Análisis Léxico y Sintáctico

Casiano R. León [1]

19 de noviembre de 2009

---

[1] DEIOC Universidad de La Laguna

Índice general

Índice de figuras

Índice de cuadros

A Juana

For it is in teaching that we learn
And it is in understanding that we are understood

Agradecimientos/Acknowledgments

I'd like to thank Tom Christiansen, Damian Conway, Simon Cozens, Francois Desarmenien, Richard Foley, Jeffrey E.F. Friedl, Joseph N. Hall, Tim Jennes, Andy Lester, Allison Randall, Randal L. Schwartz, Michael Schwern, Peter Scott, Sriram Srinivasan, Linconl Stein, Dan Sugalski, Leopold Tötsch, Nathan Torkington and Larry Wall for their books and/or their modules. To the Perl Community.

Special thanks to Larry Wall for giving us Perl.

A mis alumnos de Procesadores de Lenguajes en el primer curso de la Ingeniería Superior Informática en la Universidad de La Laguna

Erratas

Por favor, ayudanos a mejorar. Haznos saber cualquier errata que encuentres relativa a los documentos que componen este material didáctico (`casiano@ull.es`).

Capítulo 1

Expresiones Regulares en Perl

1.1. Introducción

Los rudimentos de las expresiones regulares pueden encontrarse en los trabajos pioneros de McCullogh y Pitts (1940) sobre redes neuronales. El lógico Stephen Kleene definió formalmente el algebra que denominó *conjuntos regulares* y desarrollo una notación para la descripción de dichos conjuntos, las *expresiones regulares*.

Durante las décadas de 1960 y 1970 hubo un desarrollo formal de las expresiones regulares. Una de las priemras publicaciones que utilizan las expresiones regulares en un marco informático es el artículo de 1968 de Ken Thompson *Regular Expression Search Algorithm* en el que describe un compilador de expresiones regulares que produce código objeto para un IBM 7094. Este compilador dió lugar al editor *qed*, en el cual se basó el editor de Unix *ed*. Aunque las expresiones regulares de este último no eran tan sofisticadas como las de *qed*, fueron las primeras en ser utilizadas en un contexto no académico. Se dice que el comando global `g` en su formato `g/re/p` que utilizaba para imprimir (opción `p`) las líneas que casan con la expresión regular `re` dió lugar a un programa separado al que se denomino grep.

Las expresiones regulares facilitadas por las primeras versiones de estas herramientas eran limitadas. Por ejemplo, se disponía del cierre de Kleene `*` pero no del cierre positivo `+` o del operador opcional `?`. Por eso, posteriormente, se han introducido los metacaracteres `\+` y `\?`. Existían numerosas limitaciones en dichas versiones, por ej. `$` sólo significa "final de línea" al final de la expresión regular. Eso dificulta expresiones como

```
grep 'cierre$\|^Las' viq.tex
```

Sin embargo, la mayor parte de las versiones actuales resuelven correctamente estos problemas:

```
nereida:~/viq> grep 'cierre$\|^Las' viq.tex
Las expresiones regulares facilitadas por las primeras versiones de estas herramientas
eran limitadas. Por ejemplo, se disponía del cierre de Kleene \verb|*| pero no del cierre
nereida:~/viq>
```

De hecho AT&T Bell labs añadió numerosas funcionalidades, como por ejemplo, el uso de `\{min, max\}`, tomada de *lex*. Por esa época, Alfred Aho escribió *egrep* que, no sólo proporciona un conjunto mas rico de operadores sino que mejoró la implementación. Mientras que el *grep* de Ken Thompson usaba un autómata finito no determinista (NFA), la versión de *egrep* de Aho usa un autómata finito determinista (DFA).

En 1986 Henry Spencer desarrolló la librería *regex* para el lenguaje `C`, que proporciona un conjunto consistente de funciones que permiten el manejo de expresiones regulares. Esta librería ha contribuido a "homogeneizar" la sintáxis y semántica de las diferentes herramientas que utilizan expresiones regulares (como *awk*, *lex*, *sed*, ...).

Véase También

- La sección *Expresiones Regulares en Otros Lenguajes* 1.4

- Regular Expressions Cookbook. Jan Goyvaerts, Steven Levithan
- PCRE (Perl Compatible Regular Expressions) en la Wikipedia
- PCRE (Perl Compatible Regular Expressions)
- Java Regular Expressions
- C# Regular Expressions
- .NET Framework Regular Expressions

1.1.1. Un ejemplo sencillo

Matching en Contexto Escalar

```
pl@nereida:~/Lperltesting$ cat -n c2f.pl
  1    #!/usr/bin/perl -w
  2    use strict;
  3
  4    print "Enter a temperature (i.e. 32F, 100C):\n";
  5    my $input = <STDIN>;
  6    chomp($input);
  7
  8    if ($input !~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i) {
  9      warn "Expecting a temperature, so don’t understand \"$input\".\n";
 10    }
 11    else {
 12      my $InputNum = $1;
 13      my $type = $3;
 14      my ($celsius, $farenheit);
 15      if ($type eq "C" or $type eq "c") {
 16        $celsius = $InputNum;
 17        $farenheit = ($celsius * 9/5)+32;
 18      }
 19      else {
 20        $farenheit = $InputNum;
 21        $celsius = ($farenheit -32)*5/9;
 22      }
 23      printf "%.2f C = %.2f F\n", $celsius, $farenheit;
 24    }
```

Véase también:

- `perldoc perlrequick`
- `perldoc perlretut`
- `perldoc perlre`
- `perldoc perlreref`

Ejecución con el depurador:

```
pl@nereida:~/Lperltesting$ perl -wd c2f.pl
Loading DB routines from perl5db.pl version 1.28
Editor support available.
```

```
Enter h or ‘h h’ for help, or ‘man perldebug’ for more help.
main::(c2f.pl:4):       print "Enter a temperature (i.e. 32F, 100C):\n";
DB<1> c 8
Enter a temperature (i.e. 32F, 100C):
32F
main::(c2f.pl:8):       if ($input !~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i) {
DB<2> n
main::(c2f.pl:12):        my $InputNum = $1;
DB<2> x ($1, $2, $3)
0  32
1  undef
2  ’F’
DB<3> use YAPE::Regex::Explain
DB<4> p YAPE::Regex::Explain->new(’([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$’)->explain
The regular expression:
(?-imsx:([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$)
matches as follows:

NODE                     EXPLANATION
----------------------------------------------------------------------
(?-imsx:                 group, but do not capture (case-sensitive)
                         (with ^ and $ matching normally) (with . not
                         matching \n) (matching whitespace and #
                         normally):
----------------------------------------------------------------------
  (                        group and capture to \1:
----------------------------------------------------------------------
    [-+]?                    any character of: ’-’, ’+’ (optional
                             (matching the most amount possible))
----------------------------------------------------------------------
    [0-9]+                   any character of: ’0’ to ’9’ (1 or more
                             times (matching the most amount
                             possible))
----------------------------------------------------------------------
    (                        group and capture to \2 (optional
                             (matching the most amount possible)):
----------------------------------------------------------------------
      \.                       ’.’
----------------------------------------------------------------------
      [0-9]*                   any character of: ’0’ to ’9’ (0 or
                               more times (matching the most amount
                               possible))
----------------------------------------------------------------------
    )?                       end of \2 (NOTE: because you’re using a
                             quantifier on this capture, only the
                             LAST repetition of the captured pattern
                             will be stored in \2)
----------------------------------------------------------------------
  )                        end of \1
----------------------------------------------------------------------
  \s*                      whitespace (\n, \r, \t, \f, and " ") (0 or
                           more times (matching the most amount
                           possible))
```

```
----------------------------------------------------------------------
  (                        group and capture to \3:
----------------------------------------------------------------------
    [CF]                     any character of: 'C', 'F'
----------------------------------------------------------------------
  )                        end of \3
----------------------------------------------------------------------
  $                        before an optional \n, and the end of the
                           string
----------------------------------------------------------------------
)                        end of grouping
----------------------------------------------------------------------
```

Dentro de una expresión regular es necesario referirse a los textos que casan con el primer, paréntesis, segundo, etc. como `\1`, `\2,` etc. La notación `$1` se refieré a lo que casó con el primer paréntesis en el último *matching*, no en el actual. Veamos un ejemplo:

```
pl@nereida:~/Lperltesting$ cat -n dollar1slash1.pl
   1    #!/usr/bin/perl -w
   2    use strict;
   3
   4    my $a = "hola juanito";
   5    my $b = "adios anita";
   6
   7    $a =~ /(ani)/;
   8    $b =~ s/(adios) *($1)/\U$1 $2/;
   9    print "$b\n";
```

Observe como el `$1` que aparece en la cadena de reemplazo (línea 8) se refiere a la cadena `adios` mientras que el `$1` en la primera parte contiene `ani`:

```
pl@nereida:~/Lperltesting$ ./dollar1slash1.pl
ADIOS ANIta
```

Ejercicio 1.1.1. *Indique cuál es la salida del programa anterior si se sustituye la línea 8 por*

```
$b =~ s/(adios) *(\1)/\U$1 $2/;
```

Número de Paréntesis

El número de paréntesis con memoria no está limitado:

```
pl@nereida:~/Lperltesting$ perl -wde 0
main::(-e:1):   0
            123456789ABCDEF
DB<1> $x = "123456789AAAAAA"
                   1  2  3  4  5  6  7  8  9 10 11  12
DB<2> $r = $x =~ /(.)(.)(.)(.)(.)(.)(.)(.)(.)(.)(.)\11/; print "$r\n$10\n$11\n"
1
A
A
```

Véase el siguiente párrafo de `perlre` (sección Capture buffers):

> *There is no limit to the number of captured substrings that you may use. However Perl also uses* `\10`*,* `\11`*, etc. as aliases for* `\010`*,* `\011`*, etc. (Recall that* `0` *means octal, so* `\011` *is the character at number* `9` *in your coded character set; which would be the 10th*

> *character, a horizontal tab under ASCII.) Perl resolves this ambiguity by interpreting* `\10` *as a backreference only if at least* `10` *left parentheses have opened before it. Likewise* `\11` *is a backreference only if at least* `11` *left parentheses have opened before it. And so on.* `\1` *through* `\9` *are always interpreted as backreferences.*

Contexto de Lista

Si se utiliza en un contexto que requiere una lista, el “pattern match” retorna una lista consistente en las subexpresiones casadas mediante los paréntesis, esto es `$1`, `$2`, `$3`, ... Si no hubiera emparejamiento se retorna la lista vacía. Si lo hubiera pero no hubieran paréntesis se retorna la lista `($&)`.

```
pl@nereida:~/src/perl/perltesting$ cat -n escapes.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3
     4  my $foo = "one two three four five\nsix seven";
     5  my ($F1, $F2, $Etc) = ($foo =~ /^\s*(\S+)\s+(\S+)\s*(.*)/);
     6  print "List Context: F1 = $F1, F2 = $F2, Etc = $Etc\n";
     7
     8  # This is 'almost' the same than:
     9  ($F1, $F2, $Etc) = split(/\s+/, $foo, 3);
    10  print "Split: F1 = $F1, F2 = $F2, Etc = $Etc\n";
```

Observa el resultado de la ejecución:

```
pl@nereida:~/src/perl/perltesting$ ./escapes.pl
List Context: F1 = one, F2 = two, Etc = three four five
Split: F1 = one, F2 = two, Etc = three four five
six seven
```

El modificador `s`

La opción `s` usada en una regexp hace que el punto `'.'` case con el retorno de carro:

```
pl@nereida:~/src/perl/perltesting$ perl -wd ./escapes.pl
main::(./escapes.pl:4): my $foo = "one two three four five\nsix seven";
DB<1> c 9
List Context: F1 = one, F2 = two, Etc = three four five
main::(./escapes.pl:9): ($F1, $F2, $Etc) = split(' ',$foo, 3);
DB<2> ($F1, $F2, $Etc) = ($foo =~ /^\s*(\S+)\s+(\S+)\s*(.*)/s)
DB<3> p "List Context: F1 = $F1, F2 = $F2, Etc = $Etc\n"
List Context: F1 = one, F2 = two, Etc = three four five
six seven
```

La opción `/s` hace que `.` se empareje con un `\n`. Esto es, casa con cualquier carácter.

