VAR VAR
;
%%
...               # tail section
```

4.1 Ambiguities and Conflicts

The former grammar is ambiguous. For instance, an expression like `exp '-' exp` followed by a minus `'-'` can be worked in more than one way. If we have an input like `NUM - NUM - NUM` the activity of a LALR(1) parser (the family of parsers to which Eyapp belongs) consists of a sequence of *shift and reduce actions*. A *shift action* has as consequence the reading of the next token. A *reduce action* is finding a production rule that matches and substituting the *right hand side* (rhs) of the production by the *left hand side* (lhs). For input `NUM - NUM - NUM` the activity will be as follows (the dot is used to indicate where the next input token is):

```
.NUM - NUM - NUM # shift
 NUM.- NUM - NUM # reduce exp: NUM
 exp.- NUM - NUM # shift
 exp -.NUM - NUM # shift
 exp - NUM.- NUM # reduce exp: NUM
 exp - exp.- NUM # shift/reduce conflict
```

up to this point two different decisions can be taken: the next description can be

```
 exp.- NUM # reduce by exp: exp '-' exp
```

or:

```
 exp - exp -.NUM # shift '-'
```

that is called a *shift-reduce conflict*: the parser must decide whether to shift `NUM` or to *reduce* by the rule `exp: exp - exp`.

That is also the reason for the precedence declarations in the head section. Another kind of conflicts are *reduce-reduce conflicts*. They arise when more that rhs can be applied for a reduction action.

By associating priorities with tokens the programmer can tell Eyapp what syntax tree to build in case of *conflict*.

The declarations `%nonassoc`, `%left` and `%right` declare and associate a *priority* with the tokens that follow them. Tokens declared in the same line have the same precedence. Tokens declared in lines below have more precedence than those declared above. Thus, in the example we are saying that `'+'` and `'-'` have the same precedence but higher than `'='`. The final effect of `'-'` having greater precedence than `'='` is that an expression like `a=4-5` is interpreted as `a=(4-5)` and not as `(a=4)-5`. The use of `%left` applied to `'-'` indicates that - in case of ambiguity and a match between precedences - the parser must build the tree corresponding to a left parenthesization. Thus, `4-5-9` is interpreted as `(4-5)-9`.

The `%prec` directive can be used when a rhs is involved in a conflict and has no tokens inside or it has but the precedence of the last token leads to an incorrect interpretation. A rhs can be followed by an optional `%prec token` directive giving the production the precedence of the `token`

```
exp:   '-' exp %prec NEG { -$_[1] }
```

This solves the conflict in `- NUM - NUM` between `(- NUM) - NUM` and `- (NUM - NUM)`. Since `NEG` has more priority than `'-'` the first interpretation will win.

4.2 Building the AST

`Parse::Eyapp` facilitates the construction of abstract syntax trees (AST) through the `%tree` directive. Nodes in the AST are blessed in the production `name`. A rhs can be *named* using the `%name IDENTIFIER` directive. For each *rhs name* a class/package with name `IDENTIFIER` is created.

Symbolic tokens (like `NUM PRINT` or `VAR`) are considered by default *semantic tokens*. String literals (like `'+'`, `'/'`, etc.) are - unless explictly declared using the `semantic token` directive - considered *syntactic tokens*. When building the AST syntactic tokens do not yield new nodes. Semantic tokens however have their own. Thus when feed with input `b=2*a` the generated parser produces the following AST[2]:

```
EXPS(
  ASSIGN(
    VAR(TERMINAL[b]),
    TIMES(
      NUM(TERMINAL[2]),
      VAR(TERMINAL[a]))
  )
)
```

Nodes of the AST are hashes that can be *decorated* with new keys/attributes. The only reserved field is `children` which is a reference to the array of children. Nodes named `TERMINAL` are built from the tokens provided by the lexical analyzer. The couple `($token, $attribute)` returned by the lexical analyzer is stored under the keys `token` and `attr`. `TERMINAL` nodes also have the attribute `children` which is set to an anonymous empty list. Observe the absence of `TERMINAL` nodes corresponding to tokens `'='` and `'*'`. If we change the status of `'*'` and `'='` to `semantic` using the `%semantic token` directive:

