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

January 31, 2010

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 abstract syntax tree building, dynamic conflict resolution, 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 using Parse::Eyapp to build a translator from infix expressions to Parrot Intermediate Representation.

1 Introduction

Almost any Perl programmer 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.

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)[3]. Parrot is a virtual machine (VM), similar to the Java VM and the .NET VM. However, unlike these two which are designed for statically-typed languages like Java or C#, Parrot is designed for use with dynamically typed languages such as Perl, Python, Ruby, or PHP.

The input to the program will be a (semicolon separated) list of infix expressions, like in this example located in file `input1.inf`:

```
$ cat input1.inf
b = 5;
```

```
a = b+2;
a = 2*(a+b)*(2-4/2);
print a;
d = (a = a+1)*4-b;
c = a*b+d;
print c;
print d
```

and the output is the PIR resulting from the translation:

```
 1  .sub 'main' :main
 2     .local num a, b, c, d
 3     b = 5
 4     a = b + 2
 5     a = 0 # expression at line 3
 6     print "a = "     # above was
 7     print a    # reduced to zero
 8     print "\n" # at compile time
 9     a = a + 1
10     $N5 = a * 4
11     d = $N5 - b
12     $N7 = a * b
13     c = $N7 + d
14     print "c = "
15     print c
16     print "\n"
17     print "d = "
18     print d
19     print "\n"
20  .end
```

You can download the code for this example from http://nereida.deioc.ull.es/ pl/eyapsimple/source.tgz. To use it, unpack the tarball:

```
tar xvzf source.tgz
```

Change to the directory:

```
cd src
```

and compile the grammar with eyapp:

```
eyapp Infix.eyp
```

Compile also the set of tree transformations using treereg:

```
treereg -m main I2PIR.trg
```

After these two compilations we have two new modules:

```
nereida:/tmp/src> ls -ltr |tail -2
-rw-rw----  1 pl users   Infix.pm
-rw-rw----  1 pl users   I2PIR.pm
```

Module `Infix.pm` contains the parser for the grammar described in `Infix.eyp`. Module `I2PIR.pm` contains the collection of tree transformations described in `I2PIR.trg`. Now we can run the script `infix2pir.pl` which makes use of these two modules:

```
$ ./infix2pir.pl input1.inf > input1.pir
```

We can now make use of the `parrot` interpreter to execute the code:

```
$ /Users/casianorodriguezleon/src/parrot/parrot-1.9.0/parrot input1.pir
a = 0
c = 4
d = -1
```

2 A Fast Introduction to Parrot

The Parrot virtual machine [4] is register based. This means that, like a hardware CPU, it has a number of fast-access units of storage called registers. There are 4 types of register in Parrot: integers (`I`), numbers (`N`), strings (`S`) and PMCs (`P`). For each type there are several of these, named `$N0`, `$N1`, ... Number registers map to the machine native floating point type. You can download a recent version of parrot from http://www.parrot.org/download.

The code produced by `infix2pir.pl` is an example of PIR, which stands for Parrot Intermediate Representation and is also known as Intermediate Code or IMC. PIR files use the extension `.pir`. PIR is an intermediate language that can be compiled to Parrot Byte code (PBC). It was conceived as a possible target language for compilers targeting the Parrot Virtual Machine. PIR is halfway between a High Level Language (HLL) and Parrot Assembly (PASM).

PIR has a relatively simple syntax. Every line is a comment, a label, a statement, or a directive. Each statement or directive stands on its own line. There is no end-of-line symbol (such as a semicolon in C). These is a brief enumeration of the main characteristics of the language:

- **Comments** A comment begins with the # symbol, and continues until the end of the line. Comments can stand alone on a line or follow a statement or directive.
- **Statements**

  A statement is either an opcode or syntactic sugar for one or more opcodes. An opcode is a native instruction for the virtual machine; it consists of the name of the instruction followed by zero or more arguments.
- **Higher-level constructs**

  PIR also provides higher-level constructs, including symbolic operators:

```
$I1 = 2 + 5
```

These special statement forms are just syntactic sugar for regular opcodes. The `+` symbol corresponds to the `add` opcode, the `-` symbol to the `sub` opcode, and so on. The previous example is equivalent to:

```
add $I1, 2, 5
```

- **Directives**

  Directives resemble opcodes, but they begin with a period (`.`). Some directives specify actions that occur at compile time. Other directives represent complex operations that require the generation of multiple instructions. The `.local` directive, for example, declares a named variable.

  ```
  .local string hello
  ```