Veamos otro ejemplo, que imprime los nombres de los ficheros que contienen cadenas que casan con un patrón dado, incluso si este aparece disperso en varias líneas:

```
#!/usr/bin/perl -w
#use:
#smodifier.pl 'expr' files
#prints the names of the files that match with the give expr
undef $/; # input record separator
my $what = shift @ARGV;
while(my $file = shift @ARGV) {
  open(FILE, "<$file");
```

```
    $line =  <FILE>;
    if ($line =~ /$what/s) {
      print "$file\n";
    }
  }
```

Ejemplo de uso:

```
> smodifier.pl 'three.*three' double.in split.pl doublee.pl
double.in
doublee.pl
```

Vea la sección 1.5.2 para ver los contenidos del fichero `double.in`. En dicho fichero, el patrón `three.*three` aparece repartido entre varias líneas.

El modificador `m`

El modificador `s` se suele usar conjuntamente con el modificador `m`. He aquí lo que dice la seccion *Using character classes* de la sección 'Using-character-classes' en `perlretut` al respecto:

> - `m` *modifier* (`//m`)*: Treat string as a set of multiple lines.* `'.'` *matches any character except* `\n`. `^` *and* `$` *are able to match at the start or end of any line within the string.*
> - *both* `s` *and* `m` *modifiers* (`//sm`)*: Treat string as a single long line, but detect multiple lines.* `'.'` *matches any character, even* `\n` . `^` *and* `$` *, however, are able to match at the start or end of any line within the string.*
>
> *Here are examples of //s and //m in action:*
>
> ```
> $x = "There once was a girl\nWho programmed in Perl\n";
>
> $x =~ /^Who/; # doesn't match, "Who" not at start of string
> $x =~ /^Who/s; # doesn't match, "Who" not at start of string
> $x =~ /^Who/m; # matches, "Who" at start of second line
> $x =~ /^Who/sm; # matches, "Who" at start of second line
>
> $x =~ /girl.Who/; # doesn't match, "." doesn't match "\n"
> $x =~ /girl.Who/s; # matches, "." matches "\n"
> $x =~ /girl.Who/m; # doesn't match, "." doesn't match "\n"
> $x =~ /girl.Who/sm; # matches, "." matches "\n"
> ```
>
> *Most of the time, the default behavior is what is wanted, but* `//s` *and* `//m` *are occasionally very useful. If* `//m` *is being used, the start of the string can still be matched with* `\A` *and the end of the string can still be matched with the anchors* `\Z` *(matches both the end and the newline before, like* `$`*), and* `\z` *(matches only the end):*
>
> ```
> $x =~ /^Who/m; # matches, "Who" at start of second line
> $x =~ /\AWho/m; # doesn't match, "Who" is not at start of string
>
> $x =~ /girl$/m; # matches, "girl" at end of first line
> $x =~ /girl\Z/m; # doesn't match, "girl" is not at end of string
>
> $x =~ /Perl\Z/m; # matches, "Perl" is at newline before end
> $x =~ /Perl\z/m; # doesn't match, "Perl" is not at end of string
> ```

Normalmente el carácter `^` casa solamente con el comienzo de la cadena y el carácter `$` con el final. Los `\n` empotrados no casan con `^` ni `$`. El modificador `/m` modifica esta conducta. De este modo `^` y `$` casan con cualquier frontera de línea interna. Las anclas `\A` y `\Z` se utilizan entonces para casar con el comienzo y final de la cadena. Véase un ejemplo:

```
nereida:~/perl/src> perl -de 0
  DB<1> $a = "hola\npedro"
  DB<2> p "$a"
hola
pedro
  DB<3> $a =~ s/.*/x/m
  DB<4> p $a
x
pedro
  DB<5> $a =~ s/^pedro$/juan/
  DB<6> p "$a"
x
pedro
  DB<7> $a =~ s/^pedro$/juan/m
  DB<8>  p "$a"
x
juan
```

El conversor de temperaturas reescrito usando contexto de lista

Reescribamos el ejemplo anterior usando un contexto de lista:

```
casiano@millo:~/Lperltesting$ cat -n c2f_list.pl
#!/usr/bin/perl -w
use strict;

print "Enter a temperature (i.e. 32F, 100C):\n";
my $input = <STDIN>;
chomp($input);

my ($InputNum, $type);

($InputNum, $type) = $input =~ m/^
                                    ([-+]?[0-9]+(?:\.[0-9]*)?) # Temperature
                                    \s*
                                    ([cCfF]) # Celsius or Farenheit
                                 $/x;

die "Expecting a temperature, so don't understand \"$input\".\n" unless defined($InputN

my ($celsius, $fahrenheit);
if ($type eq "C" or $type eq "c") {
  $celsius = $InputNum;
  $fahrenheit = ($celsius * 9/5)+32;
}
else {
  $fahrenheit = $InputNum;
  $celsius = ($fahrenheit -32)*5/9;
}
printf "%.2f C = %.2f F\n", $celsius, $fahrenheit;
```

La opción `x`

La opción `/x` en una regexp permite utilizar comentarios y espacios dentro de la expresión regular. Los espacios dentro de la expresión regular dejan de ser significativos. Si quieres conseguir un espacio que sea significativo, usa `\s` o bien escápalo. Véase la sección ’Modifiers’ en `perlre` y la sección ’Building-a-regexp’ en `perlretut`.

Paréntesis sin memoria

La notación `(?: ... )` se usa para introducir paréntesis de agrupamiento sin memoria. `(?: ...)` Permite agrupar las expresiones tal y como lo hacen los paréntesis ordinarios. La diferencia es que no “memorizan” esto es no guardan nada en `$1`, `$2`, etc. Se logra así una compilación mas eficiente. Veamos un ejemplo:

```
> cat groupingpar.pl
#!/usr/bin/perl

  my $a = shift;

  $a =˜ m/(?:hola )*(juan)/;
  print "$1\n";
nereida:˜/perl/src> groupingpar.pl ’hola juan’
juan
```

Interpolación en los patrones: La opción `o`

El patrón regular puede contener variables, que serán interpoladas (en tal caso, el patrón será recompilado). Si quieres que dicho patrón se compile una sóla vez, usa la opción `/o`.

```
pl@nereida:˜/Lperltesting$ cat -n mygrep.pl
     1  #!/usr/bin/perl -w
     2  my $what = shift @ARGV || die "Usage $0 regexp files ...\n";
     3  while (<>) {
     4    print "File $ARGV, rel. line $.: $_" if (/$what/o); # compile only once
     5  }
     6
```

Sigue un ejemplo de ejecución:

```
pl@nereida:˜/Lperltesting$ ./mygrep.pl
Usage ./mygrep.pl regexp files ...
pl@nereida:˜/Lperltesting$ ./mygrep.pl if labels.c
File labels.c, rel. line 7:        if (a < 10) goto LABEL;
```

El siguiente texto es de la sección ’Using-regular-expressions-in-Perl’ en `perlretut`:

> *If* `$pattern` *won’t be changing over the lifetime of the script, we can add the //o modifier, which directs Perl to only perform variable substitutions once*

Otra posibilidad es hacer una compilación previa usando el operador `qr` (véase la sección ’Regexp-Quote-Like-Operators’ en `perlop`). La siguiente variante del programa anterior también compila el patrón una sóla vez:

```
pl@nereida:˜/Lperltesting$ cat -n mygrep2.pl
     1  #!/usr/bin/perl -w
     2  my $what = shift @ARGV || die "Usage $0 regexp files ...\n";
     3  $what = qr{$what};
     4  while (<>) {
     5    print "File $ARGV, rel. line $.: $_" if (/$what/);
     6  }
```

Véase

- El nodo en perlmonks /o is dead, long live qr//! por diotalevi

Cuantificadores greedy

El siguiente extracto de la sección *Matching Repetitions* en la sección ’Matching-repetitions’ en `perlretut` ilustra la semántica *greedy* de los operadores de repetición `*+{}?` etc.

> *For all of these quantifiers, Perl will try to match as much of the string as possible, while still allowing the regexp to succeed. Thus with* `/a?.../`*, Perl will first try to match the regexp with the a present; if that fails, Perl will try to match the regexp without the a present. For the quantifier* `*` *, we get the following:*
>
> ```
> $x = "the cat in the hat";
> $x =~ /^(.*)(cat)(.*)$/; # matches,
> # $1 = 'the '
> # $2 = 'cat'
> # $3 = ' in the hat'
> ```
>
> *Which is what we might expect, the match finds the only cat in the string and locks onto it. Consider, however, this regexp:*
>
> ```
> $x =~ /^(.*)(at)(.*)$/; # matches,
> # $1 = 'the cat in the h'
> # $2 = 'at'
> # $3 = '' (0 characters match)
> ```
>
> *One might initially guess that Perl would find the* `at` *in* `cat` *and stop there, but that wouldn't give the longest possible string to the first quantifier* `.*`*. Instead, the first quantifier* `.*` *grabs as much of the string as possible while still having the regexp match. In this example, that means having the* `at` *sequence with the final* `at` *in the string.*
>
> *The other important principle illustrated here is that when there are two or more elements in a regexp, the leftmost quantifier, if there is one, gets to grab as much the string as possible, leaving the rest of the regexp to fight over scraps. Thus in our example, the first quantifier* `.*` *grabs most of the string, while the second quantifier* `.*` *gets the empty string. Quantifiers that grab as much of the string as possible are called maximal match or greedy quantifiers.*
>
> *When a regexp can match a string in several different ways, we can use the principles above to predict which way the regexp will match:*
>
> - **Principle 0:** *Taken as a whole, any regexp will be matched at the earliest possible position in the string.*
> - **Principle 1:** *In an alternation* `a|b|c...` *, the leftmost alternative that allows a match for the whole regexp will be the one used.*
> - **Principle 2:** *The maximal matching quantifiers* `?`*,* `*`*,* `+` *and* `{n,m}` *will in general match as much of the string as possible while still allowing the whole regexp to match.*
> - **Principle 3:** *If there are two or more elements in a regexp, the leftmost greedy quantifier, if any, will match as much of the string as possible while still allowing the whole regexp to match. The next leftmost greedy quantifier, if any, will try to match as much of the string remaining available to it as possible, while still allowing the whole regexp to match. And so on, until all the regexp elements are satisfied.*