```
1   %semantic token '*' '='
2   %right  '='
3   ....  etc.
```

we get a - concrete - syntax tree:

```
EXPS(
  ASSIGN(
    VAR(TERMINAL[b]),
```

[2]The information between brackets shows the attribute for `TERMINAL` nodes

```
    TERMINAL[=],
    TIMES(
      NUM(TERMINAL[2]),
      TERMINAL[*],
      VAR(TERMINAL[a])
    ) # TIMES
  ) # ASSIGN
)
```

Let us now consider the input `2*(a+1)`. The parser yields the tree:

```
EXPS(
  TIMES(
    NUM(
     TERMINAL[2]),
     exp_14(
       PLUS(
         VAR(TERMINAL[a]),
         NUM(TERMINAL[1]))
       ) # PLUS
  ) # TIMES
)
```

Two features are noticeable: the parenthesis rule `exp: '(' exp ')'` had no name and got automatically one: `exp_14`. The *name of a rhs* by default results from concatenating the left hand side of the rule with the ordinal number of the rule[3]. The second is that node `exp_14` is useless and can be suppressed.

The `%tree` directive can be accompanied of the `%bypass` clause. A `%tree bypass` produces an automatic *bypass* of any node with only one child at *tree-construction-time*. A *bypass operation* consists in *returning the only child of the node being visited to the father of the node and re-typing (re-blessing) the node in the name of the production*[4].

Changing the line `%tree` by `%tree bypass` in file `Infix.eyp` we get a more suitable AST for input `2*(a+1)`:

```
EXPS(TIMES(NUM[2],PLUS(VAR[a],NUM[1])))
```

The node `exp_14` has disapeared in this version since the *bypass operation* applies to the rhs of the rule `exp: '(' exp ')'`: Tokens `'('` and `')'` are syntactic tokens and therefore at *tree construction time* only one child is left. Observe also the absence of `TERMINAL` nodes. Bypass clearly applies to rules `exp: NUM` and `exp: VAR` since they have only one element on their rhs. Therefore the `TERMINAL` node is re-blessed as `NUM` and `VAR` respectively.

A consequence of the global scope application of `%tree bypass` is that undesired bypasses may occur. Consider the tree rendered for input `-a*2`:

```
EXPS(TIMES(NEG,NUM))
```

[3]As it appears in the `.output` file. The `.output` file can be generated using the `-v` option of `eyapp`

[4]If the production has an explicit name. Otherwise there is no re-blessing

What happened? The bypass is applied to the rhs `'-' exp`. Though the rhs has two symbols, token `'-'` is a syntactic token and at *tree-construction-time* only `exp` is left. The *bypass* operation applies when building this node. This undesired *bypass* can be avoided applying the `no bypass` directive to the production:

```
 exp : %no bypass NEG
        '-' exp %prec NEG
```

Now the AST for `-a*2` is correct:

```
EXPS(TIMES(NEG(VAR),NUM))
```

Eyapp provides operators `+`, `*` and `?` for the creation of lists and optionals as in:

```
line: sts <EXPS + ';'>
```

which states that a `line` is made of a non empty list of `EXPS` separated by semicolons. By default the class name for such list is `_PLUS_LIST`. The `%name` directive can be used to modify the default name:

```
line: sts <%name EXPS + ';'>
```

Explicit actions can be specified by the programmer. They are managed as anonymous subroutines that receive as arguments the attributes of the symbols in the rule and are executed each time a *reduction* by that rule occurs. When running under the `%tree` directive this provides a mechanism to influence the shape of the AST. Observe however that the grammar in the example is clean of actions: *Parse::Eyapp allowed us to produce a suitable AST without writing any explicit actions*.

5 Tree Transformations

Once we have the AST we can transform it using the *Treeregexp* language. The code below (in file `I2PIR.trg`) shows a set of algebraic tree transformations whose goal is to produce machine independent optimizations.