- **Literals**

  - **Numbers**

    Integers and floating point numbers are numeric literals. They can be positive or negative. Integer literals can also be binary, octal, or hexadecimal:

    ```
    $I0 = 42       # positive
    $I1 = -1       # negative
    $I1 = 0b01010  # binary
    $I2 = 0o72     # octal
    $I3 = 0xA5     # hexadecimal
    ```

    Floating point number literals have a decimal point and can use scientific notation:

    ```
    $N0 = 3.14
    $N2 = -1.2e+4
    ```

  - **Strings**

    String literals are enclosed in single or double-quotes. Strings in double-quotes allow escape sequences using backslashes. Strings in single-quotes only allow escapes for nested quotes

    ```
    $S0 = "This is a valid literal string"
    $S1 = 'This is also a valid literal string'
    ```

- **Variables**

  PIR variables can store four different kinds of valuesintegers, numbers (floating point), strings, and objects. Parrot's objects are called PMCs, for "PolyMorphic Container".

The simplest kind of variable is a register variable. The name of a register variable always starts with a dollar sign (`$`), followed by a single character which specifies the type of the variable – integer (`I`), number (`N`), string (`S`), or PMC (`P`) – and ends with a unique number. You need not predeclare register variables:

```
$S0 = "Who's a pretty boy, then?"
say $S0
```

PIR also has named variables; the `.local` directive declares them. As with register variables, there are four valid types: `int`, `num`, `string`, and `pmc`. You must declare named variables; otherwise they behave exactly the same as register variables.

```
.local string hello
hello = "'Allo, 'allo, 'allo."
say hello
```

- **Constants**

  The `.const` directive declares a named constant. Named constants are similar to named variables, but the values set in the declaration may never change. Like `.local`, `.const` takes a type and a name. It also requires a literal argument to set the value of the constant.

```
.const int    frog = 4                      # integer
.const string name = "Superintendent Parrot" # string
.const num    pi   = 3.14159                # floating point
```

  You may use a named constant anywhere you may use a literal, but you must declare the named constant beforehand.

- **Keys**

  A key is a special kind of constant used for accessing elements in complex variables (such as an array). A key is either an integer or a string; and it's always enclosed in square brackets (`[` and `]`). You do not have to declare literal keys. This code example stores the string `"foo"` in `$P0` as element `5`, and then retrieves it.

```
$P0[5] = "foo"
$S1    = $P0[5]
```

  PIR supports multi-part keys. Use a semicolon to separate each part.

```
$P0['my';'key'] = 472
$I1             = $P0['my';'key']
```

- **Control Structures**
  - **goto**
    The `goto label` statement jumps to a named label. It can only jump inside a subroutine, and only to a named label. Example:

```
    goto GREET
      # ... some skipped code ...
  GREET:
    say "'Allo, 'allo, 'allo."
```

  - **if**
    Variations on the basic `goto` check whether a particular condition is true or false before jumping:

```
  if $I0 > 5 goto GREET
```

- **Subroutines**

  A PIR subroutine starts with the `.sub` directive and ends with the `.end` directive. Parameter declarations use the `.param` directive; they resemble named variable declarations.

  This example declares a subroutine named greeting, that takes a single string parameter named `hello`:

```
.sub 'greeting'
    .param string hello
    say hello
.end
```

3 The Phases of a Translator

The code below (file `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 $filename = shift;
my $parser = Infix->new();

# read input
$parser->slurp_file($filename);

# lexical and syntax analysis
```

```
my $t = $parser->YYParse();

# tree transformations:
# machine independent optimizations
$t->s(our @algebra);

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

# Translate to PARROT
$t->bud(our @translation);

# variable declarations
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 5) 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 6 and 8):

```
$ 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 generated tree transformations inside the `main` namespace. Is in this file that the variables `@algebra`, `@translation` and `$reg_assign` used during the machine-independent optimization, code generation and register allocation phases are defined.

4 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. Each time the *parser* requires a new token, the lexer returns the couple `(token, attribute)` that matched. When the end of input is reached the

lexer returns the couple `('', undef)`. You don't have to write a lexical analyzer: Parse::Eyapp automatically generates one for you using your `%token` definitions (file `Infix.eyp`):

```
%token NUM   = /([0-9]+(?:\.[0-9]+)?)/
%token PRINT = /print\b/
%token VAR   = /([A-Za-z_][A-Za-z0-9_]*)/
```

Here the order is important. The regular expression for `PRINT` will be tried before the regular expression fo `VAR`. The parenthesis are also important. The lexical analyzer built from the regular expression for `VAR` returns `('VAR'. $1)`. Be sure your first memory parenthesis holds the desired attribute.

The lexical analyzer can also be specified through the `%lexer` directive (see the head section in file `InfixWithLexerDirective.eyp` of the `Parse::Eyapp` distribution). The directive `%lexer` is followed by the code of the lexical analyzer. Inside such code the variable `$_` contains the input string. The special variable `$self` refers to the parser object. The pair `('', undef)` is returned by the generated lexer when the end of input is detected.