Regexp y Bucles Infinitos

El siguiente párrafo está tomado de la sección ’Repeated-Patterns-Matching-a-Zero-length-Substring’ en `perlre`:

> *Regular expressions provide a terse and powerful programming language. As with most other power tools, power comes together with the ability to wreak havoc.*
>
> *A common abuse of this power stems from the ability to make infinite loops using regular expressions, with something as innocuous as:*

```
'foo' =~ m{ ( o? )* }x;
```

The `o?` *matches at the beginning of* `'foo'` *, and since the position in the string is not moved by the match,* `o?` *would match again and again because of the* `*` *quantifier.*

Another common way to create a similar cycle is with the looping modifier `//g` *:*

```
@matches = ( 'foo' =~ m{ o? }xg );
```

or

```
print "match: <$&>\n" while 'foo' =~ m{ o? }xg;
```

or the loop implied by `split()`*.*

... Perl allows such constructs, by forcefully breaking the infinite loop. The rules for this are different for lower-level loops given by the greedy quantifiers `*+{}` *, and for higher-level ones like the* `/g` *modifier or* `split()` *operator.*

The lower-level loops are interrupted (that is, the loop is broken) when Perl detects that a repeated expression matched a zero-length substring. Thus

```
m{ (?: NON_ZERO_LENGTH | ZERO_LENGTH )* }x;
```

is made equivalent to

```
m{ (?: NON_ZERO_LENGTH )*
|
(?: ZERO_LENGTH )?
}x;
```

The higher level-loops preserve an additional state between iterations: whether the last match was zero-length. To break the loop, the following match after a zero-length match is prohibited to have a length of zero. This prohibition interacts with backtracking (see Backtracking), and so the second best match is chosen if the best match is of zero length.

For example:

```
$_ = 'bar';
s/\w??/<$&>/g;
```

results in `<><b><><a><><r><>` *. At each position of the string the best match given by non-greedy* `??` *is the zero-length match, and the second best match is what is matched by* `\w` *. Thus zero-length matches alternate with one-character-long matches.*

Similarly, for repeated `m/()/g` *the second-best match is the match at the position one notch further in the string.*

The additional state of being matched with zero-length is associated with the matched string, and is reset by each assignment to `pos()`*. Zero-length matches at the end of the previous match are ignored during* `split`*.*

Ejercicio 1.1.2.

- *Explique la conducta del siguiente matching:*

```
  DB<25> $c = 0

  DB<26>  print(($c++).": <$&>\n") while 'aaaabababab' =~ /a*(ab)*/g;
0: <aaaa>
1: <>
2: <a>
3: <>
4: <a>
5: <>
6: <a>
7: <>
8: <>
```

Cuantificadores *lazy*

Las expresiones *lazy* o *no greedy* hacen que el NFA se detenga en la cadena mas corta que casa con la expresión. Se denotan como sus análogas *greedy* añadiéndole el postfijo `?`:

- `{n,m}?`
- `{n,}?`
- `{n}?`
- `*?`
- `+?`
- `??`

Repasemos lo que dice la sección Matching Repetitions en la sección 'Matching-repetitions' en `perlretut`:

> *Sometimes greed is not good. At times, we would like quantifiers to match a minimal piece of string, rather than a maximal piece. For this purpose, Larry Wall created the minimal match or non-greedy quantifiers* `??` *,*`*?`*,* `+?`*, and* `{}?`*. These are the usual quantifiers with a ? appended to them. They have the following meanings:*
>
> - `a??` *means: match 'a' 0 or 1 times. Try 0 first, then 1.*
> - `a*?` *means: match 'a' 0 or more times, i.e., any number of times, but as few times as possible*
> - `a+?` *means: match 'a' 1 or more times, i.e., at least once, but as few times as possible*
> - `a{n,m}?` *means: match at least n times, not more than m times, as few times as possible*
> - `a{n,}?` *means: match at least n times, but as few times as possible*
> - `a{n}?` *means: match exactly n times. Because we match exactly n times, an? is equivalent to an and is just there for notational consistency.*
>
> *Let's look at the example above, but with minimal quantifiers:*
>
> ```
> 1. $x = "The programming republic of Perl";
> 2. $x =~ /^(.+?)(e|r)(.*)$/; # matches,
> 3. # $1 = 'Th'
> 4. # $2 = 'e'
> 5. # $3 = ' programming republic of Perl'
> ```

The minimal string that will allow both the start of the string `^` *and the alternation to match is* `Th` *, with the alternation* `e|r` *matching* `e`*. The second quantifier* `.*` *is free to gobble up the rest of the string.*

```
$x =~ /(m{1,2}?)(.*?)$/; # matches,
# $1 = ’m’
# $2 = ’ming republic of Perl’
```

The first string position that this regexp can match is at the first `m` *in programming . At this position, the minimal* `m{1,2}?` *matches just one* `m` *. Although the second quantifier* `.*?` *would prefer to match no characters, it is constrained by the end-of-string anchor* `$` *to match the rest of the string.*

```
$x =~ /(.*?)(m{1,2}?)(.*)$/; # matches,
# $1 = ’The progra’
# $2 = ’m’
# $3 = ’ming republic of Perl’
```

In this regexp, you might expect the first minimal quantifier `.*?` *to match the empty string, because it is not constrained by a* `^` *anchor to match the beginning of the word. Principle 0 applies here, however. Because it is possible for the whole regexp to match at the start of the string, it will match at the start of the string. Thus the first quantifier has to match everything up to the first m. The second minimal quantifier matches just one* `m` *and the third quantifier matches the rest of the string.*

```
$x =~ /(.??)(m{1,2})(.*)$/; # matches,
# $1 = ’a’
# $2 = ’mm’
# $3 = ’ing republic of Perl’
```

Just as in the previous regexp, the first quantifier `.??` *can match earliest at position* `a` *, so it does. The second quantifier is greedy, so it matches mm , and the third matches the rest of the string.*

We can modify principle 3 above to take into account non-greedy quantifiers:

- **Principle 3:** *If there are two or more elements in a regexp, the leftmost greedy (non-greedy) quantifier, if any, will match as much (little) of the string as possible while still allowing the whole regexp to match. The next leftmost greedy (non-greedy) quantifier, if any, will try to match as much (little) of the string remaining available to it as possible, while still allowing the whole regexp to match. And so on, until all the regexp elements are satisfied.*

Ejercicio 1.1.3. *Explique cuál será el resultado de el segundo comando de matching introducido en el depurador:*

```
casiano@millo:~/Lperltesting$ perl -wde 0
main::(-e:1):   0
  DB<1> x ('1'x34) =~ m{^(11+)\1+$}
0  11111111111111111
  DB<2> x ('1'x34) =~ m{^(11+?)\1+$}
????????????????????????????????????
```

Descripción detallada del proceso de matching Veamos en detalle lo que ocurre durante un matching. Repasemos lo que dice la sección Matching Repetitions en la sección 'Matching-repetitions' en `perlretut`:

> *Just like alternation, quantifiers are also susceptible to backtracking. Here is a step-by-step analysis of the example*
>
> ```
> 1. $x = "the cat in the hat";
> 2. $x =~ /^(.*)(at)(.*)$/; # matches,
> 3. # $1 = 'the cat in the h'
> 4. # $2 = 'at'
> 5. # $3 = '' (0 matches)
> ```
>
> 1. *Start with the first letter in the string 't'.*
> 2. *The first quantifier '.*' starts out by matching the whole string 'the cat in the hat'.*
> 3. *'a' in the regexp element 'at' doesn't match the end of the string. Backtrack one character.*
> 4. *'a' in the regexp element 'at' still doesn't match the last letter of the string 't', so backtrack one more character.*
> 5. *Now we can match the 'a' and the 't'.*
> 6. *Move on to the third element '.*'. Since we are at the end of the string and '.*' can match 0 times, assign it the empty string.*
> 7. *We are done!*

Rendimiento

La forma en la que se escribe una regexp puede dar lugar agrandes variaciones en el rendimiento. Repasemos lo que dice la sección Matching Repetitions en la sección 'Matching-repetitions' en `perlretut`:

> *Most of the time, all this moving forward and backtracking happens quickly and searching is fast. There are some pathological regexps, however, whose execution time exponentially grows with the size of the string. A typical structure that blows up in your face is of the form*
>
> ```
> /(a|b+)*/;
> ```
>
> *The problem is the nested indeterminate quantifiers. There are many different ways of partitioning a string of length n between the* `+` *and* `*`*: one repetition with* `b+` *of length n, two repetitions with the first* `b+` *length k and the second with length $n - k$, m repetitions whose bits add up to length n, etc.*
>
> *In fact there are an exponential number of ways to partition a string as a function of its length. A regexp may get lucky and match early in the process, but if there is no match, Perl will try every possibility before giving up. So be careful with nested* `*`*'s,* `{n,m}`*'s, and* `+` *'s.*
>
> *The book Mastering Regular Expressions by Jeffrey Friedl [1] gives a wonderful discussion of this and other efficiency issues.*

Eliminación de Comentarios de un Programa C

El siguiente ejemplo elimina los comentarios de un programa `C`.

```
casiano@millo:~/Lperltesting$ cat -n comments.pl
   1    #!/usr/bin/perl -w
   2    use strict;
```

```

my $progname = shift @ARGV or die "Usage:\n$0 prog.c\n";
open(my $PROGRAM,"<$progname") || die "can’t find $progname\n";
my $program = ’’;
{
  local $/ = undef;
  $program = <$PROGRAM>;
}
$program =~ s{
  /\*  # Match the opening delimiter
  .*?  # Match a minimal number of characters
  \*/  # Match the closing delimiter
}[]gsx;

print $program;
```

Veamos un ejemplo de ejecución. Supongamos el fichero de entrada:

```
> cat hello.c
#include <stdio.h>
/* first
comment
*/
main() {
  printf("hello world!\n"); /* second comment */
}
```

Entonces la ejecución con ese fichero de entrada produce la salida:

```
> comments.pl hello.c
#include <stdio.h>

main() {
  printf("hello world!\n");
}
```