```
{ #  Example of support code
  use List::Util qw(reduce);
  my %Op = (PLUS=>'+', MINUS => '-',
            TIMES=>'*', DIV => '/');
}
algebra = fold wxz zxw neg;

fold: /TIMES|PLUS|DIV|MINUS/:b(NUM, NUM)
=> {
  my $op = $Op{ref($b)};
  $NUM[0]->{attr} = eval
  "$NUM[0]->{attr} $op $NUM[1]->{attr}";
  $_[0] = $NUM[0];
}
zxw: TIMES(NUM, .) and {$NUM->{attr}==0}
=> { $_[0] = $NUM }
wxz: TIMES(., NUM) and {$NUM->{attr}==0}
=> { $_[0] = $NUM }
neg: NEG(NUM)
=> { $NUM->{attr} = -$NUM->{attr};
     $_[0] = $NUM }
```

A Treeregexp programs is made of *treeregexp* rules that describe what subtrees match and how transform them:

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

A rule has a *name* (wxz in the example), a *term* describing the shape of the subtree to match "TIMES(., NUM)" and two optional fields: a *semantic condition* expliciting the attribute constraints (the code after the reserved word and) and some *transformation code* that tells how to modify the subtree (the code after the big arrow =>). Each rule is translated into a subroutine [5] with name the treerexexp rule *name*. Therefore, after compilation a subroutine wxz will be available. The dot in the *term* TIMES(., NUM) matches any tree. The semantic condition states that the attr entry of node NUM must be zero. The *transformation code* - that will be applied only if the matching succeeded - substitutes the whole subtree by its right child.

References to the nodes associated with some CLASS appearing in the *term* section can be accessed inside the semantic parts through the lexical variable $CLASS. If there is more than one node the associated variable is @CLASS. Variable $_[0] refers to the root of the subtree that matched.

Nodes inside a *term* can be described using linear regular expressions like in the fold transformation:

```
/TIMES|PLUS|DIV|MINUS/:b(NUM, NUM)
```

In such cases an optional identifier to later refer the node that matched can be specified (b in the example).

Tree transformations can be grouped in families:

```
algebra = fold wxz zxw neg;
```

Such families - and the objects they collect - are available inside the client program (read anew the code of the driver in section 2). Thus, if $t holds the AST resulting from the parsing phase, we can call its method s (for substitute) with args the @algebra family:

```
$t->s(our @algebra);
```

The s method of Parse::Eyapp::Node[6] proceeds to apply all the transformation in the family @algebra to tree $t until none of them matches. Thus, for input a = 2*(a+b)*(2-4/2) the parser produces the tree:

```
EXPS(
  ASSIGN(
    VAR[a],
    TIMES(
      TIMES(NUM[2],PLUS(VAR[a],VAR[b])),
      MINUS(NUM[2],DIV(NUM[4],NUM[2])
    )
  )
)
```

which is transformed by the call $t->s(@algebra) in:

```
EXPS(ASSIGN(VAR[a],NUM[0]))
```

[5]The sub must be accessed through a proxy Parse::Eyapp::YATW object. YATW stands for *Yet Another Tree Walker*

[6]All the classes in the AST inherit from Parse::Eyapp::Node

6 Resource Allocation

The back-end of the translator starts with resource assignment. The only resource to consider here is memory. We have to assign a memory location and/or machine register to each of the variables and inner nodes in the AST. The final target machine, Parrot, is a register based interpreter with 32 floating point registers. On top of the Parrot machine is a layer named Parrot Intermediate Representation (PIR). The PIR language and its compiler (imcc) make remarkably easier the task of mapping variables to registers: PIR provides an infinite number of virtual numeric registers named $N1, $N2, etc. and solves the problem of mapping variables into registers via Graph Coloring [10].

```
{{ my $num = 1; # closure
  sub new_N_register {
    return '$N'.$num++;
  }
}}

reg_assign: $x  => {
  if (ref($x) =~ /VAR|NUM/) {
    $x->{reg} = $x->{attr};
    return 1;
  }
  if (ref($x) =~ /ASSIGN/) {
    $x->{reg} = $x->child(0)->{attr};
    return 1;
  }
  $_[0]->{reg} = new_N_register();
}
```

As it shows the code above (in file I2PIR.trg), the resource allocation stage is limited to assign virtual registers to the inner nodes.

A treeregexp term like $x matches any node and creates a lexical variable $x containing a reference to the node that matched.

In between Treeregexp rules the programmer can insert Perl code between curly brackets. The code will be inserted verbatim[7] at that relative point by the treereg compiler.

The Parse::Eyapp::YATW object $reg_assign generated by the compiler is available inside the main driver (revise section 2):