```
%lexer  {
 m{\G[ \t]*}gc;
 m{\G(\n)+}gc                    and $self->tokenline($1 =~ tr/\n//);
 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);
}
```

In the code example above the attribute associated with token `NUM` is its numerical value and the attribute associated with token `VAR` is the actual string. 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)`.

When feed it with input `b = 1` the lexer will produce the sequence

```
(VAR, 'b') ('=', '=') ('NUM', '1') ('', undef)
```

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 [5] and Tambouras [6].

5 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` [7]. 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. In the case of `eyapp` the lexical analyzer is usually automatically generated from the token definitions or defined through a `%lexer` directive inside the head section. Also, for most cases, there is no need to overwrite the provided default `error` diagnostic method.

```
%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
;
%%
```

5.1 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. An expression like:

```
4 - 3 - 1
```

Is ambiguous. If you can't see it, it is because after so many years in school, your mind has ruled out one of the interpretations. Two interpretations of the former phrase are:

```
(4 - 3) - 1
4 - (3 - 1)
```

In our planet the first interpretation is preferred over the second.

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`. A shift-reduce conflict means that the parser is not in condition to decide whether to associate the processed phrase (left association) or to continue reading more input to make an association later (right association). This incapability usually comes from the fact that the grammar is ambiguous but can also be due to other reasons, as the myopic condition of the parser, being able only to see one token ahead.

Another kind of conflicts are *reduce-reduce conflicts*. They arise when more that rhs can be applied for a reduction action.

The precedence declarations in the head section tells the parser what to do in case of ambiguity.

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.
- The precedence of a rhs is the precedence of the rightmost token in that rhs. Thus, the production named `MINUS`: `exp -> exp '-' exp` has the precedence of `'-'`.
- We can always give an explicit priority to a rhs using the `%prec TOKEN` directive, like in the rhs named `NEG`:

```
| %name NEG
        '-' exp %prec NEG
```

When there is a shift-reduce conflict the precedence of the rule and the precedence of the incoming token are compared

- If the precedence of the rule is greater, a reduction (left association) takes place
- If the precedence of the token is greater, a shift (right association) takes place
- If both token and rule have the same precedence the parser action depends on whether the declaration was made using the `%left` or `%right` directive. In the first case the reduction takes place, in the second the shift predominates.

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`.

As was said, 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.

5.2 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[1]:

```
~/LEyapp/examples/ParsingStringsAndTrees$ perl -wd infix2pir.pl  # start the

Loading DB routines from perl5db.pl version 1.31
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(infix2pir.pl:48):    my $filename = shift;
  DB<1> l 48,55               # let us see the source code ...
48==>   my $filename = shift;
49: my $parser = Infix->new();
50: $parser->slurp_file($filename);
51
52: my $t = $parser->YYParse();
53
54  # Machine independent optimizations
55: $t->s(our @algebra);
  DB<2> c 55  # continue until the abstract syntax tree (AST) is built
b = 2 * a     # <- user input
main::(infix2pir.pl:55):    $t->s(our @algebra);
  DB<3> p $t->str  # show us the AST
line_3(EXPS(sts_5(ASSIGN(VAR(TERMINAL[b]),TIMES(NUM(TERMINAL[2]),VAR(TERMINAL
```

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]),
    TERMINAL[=],
```

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

```
    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[2]. 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*[3].

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))
```

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

[3] 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*.

6 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)};
    croak "Unexpected tree shape: ".$_[0]->str." can't find number in the exp
  unless exists ($NUM[0]->{attr}) && ($NUM[0]->{attr} =~ /^\d+/);

  $NUM[0]->{attr} = eval  "$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 }
```

This rule comes to say

> Wherever you find a node labelled `TIMES` whose right child is a `NUM` node and the value of such `NUM` is zero, *whatever the left child subtree is* proceed to substitute the whole tree by its right child, i.e. by zero.

A rule has a *name* (`wxz` in the example. `wxz` stands for *whatever times zero*), 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 [4] 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;
```