Veamos la diferencia de comportamiento entre `*` y `*?` en el ejemplo anterior:

```
pl@nereida:~/src/perl/perltesting$ perl5_10_1 -wde 0
main::(-e:1):   0
  DB<1>  use re ’debug’; ’main() /* 1c */ { /* 2c */ return; /* 3c */ }’ =~ qr{(/\*.*\*/)}; pr
Compiling REx "(/\*.*\*/)"
Final program:
   1: OPEN1 (3)
   3:   EXACT </*> (5)
   5:   STAR (7)
   6:     REG_ANY (0)
   7:   EXACT <*/> (9)
   9: CLOSE1 (11)
  11: END (0)
anchored "/*" at 0 floating "*/" at 2..2147483647 (checking floating) minlen 4
Guessing start of match in sv for REx "(/\*.*\*/)" against "main() /* 1c */ { /* 2c */ return;
Found floating substr "*/" at offset 13...
Found anchored substr "/*" at offset 7...
Starting position does not contradict /^/m...
Guessed: match at offset 7
```

```
Matching REx "(/\*.*\*/)" against "/* 1c */ { /* 2c */ return; /* 3c */ }"
   7 <in() > </* 1c */ {>    |  1:OPEN1(3)
   7 <in() > </* 1c */ {>    |  3:EXACT </*>(5)
   9 <() /*> < 1c */ { />    |  5:STAR(7)
                                 REG_ANY can match 36 times out of 2147483647...
  41 <; /* 3c > <*/ }>       |  7:  EXACT <*/>(9)
  43 <; /* 3c */> < }>       |  9:  CLOSE1(11)
  43 <; /* 3c */> < }>       | 11:  END(0)
Match successful!

/* 1c */ { /* 2c */ return; /* 3c */
Freeing REx: "(/\*.*\*/)"

  DB<2>  use re ’debug’; ’main() /* 1c */ { /* 2c */ return; /* 3c */ }’ =˜ qr{(/\*.*?\*/)}; p
Compiling REx "(/\*.*?\*/)"
Final program:
   1: OPEN1 (3)
   3:   EXACT </*> (5)
   5:   MINMOD (6)
   6:   STAR (8)
   7:     REG_ANY (0)
   8:   EXACT <*/> (10)
  10: CLOSE1 (12)
  12: END (0)
anchored "/*" at 0 floating "*/" at 2..2147483647 (checking floating) minlen 4
Guessing start of match in sv for REx "(/\*.*?\*/)" against "main() /* 1c */ { /* 2c */ return
Found floating substr "*/" at offset 13...
Found anchored substr "/*" at offset 7...
Starting position does not contradict /ˆ/m...
Guessed: match at offset 7
Matching REx "(/\*.*?\*/)" against "/* 1c */ { /* 2c */ return; /* 3c */ }"
   7 <in() > </* 1c */ {>    |  1:OPEN1(3)
   7 <in() > </* 1c */ {>    |  3:EXACT </*>(5)
   9 <() /*> < 1c */ { />    |  5:MINMOD(6)
   9 <() /*> < 1c */ { />    |  6:STAR(8)
                                 REG_ANY can match 4 times out of 4...
  13 <* 1c > <*/ { /* 2c>    |  8:  EXACT <*/>(10)
  15 <1c */> < { /* 2c *>    | 10:  CLOSE1(12)
  15 <1c */> < { /* 2c *>    | 12:  END(0)
Match successful!

/* 1c */
Freeing REx: "(/\*.*?\*/)"

  DB<3>
```

Véase también la documentación en la sección ’Matching-repetitions’ en `perlretut` y la sección ’Quantifiers’ en `perlre`.

Negaciones y operadores *lazy* A menudo las expresiones `X[^X]*X` y `X.*?X`, donde `X` es un carácter arbitrario se usan de forma casi equivalente.

- La primera significa:

  Una cadena que no contiene `X` *en su interior y que está delimitada por* `X`*s*

- La segunda significa:

  Una cadena que comienza en `X` *y termina en la* `X` *mas próxima a la* `X` *de comienzo*

Esta equivalencia se rompe si no se cumplen las hipótesis establecidas.

En el siguiente ejemplo se intentan detectar las cadenas entre comillas dobles que terminan en el signo de exclamación:

```
pl@nereida:~/Lperltesting$ cat -n negynogreedy.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3
     4  my $b = ’Ella dijo "Ana" y yo contesté: "Jamás!". Eso fué todo.’;
     5  my $a;
     6  ($a = $b) =~ s/".*?!"/-$&-/;
     7  print "$a\n";
     8
     9  $b =~ s/"[^"]*!"/-$&-/;
    10  print "$b\n";
```

Al ejecutar el programa obtenemos:

```
> negynogreedy.pl
Ella dijo -"Ana" y yo contesté: "Jamás!"-. Eso fué todo.
Ella dijo "Ana" y yo contesté: -"Jamás!"-. Eso fué todo.
```

Copia y sustitución simultáneas El operador de *binding* `=~` nos permite “asociar” la variable con la operación de casamiento o sustitución. Si se trata de una sustitución y se quiere conservar la cadena, es necesario hacer una copia:

```
$d = $s;
$d =~ s/esto/por lo otro/;
```

en vez de eso, puedes abreviar un poco usando la siguiente “perla”:

```
($d = $s) =~ s/esto/por lo otro/;
```

Obsérvese la asociación por la izquierda del operador de asignación.

Referencias a Paréntesis Previos

Las referencias relativas permiten escribir expresiones regulares mas reciclables. Véase la documentación en la sección ’Relative-backreferences’ en `perlretut`:

> *Counting the opening parentheses to get the correct number for a backreference is error-prone as soon as there is more than one capturing group. A more convenient technique became available with Perl 5.10: relative backreferences. To refer to the immediately preceding capture group one now may write* `\g{-1}` *, the next but last is available via* `\g{-2}`*, and so on.*
>
> *Another good reason in addition to readability and maintainability for using relative backreferences is illustrated by the following example, where a simple pattern for matching peculiar strings is used:*
>
> ```
> 1. $a99a = ’([a-z])(\d)\2\1’; # matches a11a, g22g, x33x, etc.
> ```
>
> *Now that we have this pattern stored as a handy string, we might feel tempted to use it as a part of some other pattern:*

> ```
> $line = "code=e99e";
> if ($line =~ /^(\w+)=$a99a$/){ # unexpected behavior!
>   print "$1 is valid\n";
> } else {
>   print "bad line: ’$line’\n";
> }
> ```
>
> *But this doesn’t match – at least not the way one might expect. Only after inserting the interpolated* `$a99a` *and looking at the resulting full text of the regexp is it obvious that the backreferences have backfired – the subexpression* `(\w+)` *has snatched number 1 and demoted the groups in* `$a99a` *by one rank. This can be avoided by using relative backreferences:*
>
> ```
> $a99a = ’([a-z])(\d)\g{-1}\g{-2}’; # safe for being interpolated
> ```

El siguiente programa ilustra lo dicho:

```
casiano@millo:~/Lperltesting$ cat -n backreference.pl
   1    use strict;
   2    use re ’debug’;
   3
   4    my $a99a = ’([a-z])(\d)\2\1’;
   5    my $line = "code=e99e";
   6    if ($line =~ /^(\w+)=$a99a$/){ # unexpected behavior!
   7      print "$1 is valid\n";
   8    } else {
   9      print "bad line: ’$line’\n";
  10    }
```

Sigue la ejecución:

```
casiano@millo:~/Lperltesting$ perl5.10.1 -wd backreference.pl
main::(backreference.pl:4):     my $a99a = ’([a-z])(\d)\2\1’;
  DB<1> c 6
main::(backreference.pl:6):     if ($line =~ /^(\w+)=$a99a$/){ # unexpected behavior!
  DB<2> x ($line =~ /^(\w+)=$a99a$/)
  empty array
  DB<4> $a99a = ’([a-z])(\d)\g{-1}\g{-2}’
  DB<5> x ($line =~ /^(\w+)=$a99a$/)
0  ’code’
1  ’e’
2  9
```

Usando Referencias con Nombre (Perl 5.10)

El siguiente texto esta tomado de la sección ’Named-backreferences’ en `perlretut`:

> *Perl 5.10 also introduced named capture buffers and named backreferences. To attach a name to a capturing group, you write either* `(?<name>...)` *or* `(?’name’...)`*. The backreference may then be written as* `\g{name}` *.*
>
> *It is permissible to attach the same name to more than one group, but then only the leftmost one of the eponymous set can be referenced. Outside of the pattern a named capture buffer is accessible through the* `%+` *hash.*
>
> *Assuming that we have to match calendar dates which may be given in one of the three formats* `yyyy-mm-dd`*,* `mm/dd/yyyy` *or* `dd.mm.yyyy`*, we can write three suitable patterns where we use* `’d’`*,* `’m’` *and* `’y’` *respectively as the names of the buffers capturing the pertaining components of a date. The matching operation combines the three patterns as alternatives:*

```
$fmt1 = ’(?<y>\d\d\d\d)-(?<m>\d\d)-(?<d>\d\d)’;
$fmt2 = ’(?<m>\d\d)/(?<d>\d\d)/(?<y>\d\d\d\d)’;
$fmt3 = ’(?<d>\d\d)\.(?<m>\d\d)\.(?<y>\d\d\d\d)’;
for my $d qw( 2006-10-21 15.01.2007 10/31/2005 ){
  if ( $d =˜ m{$fmt1|$fmt2|$fmt3} ){
    print "day=$+{d} month=$+{m} year=$+{y}\n";
  }
}
```

> *If any of the alternatives matches, the hash* `%+` *is bound to contain the three key-value pairs.*

En efecto, al ejecutar el programa:

```
casiano@millo:˜/Lperltesting$ cat -n namedbackreferences.pl
     1  use v5.10;
     2  use strict;
     3
     4  my $fmt1 = ’(?<y>\d\d\d\d)-(?<m>\d\d)-(?<d>\d\d)’;
     5  my $fmt2 = ’(?<m>\d\d)/(?<d>\d\d)/(?<y>\d\d\d\d)’;
     6  my $fmt3 = ’(?<d>\d\d)\.(?<m>\d\d)\.(?<y>\d\d\d\d)’;
     7
     8  for my $d qw( 2006-10-21 15.01.2007 10/31/2005 ){
     9    if ( $d =˜ m{$fmt1|$fmt2|$fmt3} ){
    10      print "day=$+{d} month=$+{m} year=$+{y}\n";
    11    }
    12  }
```

Obtenemos la salida:

```
casiano@millo:˜/Lperltesting$ perl5.10.1 -w namedbackreferences.pl
day=21 month=10 year=2006
day=15 month=01 year=2007
day=31 month=10 year=2005
```

Como se comentó:

> *... It is permissible to attach the same name to more than one group, but then only the leftmost one of the eponymous set can be referenced.*

Veamos un ejemplo:

```
pl@nereida:˜/Lperltesting$ perl5.10.1 -wdE 0
main::(-e:1):   0
DB<1>  # ... only the leftmost one of the eponymous set can be referenced
DB<2> $r = qr{(?<a>[a-c])(?<a>[a-f])}
DB<3> print $+{a} if ’ad’ =˜ $r
a
DB<4> print $+{a} if ’cf’ =˜ $r
c
DB<5> print $+{a} if ’ak’ =˜ $r
```

Reescribamos el ejemplo de conversión de temperaturas usando paréntesis con nombre:

```
pl@nereida:~/Lperltesting$ cat -n c2f_5_10v2.pl
 1  #!/usr/local/bin/perl5_10_1 -w
 2  use strict;
 3
 4  print "Enter a temperature (i.e. 32F, 100C):\n";
 5  my $input = <STDIN>;
 6  chomp($input);
 7
 8  $input =~ m/^
 9              (?<farenheit>[-+]?[0-9]+(?:\.[0-9]*)?)\s*[fF]
10              |
11              (?<celsius>[-+]?[0-9]+(?:\.[0-9]*)?)\s*[cC]
12          $/x;
13
14  my ($celsius, $farenheit);
15  if (exists $+{celsius}) {
16    $celsius = $+{celsius};
17    $farenheit = ($celsius * 9/5)+32;
18  }
19  elsif (exists $+{farenheit}) {
20    $farenheit = $+{farenheit};
21    $celsius = ($farenheit -32)*5/9;
22  }
23  else {
24    die "Expecting a temperature, so don't understand \"$input\".\n";
25  }
26
27  printf "%.2f C = %.2f F\n", $celsius, $farenheit;
```

La función `exists` retorna verdadero si existe la clave en el hash y falso en otro caso.