```
our $reg_assign;
$reg_assign->s($t);
```

Now we have an AST *decorated* with a new attribute reg. The following session with the debugger illustrates the way to expose the AST and its attributes:

```
$ perl -wd infix2pir.pl simple5.inf
main::(59): my $filename = shift;
DB<1> c 72
-a*2
EXPS(TIMES(NEG(VAR),NUM))    # The AST
```

[7]Without the outer curly brackets. If it weren't for the second pair of curly brackets the lexical variable $num would be visible up to the end of the file

We have stopped the execution just before the call to `$reg_assign->s($t)`. The AST for input `-a*2` was displayed.

```
main::(72): $reg_assign->s($t);
DB<2> n
main::(75): $t->bud(our @translation);
```

After the register assignment phase the nodes have been decorated with the attribute `$reg`. To display a tree we use the `str` method of `Parse::Eyapp::Node`. The `str` method traverses the syntax tree dumping the type of the node being visited in a string. If the node being visited has a method `info` it will be executed and its result inserted between `$DELIMITER`s into the string. The package variable `$INDENT`[8] controls the way the tree is displayed. Thus, the next three commands display the AST and the values of the `reg` attributes:

```
DB<2> *TIMES::info = *NEG::info = \
*VAR::info=*NUM::info=sub {$_[0]{reg}}
DB<3> $Parse::Eyapp::Node::INDENT=2
DB<4> x $t->str       # Decorated tree
0  '
EXPS(
  TIMES[$N2](
    NEG[$N1](
      VAR[a]
    ),
    NUM[2]
  ) # TIMES
) # EXPS'
```

Observe that no registers were allocated for variables and numbers.

7 Code Generation

The translation is approached as a particular case of *tree decoration*. Each node is decorated with a new attribute - `trans` - that will held the translation for such node. To compute it, we must define transformations for each of the types in the AST:

```
translation = t_num t_var t_op t_neg
              t_assign t_list t_print;
```

Some of these transformations are straightforward:

```
t_num: NUM
  => { $NUM->{tr} = $NUM->{attr} }
t_op:  /TIMES|PLUS|DIV|MINUS/:b($x, $y)
  => {
    my $op = $Op{ref($b)};
    $b->{tr} = "$b->{reg} = $x->{reg}"
                  ." $op $y->{reg}";
  }
```

To keep track of the involved variables a hash is used as a rudimentary symbol table:

```
{ our %s; }
t_assign: ASSIGN($v, $e) => {
  $s{$v->{attr}} = "num";
  $ASSIGN->{tr} = "$v->{reg} = $e->{reg}"
}
```

The translation of the root node (`EXPS`) consists of concatenating the translations of its children:

```
{
  sub cat_trans {
    my $t = shift;

    my $tr = "";
    for ($t->children) {
      (ref($_) =~ m{NUM|VAR|TERMINAL})
        or $tr .= cat_trans($_)."\n"
    }
    $tr .= $t->{tr} ;
  }
}

t_list: EXPS(@S)
  => {
    $EXPS->{tr} = "";
    my @tr = map { cat_trans($_) } @S;
    $EXPS->{tr} =
      reduce { "$a\n$b" } @tr if @tr;
  }
```

The treeregexp `@S` matches the children of the `EXPS` node. The associated lexical variable `@S` contains the references to the nodes that matched.

The method `bud`[9] of `Parse::Eyapp::Node` nodes makes a bootom up traversing of the AST applying to the node being visited the only one transformation that matches[10]. After the call

```
$t->bud(our @translation);
```

the attribute `$t->{trans}` contains a translation to PIR for the whole tree.

8 Peephole Transformations

The name *peephole optimizer* comes from the image of sliding a small window over the target code attempting to replace patterns of instructions by better ones. If we have a look at the code generated in the previous phase for the input `a = 5-b*2` we see that produces:

```
$N1 = b * 2
$N2 = 5 - $N1
a = $N2
```

PIR allows memory instructions involving three arguments like `a = b + c`. This fact and the observation that `$N2` is used only once lead us to conclude that the former translation can be changed to:

[8]Other `Parse::Eyapp::Node` variables governing the behavior of `str` are: `PREFIXES`, `$STRSEP`, `$FOOTNOTE_HEADER`, `$FOOTNOTE_SEP`, `$FOOTNOTE_LEFT`, `$FOOTNOTE_RIGHT` and `$LINESEP`

[9]Bottom-Up Decorator

[10]When `bud` is applied the family of transformations must constitute a *partition* of the AST classes, i.e. for each node one and only one transformation matches

```
$N1 = b * 2
a = 5 - $N1
```

Perl regular expressions constitute a formidable tool to implement *peephole optimization*. The regexp below finds patterns

```
$N# = something
IDENT = $N#
```

and substitutes them by `IDENT = something`:

```
sub peephole_optimization {
  $_[0] =~
   s{(\$N\d+)\s*=\s*(.*\n)\s*
     ([a-zA-Z_]\w*)\s*=\s*\1}
    {$3 = $2}gx;
}
```

9 Output Generation

Emitting the code is the simplest of all the phases. Since Parrot requires all the variables to be declared, a comma separated string `$dec` is built concatenating the keys of the symbol table hash `%s`. The code is then indented and the different components are articulated through a HERE document:

```
sub output_code {
  my ($trans, $dec) = @_;

  # Indent
  $$trans =~ s/^/\t/gm;

  # Output the code
print << "TRANSLATION";
.sub 'main' :main
\t.local num $$dec
$$trans
.end
TRANSLATION
```

The call to `output_code` finishes the job:

```
output_code(\$t->{trans}, \$dec);
```

10 Conclusions and Future Work

There is a shortage of compiler toolkits in CPAN/Perl 5. It will be beneficial to have a CPAN wider covering of translator components: attribute grammars, tree transformations tools and code generator generators (see iburg [11]).

This work presented `Parse::Eyapp`, a work in progress in that direction. `Yacc` and `Parse::Yapp` programmers will feel at home in `Parse::Eyapp`. Additionally to the beneficial mature approach to parsing provided by `Yacc`-like parser generators, `Parse::Eyapp` delivers a set of extensions that give support to the later phases of text processing.

11 About the Author

Casiano Rodriguez-Leon is a Professor of Computer Science at Universidad de La Laguna. His research focuses on Parallel Computing.

12 Acknowledgements

This work has been supported by the EC (FEDER) and by the Spanish Ministry of Education and Science inside the 'Plan Nacional de I+D+i' with contract number TIN2005-08818-C04-04. Thanks To Francois Desarmenien. `Parse::Eyapp` shares a large percentage of code with `Parse::Yapp`.

References

[1] Damian Conway. *Parse::RecDescent, Generate Recursive-Descent Parsers*. CPAN, 2003.

[2] Francois Desarmenien. *Parse::Yapp, Perl extension for generating and using LALR parsers*. CPAN, 2001.

[3] Luke Palmer. *Language::AttributeGrammar, Attribute grammars for doing computations over trees*. CPAN, 2006.

[4] Patrick Michaud. *Parrot Grammar Engine (PGE)*. CPAN, 2007.

[5] Allison Randal. *TGE, A tree grammar engine*. CPAN, 2007.

[6] Allison Randal, Dan Sugalski, and Leopold Toetsch. *Perl 6 and Parrot Essentials, Second Edition*. O'Reilly Media, Inc., 2004.

[7] Alberto Manuel Simoes. Cooking Perl with flex. *The Perl Review*, 0(3), May 2002.

[8] Ioannis Tambouras. *Parse::Flex, The Fastest Lexer in the West*. CPAN, 2006.

[9] Stephen C. Johnson and Ravi Sethi. Yacc: a Parser Generator. *UNIX Vol. II: research system (10th ed.)*, pages 347–374, 1990.

[10] Preston Briggs. Register Allocation via Graph Coloring. Technical Report TR92-183, 24, 1998.

[11] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a Simple, Efficient Code Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, 1992.