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

Such families - and the objects they collect - are available inside the client program (read anew the code of the driver in section 3). 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`[5] 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 following tree:

```
~/LEyapp/examples/ParsingStringsAndTrees$ perl -wd infix2pir.pl

Loading DB routines from perl5db.pl version 1.31
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(infix2pir.pl:41):    my $filename = shift;
  DB<1> l 41,52                                          # let us remind the
41==>   my $filename = shift;
42: my $parser = Infix->new();
43: $parser->slurp_file($filename);
44
45: my $t = $parser->YYParse() || exit(1);
46
47  # Machine independent optimizations
48: $t->s(our @algebra);
49
50  # Address Assignment
51: our $reg_assign;
52: $reg_assign->s($t);
  DB<2> c 48                                              # get input and bu
a = 2*(a+b)*(2-4/2)
main::(infix2pir.pl:48):    $t->s(our @algebra);
  DB<3> p $t->str
EXPS(ASSIGN(VAR[a],TIMES(TIMES(NUM[2],PLUS(VAR[a],VAR[b])),MINUS(NUM[2],DIV(NU
```

Which is transformed by the call `$t->s(@algebra)` onto this optimized version:

```
  DB<4> n
main::(infix2pir.pl:51):    our $reg_assign;
  DB<4> p $t->str
EXPS(ASSIGN(VAR[a],NUM[0]))

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

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

7 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 [8].

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

```
{{ 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();
}
```

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[6] 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 3):

```
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:

[6] 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

```
~/LEyapp/examples/ParsingStringsAndTrees$ perl -wd infix2pir.pl simple5.i

Loading DB routines from perl5db.pl version 1.31
Editor support available.

Enter h or ‘h h’ for help, or ‘man perldebug’ for more help.

main::(infix2pir.pl:41):    my $filename = shift;
  DB<1> l 41,55
41==>   my $filename = shift;
42: my $parser = Infix->new();
43: $parser->slurp_file($filename);
44
45: my $t = $parser->YYParse() || exit(1);
46
47  # Machine independent optimizations
48: $t->s(our @algebra);
49
50  # Address Assignment
51: our $reg_assign;
52: $reg_assign->s($t);
53
54  # Translate to PARROT
55: $t->bud(our @translation);
  DB<2> c 52
main::(infix2pir.pl:52):    $reg_assign->s($t);
  DB<3> x $t->str
0  'EXPS(TIMES(NEG(VAR[a]),NUM[2]))'
```

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

```
  DB<4> n
main::(infix2pir.pl:55):    $t->bud(our @translation);
```

And have a look at how registers have been allocated:

```
  DB<4> *TIMES::info=*NEG::info=*VAR::info=*NUM::info=sub {$_[0]{reg}}

  DB<5> $Parse::Eyapp::Node::INDENT=2

  DB<6> x $t->str
0  '
EXPS(
  TIMES[$N2](
    NEG[$N1](
      VAR[a]
```

```
    ),
    NUM[2]
  ) # TIMES
) # EXPS'
  DB<7>
```

Observe that no registers were allocated for variables and numbers.

After the register assignment phase the nodes have been decorated with the attribute `$reg`.

To display the tree we use the `str` method of the AST nodes. This method is inherited from 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 `$DELIMITER`s into the string. The Parse::Eyapp::Node variable `$INDENT`[7] controls the way the tree is displayed.

8 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 define a translation transformation `t_class` for each type of node `class` in the AST:

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

Some of these transformations are straightforward. The translation of a `NUM` node is its value:

```
t_num: NUM => { $NUM->{tr} = $NUM->{attr} }
```

The translation of a binary operation node `$b` is to apply the associated binary operator `$op` to the registers `$x->{reg}` and `$y->{reg}` where the operands were stored and store it in the register `$b->{reg}` associated with the node:

```
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}"
}
```

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

The former rule says that the translation of an ASSIGN node consists in assigning the contents of the register assigned to the expression subtree to the register assigned to the left hand side.

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[8] 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[9]. After the call

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

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

9 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:

[8] Bottom-Up Decorator

[9] 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
$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:

```
$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*  # $N#num     = ... something ...
             ([a-zA-Z_]\w*)\s*=\s*\1   # IDENTIFIER = $N#num
            }
            {$3 = $2}gx;               # IDENTIFIER = ... something ...
}
```

10 Output Generation

Emitting the code is the simplest of all 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);
```

11 Conclusions and Future Work

This work presented `Parse::Eyapp`, a bottom-up parser that extends the `Yacc` LALR algorithm and conflict resolution mechanism with dynamic conflict resolution.

`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.

12 About the Author

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

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

[4] The Parrot Group. *Parrot Documentation*. http://docs.parrot.org/, http://www.parrot.org/people.

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

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

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

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