Grupos con Nombre y Factorización

El uso de nombres hace mas robustas y mas factorizables las expresiones regulares. Consideremos la siguiente regexp que usa notación posicional:

```
pl@nereida:~/Lperltesting$ perl5.10.1 -wde 0
main::(-e:1):   0
  DB<1> x "abbacddc" =~ /(.)(.)\2\1/
0  'a'
1  'b'
```

Supongamos que queremos reutilizar la regexp con repetición

```
  DB<2> x "abbacddc" =~ /((.)(.)\2\1){2}/
  empty array
```

¿Que ha ocurrido? La introducción del nuevo paréntesis nos obliga a renombrar las referencias a las posiciones:

```
  DB<3> x "abbacddc" =~ /((.)(.)\3\2){2}/
0  'cddc'
1  'c'
2  'd'
  DB<4> "abbacddc" =~ /((.)(.)\3\2){2}/; print "$&\n"
abbacddc
```

Esto no ocurre si utilizamos nombres. El operador `\k<a>` sirve para hacer referencia al valor que ha casado con el paréntesis con nombre `a`:

```
  DB<5> x "abbacddc" =~ /((?<a>.)(?<b>.)\k<b>\k<a>){2}/
0  ’cddc’
1  ’c’
2  ’d’
```

El uso de grupos con nombre y `\k`[1] en lugar de referencias numéricas absolutas hace que la regexp sea mas reutilizable.

LLamadas a expresiones regulares via paréntesis con memoria

Es posible también llamar a la expresión regular asociada con un paréntesis.
Este parrafo tomado de la sección ’Extended-Patterns’ en `perlre` explica el modo de uso:
`(?PARNO) (?-PARNO) (?+PARNO) (?R) (?0)`

> `PARNO` *is a sequence of digits (not starting with 0) whose value reflects the paren-number of the capture buffer to recurse to.*
>
> *....*
>
> *Capture buffers contained by the pattern will have the value as determined by the outermost recursion. ....*
>
> *If* `PARNO` **is preceded by a plus or minus sign then it is assumed to be relative, with negative numbers indicating preceding capture buffers and positive ones following**. *Thus* `(?-1)` *refers to the most recently declared buffer, and* `(?+1)` *indicates the next buffer to be declared.*
>
> **Note that the counting for relative recursion differs from that of relative backreferences, in that with recursion unclosed buffers are included.**

Veamos un ejemplo:

```
casiano@millo:~/Lperltesting$ perl5.10.1 -wdE 0
main::(-e:1):   0
  DB<1> x "AABB" =~ /(A)(?-1)(?+1)(B)/
0  ’A’
1  ’B’
  # Parenthesis:       1   2   2                  1
  DB<2> x ’ababa’ =~ /^((?:([ab])(?1)\g{-1}|[ab]?))$/
0  ’ababa’
1  ’a’
  DB<3> x ’bbababab’ =~ /^((?:([ab])(?1)\g{-1}|[ab]?))$/
0  ’bbababab’
1  ’b’
```

Véase también:

- Perl Training Australia: Regular expressions in Perl 5.10
- Perl 5.10 Advanced Regular Expressions by Yves Orton
- Gabor: Regular Expressions in Perl 5.10

---

[1] Una diferencia entre *k* y *g* es que el primero sólo admite un nombre como argumento mientras que *g* admite enteros

Reutilizando Expresiones Regulares

La siguiente reescritura de nuestro ejemplo básico utiliza el módulo Regexp::Common para factorizar la expresión regular:

```
casiano@millo:~/src/perl/perltesting$ cat -n c2f_5_10v3.pl
#!/soft/perl5lib/bin/perl5.10.1 -w
use strict;
use Regexp::Common;

print "Enter a temperature (i.e. 32F, 100C):\n";
my $input = <STDIN>;
chomp($input);

$input =~ m/^
            (?<farenheit>$RE{num}{real})\s*[fF]
            |
            (?<celsius>$RE{num}{real})\s*[cC]
         $/x;

my ($celsius, $farenheit);
if ('celsius' ~~ %+) {
  $celsius = $+{celsius};
  $farenheit = ($celsius * 9/5)+32;
}
elsif ('farenheit' ~~ %+) {
  $farenheit = $+{farenheit};
  $celsius = ($farenheit -32)*5/9;
}
else {
  die "Expecting a temperature, so don't understand \"$input\".\n";
}

printf "%.2f C = %.2f F\n", $celsius, $farenheit;
```

Véase:

- La documentación del módulo Regexp::Common por Abigail
- Smart Matching: Perl Training Australia: Smart Match
- Rafael García Suárez: la sección 'Smart-matching-in-detail' en `perlsyn`
- Enrique Nell (Barcelona Perl Mongers): Novedades en Perl 5.10

El Módulo Regexp::Common

El módulo Regexp::Common provee un extenso número de expresiones regulares que son accesibles vía el hash `%RE`. sigue un ejemplo de uso:

```
casiano@millo:~/Lperltesting$ cat -n regexpcommonsynopsis.pl
use strict;
use Perl6::Say;
use Regexp::Common;

while (<>) {
    say q{a number}              if /$RE{num}{real}/;

```

```
    say q{a [’"‘] quoted string} if /$RE{quoted}/;

    say q{a /.../ sequence}      if m{$RE{delimited}{’-delim’=>’/’}};

    say q{balanced parentheses}  if /$RE{balanced}{’-parens’=>’()’}/;

    die q{a #*@%-ing word}."\n"  if /$RE{profanity}/;

}

```

Sigue un ejemplo de ejecución:

```
casiano@millo:~/Lperltesting$ perl regexpcommonsynopsis.pl
43
a number
"2+2 es" 4
a number
a [’"‘] quoted string
x/y/z
a /.../ sequence
(2*(4+5/(3-2)))
a number
balanced parentheses
fuck you!
a #*@%-ing word
```

El siguiente fragmento de la documentación de Regexp::Common explica el modo simplificado de uso:

> *To access a particular pattern,* `%RE` *is treated as a hierarchical hash of hashes (of hashes...), with each successive key being an identifier. For example, to access the pattern that matches real numbers, you specify:*
>
> ```
> $RE{num}{real}
> ```
>
> *and to access the pattern that matches integers:*
>
> ```
> $RE{num}{int}
> ```
>
> *Deeper layers of the hash are used to specify flags: arguments that modify the resulting pattern in some way.*
>
> - *The keys used to access these layers are prefixed with a minus sign and may have a value;*
> - *if a value is given, it's done by using a multidimensional key.*
>
> *For example, to access the pattern that matches base-2 real numbers with embedded commas separating groups of three digits (e.g.* `10,101,110.110101101`*):*
>
> ```
> $RE{num}{real}{-base => 2}{-sep => ’,’}{-group => 3}
> ```
>
> *Through the magic of Perl, these flag layers may be specified in any order (and even interspersed through the identifier keys!) so you could get the same pattern with:*

```
$RE{num}{real}{-sep => ’,’}{-group => 3}{-base => 2}
```

or:

```
$RE{num}{-base => 2}{real}{-group => 3}{-sep => ’,’}
```

or even:

```
$RE{-base => 2}{-group => 3}{-sep => ’,’}{num}{real}
```

etc.

Note, however, that the relative order of amongst the identifier keys is significant. That is:

```
$RE{list}{set}
```

would not be the same as:

```
$RE{set}{list}
```

Veamos un ejemplo con el depurador:

```
casiano@millo:~/Lperltesting$ perl -MRegexp::Common -wde 0
main::(-e:1):   0
  DB<1> x ’numero: 10,101,110.110101101 101.1e-1 234’ =~ m{($RE{num}{real}{-base => 2}{-sep =>
0  ’10,101,110.110101101’
1  ’101.1e-1’
```

La expresión regular para un número real es relativamente compleja:

```
casiano@millo:~/src/perl/perltesting$ perl5.10.1 -wd c2f_5_10v3.pl
main::(c2f_5_10v3.pl:5):     print "Enter a temperature (i.e. 32F, 100C):\n";
  DB<1> p $RE{num}{real}
(?:(?i)(?:[+-]?)(?:(?=[0123456789]|[.])(?:[0123456789]*)(?:(?:[.])(?:[0123456789]{0,}))?)(?:(?
```

Si se usa la opción `-keep` el patrón proveído usa paréntesis con memoria:

```
casiano@millo:~/Lperltesting$ perl -MRegexp::Common -wde 0
main::(-e:1):   0
DB<2> x ’one, two, three, four, five’ =~ /$RE{list}{-pat => ’\w+’}/
0  1
DB<3> x ’one, two, three, four, five’ =~ /$RE{list}{-pat => ’\w+’}{-keep}/
0  ’one, two, three, four, five’
1  ’, ’
```

Smart Matching

Perl 5.10 introduce el operador de smart matching. El siguiente texto es tomado casi verbatim del site de la compañía Perl Training Australia[2]:

---

[2] This Perl tip and associated text is copyright Perl Training Australia

Perl 5.10 introduces a new-operator, called smart-match, written `~~`*. As the name suggests, smart-match tries to compare its arguments in an intelligent fashion. Using smart-match effectively allows many complex operations to be reduces to very simple statements.*

Unlike many of the other features introduced in Perl 5.10, there's no need to use the feature pragma to enable smart-match, as long as you're using 5.10 it's available.

The smart-match operator is always commutative. That means that `$x ~~ $y` *works the same way as* `$y ~~ $x`*. You'll never have to remember which order to place to your operands with smart-match. Smart-match in action.*

As a simple introduction, we can use smart-match to do a simple string comparison between simple scalars. For example:

```
use feature qw(say);

my $x = "foo";
my $y = "bar";
my $z = "foo";

say '$x and $y are identical strings' if $x ~~ $y;
say '$x and $z are identical strings' if $x ~~ $z;    # Printed
```

If one of our arguments is a number, then a numeric comparison is performed:

```
my $num   = 100;
my $input = <STDIN>;

say 'You entered 100' if $num ~~ $input;
```

This will print our message if our user enters 100, 100.00, +100, 1e2, or any other string that looks like the number 100.

We can also smart-match against a regexp:

```
my $input  = <STDIN>;

say 'You said the secret word!' if $input ~~ /xyzzy/;
```

Smart-matching with a regexp also works with saved regexps created with qr.

So we can use smart-match to act like eq, `==` *and* `=~`*, so what? Well, it does much more than that.*

We can use smart-match to search a list:

```
casiano@millo:~/Lperltesting$ perl5.10.1 -wdE 0
main::(-e:1):   0
  DB<1> @friends = qw(Frodo Meriadoc Pippin Samwise Gandalf)
  DB<2> print "You're a friend" if 'Pippin' ~~ @friends
You're a friend
  DB<3> print "You're a friend" if 'Mordok' ~~ @friends
```

It's important to note that searching an array with smart-match is extremely fast. It's faster than using grep, it's faster than using `first` *from Scalar::Util, and it's faster than walking through the loop with* `foreach`*, even if you do know all the clever optimisations.*

Esta es la forma típica de buscar un elemento en un array en versiones anteriores a la 5.10:

```
casiano@millo:~$ perl -wde 0
main::(-e:1):   0
  DB<1> use List::Util qw{first}
  DB<2> @friends = qw(Frodo Meriadoc Pippin Samwise Gandalf)
  DB<3> x first { $_ eq 'Pippin'} @friends
0  'Pippin'
  DB<4> x first { $_ eq 'Mordok'} @friends
0  undef
```

We can also use smart-match to compare arrays:

```
  DB<4> @foo = qw(x y z xyzzy ninja)
  DB<5> @bar = qw(x y z xyzzy ninja)
  DB<7> print "Identical arrays" if @foo ~~ @bar
Identical arrays
  DB<8> @bar = qw(x y z xyzzy nOnjA)
  DB<9> print "Identical arrays" if @foo ~~ @bar
  DB<10>
```

And even search inside an array using a string:

```
 DB<11> x @foo = qw(x y z xyzzy ninja)
0  'x'
1  'y'
2  'z'
3  'xyzzy'
4  'ninja'
  DB<12> print "Array contains a ninja " if @foo ~~ 'ninja'
```

or using a regexp:

```
  DB<13> print "Array contains magic pattern" if @foo ~~ /xyz/
Array contains magic pattern
  DB<14> print "Array contains magic pattern" if @foo ~~ /\d+/
```

Smart-match works with array references, too[3]*:*

```
  DB<16> $array_ref = [ 1..10 ]
  DB<17> print "Array contains 10" if 10 ~~ $array_ref
Array contains 10
  DB<18> print "Array contains 10" if $array_ref ~~ 10
  DB<19>
```

En el caso de un número y un array devuelve cierto si el escalar aparece en un array anidado:

```
casiano@millo:~/Lperltesting$ perl5.10.1 -E 'say "ok" if 42 ~~  [23, 17, [40..50], 70];'
ok
casiano@millo:~/Lperltesting$ perl5.10.1 -E 'say "ok" if 42 ~~  [23, 17, [50..60], 70];'
casiano@millo:~/Lperltesting$
```

Of course, we can use smart-match with more than just arrays and scalars, it works with searching for the key in a hash, too!

---

[3] En este caso la conmutatividad no funciona

```
  DB<19> %colour = ( sky   => ’blue’, grass => ’green’, apple => ’red’,)
  DB<20> print "I know the colour" if ’grass’ ~~ %colour
I know the colour
  DB<21> print "I know the colour" if ’cloud’ ~~ %colour
  DB<22>
  DB<23> print "A key starts with ’gr’" if %colour ~~ /^gr/
A key starts with ’gr’
  DB<24> print "A key starts with ’clou’" if %colour ~~ /^clou/
  DB<25>
```

> *You can even use it to see if the two hashes have identical keys:*

```
  DB<26> print ’Hashes have identical keys’ if %taste ~~ %colour;
Hashes have identical keys
```

La conducta del operador de smart matching viene dada por la siguiente tabla tomada de la sección ’Smart-matching-in-detail’ en `perlsyn`:

> *The behaviour of a smart match depends on what type of thing its arguments are. The behaviour is determined by the following table: the first row that applies determines the match behaviour (which is thus mostly determined by the type of the right operand). Note that the smart match implicitly dereferences any non-blessed hash or array ref, so the ”Hash.^{a}nd .^{A}rray.^{en}tries apply in those cases. (For blessed references, the .^{O}bject.^{en}tries apply.)*
>
> *Note that the ”Matching Codeçolumn is not always an exact rendition. For example, the smart match operator short-circuits whenever possible, but grep does not.*

```
 $a      $b        Type of Match Implied    Matching Code
 ======  =====     =====================    =============
 Any     undef     undefined                !defined $a

 Any     Object    invokes ~~ overloading on $object, or dies

 Hash    CodeRef   sub truth for each key[1] !grep { !$b->($_) } keys %$a
 Array   CodeRef   sub truth for each elt[1] !grep { !$b->($_) } @$a
 Any     CodeRef   scalar sub truth          $b->($a)

 Hash    Hash      hash keys identical (every key is found in both hashes)
 Array   Hash      hash slice existence     grep { exists $b->{$_} } @$a
 Regex   Hash      hash key grep            grep /$a/, keys %$b
 undef   Hash      always false (undef can’t be a key)
 Any     Hash      hash entry existence     exists $b->{$a}

 Hash    Array     hash slice existence     grep { exists $a->{$_} } @$b
 Array   Array     arrays are comparable[2]
 Regex   Array     array grep               grep /$a/, @$b
 undef   Array     array contains undef     grep !defined, @$b
 Any     Array     match against an array element[3]
                                            grep $a ~~ $_, @$b

 Hash    Regex     hash key grep            grep /$b/, keys %$a
 Array   Regex     array grep               grep /$b/, @$a
 Any     Regex     pattern match            $a =~ /$b/
```

```
Object  Any       invokes ~~ overloading on $object, or falls back:
Any     Num       numeric equality         $a == $b
Num     numish[4] numeric equality         $a == $b
undef   Any       undefined                !defined($b)
Any     Any       string equality          $a eq $b
```

Ejercicios

Ejercicio 1.1.4.

- *Indique la salida del siguiente programa:*

```
pl@nereida:~/Lperltesting$ cat twonumbers.pl
$_ = "I have 2 numbers: 53147";
@pats = qw{
  (.*)(\d*)
  (.*)(\d+)
  (.*?)(\d*)
  (.*?)(\d+)
  (.*)(\d+)$
  (.*?)(\d+)$
  (.*)\b(\d+)$
  (.*\D)(\d+)$
};

print "$_\n";
for $pat (@pats) {
  printf "%-12s ", $pat;
  <>;
  if ( /$pat/ ) {
    print "<$1> <$2>\n";
  } else {
    print "FAIL\n";
  }
}
```

1.1.2. Depuración de Expresiones Regulares

Para obtener información sobre la forma en que es compilada una expresión regular y como se produce el proceso de matching podemos usar la opción `'debug'` del módulo `re`. La versión de Perl 5.10 da una información algo mas legible que la de las versiones anteriores:

```
pl@nereida:~/Lperltesting$ perl5_10_1 -wde 0

Loading DB routines from perl5db.pl version 1.32
Editor support available.

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

main::(-e:1):   0
  DB<1> use re 'debug'; 'astr' =~ m{[sf].r}
Compiling REx "[sf].r"
Final program:
   1: ANYOF[fs][] (12)
  12: REG_ANY (13)
  13: EXACT <r> (15)
```

```
  15: END (0)
anchored "r" at 2 (checking anchored) stclass ANYOF[fs][] minlen 3
Guessing start of match in sv for REx "[sf].r" against "astr"
Found anchored substr "r" at offset 3...
Starting position does not contradict /^/m...
start_shift: 2 check_at: 3 s: 1 endpos: 2
Does not contradict STCLASS...
Guessed: match at offset 1
Matching REx "[sf].r" against "str"
   1 <a> <str>              |  1:ANYOF[fs][](12)
   2 <as> <tr>              | 12:REG_ANY(13)
   3 <ast> <r>              | 13:EXACT <r>(15)
   4 <astr> <>              | 15:END(0)
Match successful!
Freeing REx: "[sf].r"
```

Si se usa la opción `debug` de `re` con objetos expresión regular, se obtendrá información durante el proceso de matching:

```
 DB<3> use re 'debug'; $re = qr{[sf].r}
Compiling REx "[sf].r"
Final program:
   1: ANYOF[fs][] (12)
  12: REG_ANY (13)
  13: EXACT <r> (15)
  15: END (0)
anchored "r" at 2 (checking anchored) stclass ANYOF[fs][] minlen 3

  DB<4> 'astr' =~ $re
Guessing start of match in sv for REx "[sf].r" against "astr"
Found anchored substr "r" at offset 3...
Starting position does not contradict /^/m...
start_shift: 2 check_at: 3 s: 1 endpos: 2
Does not contradict STCLASS...
Guessed: match at offset 1
Matching REx "[sf].r" against "str"
   1 <a> <str>              |  1:ANYOF[fs][](12)
   2 <as> <tr>              | 12:REG_ANY(13)
   3 <ast> <r>              | 13:EXACT <r>(15)
   4 <astr> <>              | 15:END(0)
Match successful!
```

1.1.3. Tablas de Escapes, Metacarácteres, Cuantificadores, Clases

Sigue una sección de tablas con notaciones tomada de `perlre`:

Metacharacters

The following metacharacters have their standard egrep-ish meanings:

```
1. \ Quote the next metacharacter
2. ^ Match the beginning of the line
3. . Match any character (except newline)
4. $ Match the end of the line (or before newline at the end)
5. | Alternation
```

6. () Grouping
7. [] Character class

Standard greedy quantifiers

The following standard greedy quantifiers are recognized:

1. * Match 0 or more times
2. + Match 1 or more times
3. ? Match 1 or 0 times
4. {n} Match exactly n times
5. {n,} Match at least n times
6. {n,m} Match at least n but not more than m times

Non greedy quantifiers

The following non greedy quantifiers are recognized:

1. *? Match 0 or more times, not greedily
2. +? Match 1 or more times, not greedily
3. ?? Match 0 or 1 time, not greedily
4. {n}? Match exactly n times, not greedily
5. {n,}? Match at least n times, not greedily
6. {n,m}? Match at least n but not more than m times, not greedily

Possesive quantifiers

The following possesive quantifiers are recognized:

1. *+ Match 0 or more times and give nothing back
2. ++ Match 1 or more times and give nothing back
3. ?+ Match 0 or 1 time and give nothing back
4. {n}+ Match exactly n times and give nothing back (redundant)
5. {n,}+ Match at least n times and give nothing back
6. {n,m}+ Match at least n but not more than m times and give nothing back

Escape sequences

1. \t tab (HT, TAB)
2. \n newline (LF, NL)
3. \r return (CR)
4. \f form feed (FF)
5. \a alarm (bell) (BEL)
6. \e escape (think troff) (ESC)
7. \033 octal char (example: ESC)
8. \x1B hex char (example: ESC)
9. \x{263a} long hex char (example: Unicode SMILEY)
10. \cK control char (example: VT)
11. \N{name} named Unicode character
12. \l lowercase next char (think vi)
13. \u uppercase next char (think vi)
14. \L lowercase till \E (think vi)
15. \U uppercase till \E (think vi)
16. \E end case modification (think vi)
17. \Q quote (disable) pattern metacharacters till \E

Ejercicio 1.1.5. *Explique la salida:*

```
casiano@tonga:˜$ perl -wde 0
main::(-e:1):   0
  DB<1> $x = ’([a-z]+)’
  DB<2> x ’hola’ =˜ /$x/
0  ’hola’
  DB<3> x ’hola’ =˜ /\Q$x/
  empty array
  DB<4> x ’([a-z]+)’ =˜ /\Q$x/
0  1
```

Character Classes and other Special Escapes

```
\w Match a "word" character (alphanumeric plus "_")
\W Match a non-"word" character
\s Match a whitespace character
\S Match a non-whitespace character
\d Match a digit character
\D Match a non-digit character
\pP Match P, named property. Use \p{Prop} for longer names.
\PP Match non-P
\X Match eXtended Unicode "combining character sequence",
   equivalent to (?>\PM\pM*)
\C Match a single C char (octet) even under Unicode.
   NOTE: breaks up characters into their UTF-8 bytes,
   so you may end up with malformed pieces of UTF-8.
   Unsupported in lookbehind.
\1 Backreference to a specific group.
’1’ may actually be any positive integer.
\g1 Backreference to a specific or previous group,
\g{-1} number may be negative indicating a previous buffer and may
       optionally be wrapped in curly brackets for safer parsing.
\g{name} Named backreference
\k<name> Named backreference
\K Keep the stuff left of the \K, don’t include it in $&
\v Vertical whitespace
\V Not vertical whitespace
\h Horizontal whitespace
\H Not horizontal whitespace
\R Linebreak
```

Zero width assertions

Perl defines the following zero-width assertions:

```
\b Match a word boundary
\B Match except at a word boundary
\A Match only at beginning of string
\Z Match only at end of string, or before newline at the end
\z Match only at end of string
\G Match only at pos() (e.g. at the end-of-match position
of prior m//g)
```

The POSIX character class syntax

The POSIX character class syntax:

```
[:class:]
```

is also available. Note that the `[` and `]` brackets are literal; they must always be used within a character class expression.

```
# this is correct:
$string =~ /[[:alpha:]]/;

# this is not, and will generate a warning:
$string =~ /[:alpha:]/;
```

Available classes

The available classes and their backslash equivalents (if available) are as follows:

```
alpha
alnum
ascii
blank
cntrl
digit \d
graph
lower
print
punct
space \s
upper
word \w
xdigit
```

For example use `[:upper:]` to match all the uppercase characters. Note that the `[]` are part of the `[::]` construct, not part of the whole character class. For example:

```
[01[:alpha:]%]
```

matches zero, one, any alphabetic character, and the percent sign.

Equivalences to Unicode

The following equivalences to Unicode `\p{}` constructs and equivalent backslash character classes (if available), will hold:

```
[[:...:]] \p{...} backslash

alpha IsAlpha
alnum IsAlnum
ascii IsASCII
blank
cntrl IsCntrl
digit IsDigit \d
graph IsGraph
lower IsLower
print IsPrint
punct IsPunct
space IsSpace
IsSpacePerl \s
upper IsUpper
word IsWord \w
xdigit IsXDigit
```

Negated character classes

You can negate the `[::]` character classes by prefixing the class name with a `'^'`. This is a Perl extension. For example:

```
POSIX traditional Unicode

[[:^digit:]] \D \P{IsDigit}
[[:^space:]] \S \P{IsSpace}
[[:^word:]] \W \P{IsWord}
```

1.1.4. Variables especiales después de un emparejamiento

Despues de un emparejamiento con éxito, las siguientes variables especiales quedan definidas:

`$&`	El texto que casó
`` $` ``	El texto que está a la izquierda de lo que casó
`$'`	El texto que está a la derecha de lo que casó
`$1, $2, $3`, etc.	Los textos capturados por los paréntesis
`$+`	Una copia del `$1, $2,` . . . con número mas alto
`@-`	Desplazamientos de las subcadenas que casan en `$1` . . .
`@+`	Desplazamientos de los finales de las subcadenas en `$1` . . .
`$#-`	El índice del último paréntesis que casó
`$#+`	El índice del último paréntesis en la última expresión regular

Las Variables de match, pre-match y post-mach

Ejemplo:

```
#!/usr/bin/perl -w
if ("Hello there, neighbor" =~ /\s(\w+),/) {
  print "That was: ($`)($&)($').\n",
}
```

```
> matchvariables.pl
That was: (Hello)( there,)( neighbor).
```

El uso de estas variables tenía un efecto negativo en el rendimiento de la regexp. Véase `perlfaq6` la sección ``Why does using $&, $`, or $' slow my program down?``.

> *Once Perl sees that you need one of these variables anywhere in the program, it provides them on each and every pattern match. That means that on every pattern match the entire string will be copied, part of it to* `` $` ``*, part to* `$&`*, and part to* `$'`*. Thus the penalty is most severe with long strings and patterns that match often. Avoid* `$&`*,* `$'`*, and* `` $` `` *if you can, but if you can't, once you've used them at all, use them at will because you've already paid the price. Remember that some algorithms really appreciate them. As of the 5.005 release, the* `$&` *variable is no longer .^{ex}pensive”the way the other two are.*
>
> *Since Perl 5.6.1 the special variables* `@-` *and* `@+` *can functionally replace* `` $` ``*,* `$&` *and* `$'`*. These arrays contain pointers to the beginning and end of each match (see* `perlvar` *for the full story), so they give you essentially the same information, but without the risk of excessive string copying.*
>
> *Perl 5.10 added three specials,* `${^MATCH}`*,* `${^PREMATCH}`*, and* `${^POSTMATCH}` *to do the same job but without the global performance penalty. Perl 5.10 only sets these variables if you compile or execute the regular expression with the* `/p` *modifier.*

```
pl@nereida:~/Lperltesting$ cat ampersandoldway.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1 -w
use strict;
use Benchmark qw(cmpthese timethese);

'hola juan' =~ /ju/;
my ($a, $b, $c) = ($`, $&, $');


cmpthese( -1, {
    oldway => sub { 'hola juan' =~ /ju/  },
});
pl@nereida:~/Lperltesting$ cat ampersandnewway.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1 -w
use strict;
use Benchmark qw(cmpthese timethese);

'hola juan' =~ /ju/p;
my ($a, $b, $c) = (${^PREMATCH}, ${^MATCH}, ${^POSTMATCH});


cmpthese( -1, {
    newway => sub { 'hola juan' =~ /ju/  },
});

pl@nereida:~/Lperltesting$ time ./ampersandoldway.pl
            Rate oldway
oldway 2991861/s     --

real    0m3.761s
user    0m3.740s
sys     0m0.020s
pl@nereida:~/Lperltesting$ time ./ampersandnewway.pl
            Rate newway
newway 8191999/s     --

real    0m6.721s
user    0m6.704s
sys     0m0.016s
```

Véase

- `perlvar` (busque por `$MATCH`)

Texto Asociado con el Último Paréntesis

La variable `$+` contiene el texto que casó con el último paréntesis en el patrón. Esto es útil en situaciones en las cuáles una de un conjunto de alternativas casa, pero no sabemos cuál:

```
  DB<9> "Revision: 4.5" =~ /Version: (.*)|Revision: (.*)/ && ($rev = $+);
  DB<10> x $rev
0  4.5
  DB<11> "Version: 4.5" =~ /Version: (.*)|Revision: (.*)/ && ($rev = $+);
  DB<12> x $rev
0  4.5
```

Los Offsets de los Inicios de los Casamientos: `@-`

El vector `@-` contiene los *offsets* o desplazamientos de los casamientos en la última expresión regular. La entrada `$-[0]` es el desplazamiento del último casamiento con éxito y `$-[n]` es el desplazamiento de la subcadena que casa con el `n`-ésimo paréntesis (o `undef` si el párentesis no casó). Por ejemplo:

```
#           012345678
DB<1> $z = "hola13.47"
DB<2> if ($z =~ m{a(\d+)(\.(\d+))?}) { print "@-\n"; }
3 4 6 7
```

El resultado se interpreta como sigue:

- 3 = desplazamiento de comienzo de `$& = a13.47`
- 4 = desplazamiento de comienzo de `$1 = 13`
- 6 = desplazamiento de comienzo de `$2 = .`
- 7 = desplazamiento de comienzo de `$3 = 47`

Esto es lo que dice `perlvar` sobre `@-`:

> *This array holds the offsets of the beginnings of the last successful submatches in the currently active dynamic scope.* `$-[0]` *is the offset into the string of the beginning of the entire match. The nth element of this array holds the offset of the nth submatch, so* `$-[1]` *is the offset where* `$1` *begins,* `$-[2]` *the offset where* `$2` *begins, and so on.*
>
> *After a match against some variable* `$var`*:*
>
> ```
> $‘ is the same as substr($var, 0, $-[0])
> $& is the same as substr($var, $-[0], $+[0] - $-[0])
> $’ is the same as substr($var, $+[0])
> $1 is the same as substr($var, $-[1], $+[1] - $-[1])
> $2 is the same as substr($var, $-[2], $+[2] - $-[2])
> $3 is the same as substr($var, $-[3], $+[3] - $-[3])
> ```

Desplazamientos de los Finales de los Emparejamientos: `@+`

El array `@+` contiene los desplazamientos de los finales de los emparejamientos. La entrada `$+[0]` contiene el desplazamiento del final de la cadena del emparejamiento completo. Siguiendo con el ejemplo anterior:

```
#            0123456789
DB<17> $z = "hola13.47x"
DB<18> if ($z =~ m{a(\d+)(\.)(\d+)?}) { print "@+\n"; }
9 6 7 9
```

El resultado se interpreta como sigue:

- 9 = desplazamiento final de `$& = a13.47x`
- 6 = desplazamiento final de `$1 = 13`
- 7 = desplazamiento final de `$2 = .`
- 9 = desplazamiento final de `$3 = 47`

Número de paréntesis en la última regexp con éxito

Se puede usar `$#+` para determinar cuantos parentesis había en el último emparejamiento que tuvo éxito.

```
  DB<29> $z = "h"
  DB<30> print "$#+\n" if ($z =~ m{(a)(b)}) || ($z =~ m{(h)(.)?(.)?})
3
  DB<31> $z = "ab"
  DB<32> print "$#+\n" if ($z =~ m{(a)(b)}) || ($z =~ m{(h)(.)?(.)?})
2
```

Indice del Ultimo Paréntesis

La variable `$#-` contiene el índice del último paréntesis que casó. Observe la siguiente ejecución con el depurador:

```
  DB<1> $x = '13.47'; $y = '125'
  DB<2> if ($y =~ m{(\d+)(\.(\d+))?}) { print "last par = $#-, content = $+\n"; }
last par = 1, content = 125
  DB<3> if ($x =~ m{(\d+)(\.(\d+))?}) { print "last par = $#-, content = $+\n"; }
last par = 3, content = 47
```

`@-` y `@+` no tienen que tener el mismo tamaño

En general no puede asumirse que `@-` y `@+` sean del mismo tamaño.

```
  DB<1> "a" =~ /(a)|(b)/; @a = @-; @b = @+
  DB<2> x @a
0  0
1  0
  DB<3> x @b
0  1
1  1
2  undef
```

Véase También

Para saber más sobre las variables especiales disponibles consulte

- `perldoc perlretut`
- `perldoc perlvar`.

1.1.5. Ambito Automático

Como sabemos, ciertas variables (como `$1`, `$&` ...) reciben automáticamente un valor con cada operación de "matching".

Considere el siguiente código:

```
if (m/(...)/) {
  &do_something();
  print "the matched variable was $1.\n";
}
```

Puesto que `$1` es automáticamente declarada `local` a la entrada de cada bloque, no importa lo que se haya hecho en la función `&do_something()`, el valor de `$1` en la sentencia `print` es el correspondiente al "matching" realizado en el `if`.

1.1.6. Opciones

Modificador	Significado
e	evaluar: evaluar el lado derecho de una sustitución como una expresión
g	global: Encontrar todas las ocurrencias
i	ignorar: no distinguir entre mayúsculas y minúsculas
m	multilínea (`^` y `$` casan con `\n` internos)
o	optimizar: compilar una sola vez
s	`^` y `$` ignoran `\n` pero el punto `.` “casa” con `\n`
x	extendida: permitir comentarios

El Modificador `/g` La conducta de este modificador depende del contexto. En un contexto de listas devuelve una lista con todas las subcadenas casadas por todos los paréntesis en la expresión regular. Si no hubieran paréntesis devuelve una lista con todas las cadenas casadas (como si hubiera paréntesis alrededor del patrón global).

```
#!/usr/bin/perl -w
($one, $five, $fifteen) = (‘uptime‘ =~ /(\d+\.\d+)/g);
print "$one, $five, $fifteen\n";
```

Observe la salida:

```
> uptime
  1:35pm  up 19:22,  0 users,  load average: 0.01, 0.03, 0.00
> glist.pl
0.01, 0.03, 0.00
```

En un contexto escalar `m//g` itera sobre la cadena, devolviendo cierto cada vez que casa, y falso cuando deja de casar. En otras palabras, recuerda donde se quedo la última vez y se recomienza la búsqueda desde ese punto. Se puede averiguar la posicion del emparejamiento utilizando la función `pos`. Si por alguna razón modificas la cadena en cuestión, la posición de emparejamiento se reestablece al comienzo de la cadena.

```
#!/usr/bin/perl -w
# count sentences in a document
#defined as ending in [.!?] perhaps with
# quotes or parens on either side.
$/ = ""; # paragraph mode
while ($paragraph = <>) {
  print $paragraph;
  while ($paragraph =~ /[a-z][’")]*[.!?]+[’")]*\s/g) {
    $sentences++;
  }
}
print "$sentences\n";
```

Observe el uso de la variable especial `$/`. Esta variable contiene el separador de registros en el fichero de entrada. Si se iguala a la cadena vacía usará las líneas en blanco como separadores. Se le puede dar el valor de una cadena multicarácter para usarla como delimitador. Nótese que establecerla a `\n\n` es diferente de asignarla a `""`. Si se deja `undef`, la siguiente lectura leerá todo el fichero.

Sigue un ejemplo de ejecución. El programa se llama `gscalar.pl`. Introducimos el texto desde `STDIN`. El programa escribe el número de párrafos:

```
> gscalar.pl
este primer parrafo. Sera seguido de un
segundo parrafo.
```

```
"Cita de Seneca".

3
```

La opción `e`: Evaluación del remplazo La opción `/e` permite la evaluación como expresión perl de la cadena de reemplazo (En vez de considerarla como una cadena delimitada por doble comilla).

```
#!/usr/bin/perl -w
$_ = "abc123xyz\n";
s/\d+/$&*2/e;
print;
s/\d+/sprintf("%5d",$&)/e;
print;
s/\w/$& x 2/eg;
print;
```

El resultado de la ejecución es:

```
> replacement.pl
abc246xyz
abc  246xyz
aabbcc  224466xxyyzz
```

Véase un ejemplo con anidamiento de `/e`:

```
#!/usr/bin/perl
$a ="one";
$b = "two";
$_ = ’$a $b’;
print "_ = $_\n\n";
s/(\$\w+)/$1/ge;
print "After ’s/(\$\w+)/$1/ge’ _ = $_\n\n";
s/(\$\w+)/$1/gee;
print "After ’s/(\$\w+)/$1/gee’ _ = $_\n\n";
```

El resultado de la ejecución es:

```
> enested.pl
_ = $a $b

After ’s/($w+)/$b/ge’ _ = $a $b

After ’s/($w+)/$b/gee’ _ = one two
```

He aqui una solución que hace uso de `e` al siguiente ejercicio (véase ’Regex to add space after punctuation sign’ en PerlMonks) Se quiere poner un espacio en blanco después de la aparición de cada coma:

```
s/,/, /g;
```

pero se quiere que la sustitución no tenga lugar si la coma esta incrustada entre dos dígitos. Además se pide que si hay ya un espacio después de la coma, no se duplique

```
s/(\d[,.]\d)|(,(?!\s))/$1 || "$2 "/ge;
```

Se hace uso de un lookahead negativo `(?!\s)`. Véase la sección 1.2.3 para entender como funciona un lookahead negativo.

1.2. Algunas Extensiones

1.2.1. Comentarios

`(?#text)` Un comentario. Se ignora `text`. Si se usa la opción `x` basta con poner `#`.

1.2.2. Modificadores locales

Los modificadores de la conducta de una expresión regular pueden ser empotrados en una subexpresión usando el formato `(?pimsx-imsx)`.

Véase el correspondiente texto *Extended Patterns* de la sección 'Extended-Patterns' en `perlre`:

> *One or more embedded pattern-match modifiers, to be turned on (or turned off, if preceded by '`-`') for the remainder of the pattern or the remainder of the enclosing pattern group (if any). This is particularly useful for dynamic patterns, such as those read in from a configuration file, taken from an argument, or specified in a table somewhere. Consider the case where some patterns want to be case sensitive and some do not: The case insensitive ones merely need to include (?i) at the front of the pattern. For example:*
>
> ```
> $pattern = "foobar";
> if ( /$pattern/i ) { }
>
> # more flexible:
>
> $pattern = "(?i)foobar";
> if ( /$pattern/ ) { }
> ```
>
> *These modifiers are restored at the end of the enclosing group. For example,*
>
> ```
> ( (?i) blah ) \s+ \1
> ```
>
> *will match* `blah` *in any case, some spaces, and an exact (including the case!) repetition of the previous word, assuming the* `/x` *modifier, and no* `/i` *modifier outside this group.*

El siguiente ejemplo extiende el ejemplo visto en la sección 1.1.1 eliminando los comentarios `/* ... */` y `// ...` de un programa C. En dicho ejemplo se usaba el modificador `s` para hacer que el punto casara con cualquier carácter:

```
casiano@tonga:~/Lperltesting$ cat -n extendedcomments.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3
     4  my $progname = shift @ARGV or die "Usage:\n$0 prog.c\n";
     5  open(my $PROGRAM,"<$progname") || die "can't find $progname\n";
     6  my $program = '';
     7  {
     8    local $/ = undef;
     9    $program = <$PROGRAM>;
    10  }
    11  $program =~ s{(?xs)
    12    /\*  # Match the opening delimiter
    13    .*?  # Match a minimal number of characters
    14    \*/  # Match the closing delimiter
    15    |
    16      (?-s)//.* # C++ // comments. No s modifier
    17  }[]g;
    18
    19  print $program;
```

Sigue un ejemplo de ejecución. Usaremos como entrada el programa C:

```
casiano@tonga:~/Lperltesting$ cat -n ehello.c
     1  #include <stdio.h>
     2  /* first
     3  comment
     4  */
     5  main() { // A C++ comment
     6    printf("hello world!\n"); /* second comment */
     7  } // final comment
```

Al ejecutar el programa eliminamos los comentarios:

```
casiano@tonga:~/Lperltesting$ extendedcomments.pl ehello.c | cat -n
     1  #include <stdio.h>
     2
     3  main() {
     4    printf("hello world!\n");
     5  }
```

1.2.3. Mirando hacia adetrás y hacia adelante

El siguiente fragmento esta 'casi' literalmente tomado de la sección 'Looking-ahead-and-looking-behind' en `perlretut`:

Las zero-width assertions como caso particular de mirar atrás-adelante

In Perl regular expressions, most regexp elements 'eat up' a certain amount of string when they match. For instance, the regexp element `[abc}]` *eats up one character of the string when it matches, in the sense that Perl moves to the next character position in the string after the match. There are some elements, however, that don't eat up characters (advance the character position) if they match.*

The examples we have seen so far are the anchors. The anchor `^` *matches the beginning of the line, but doesn't eat any characters.*

Similarly, the word boundary anchor `\b` *matches wherever a character matching* `\w` *is next to a character that doesn't, but it doesn't eat up any characters itself.*

Anchors are examples of zero-width assertions. *Zero-width, because they consume no characters, and assertions, because they test some property of the string.*

In the context of our walk in the woods analogy to regexp matching, most regexp elements move us along a trail, but anchors have us stop a moment and check our surroundings. If the local environment checks out, we can proceed forward. But if the local environment doesn't satisfy us, we must backtrack.

Checking the environment entails either looking ahead on the trail, looking behind, or both.

- `^` *looks behind, to see that there are no characters before.*
- `$` *looks ahead, to see that there are no characters after.*
- `\b` *looks both ahead and behind, to see if the characters on either side differ in their "word-ness".*

The lookahead and lookbehind assertions are generalizations of the anchor concept. Lookahead and lookbehind are zero-width assertions that let us specify which characters we want to test for.

Lookahead assertion

The lookahead assertion is denoted by `(?=regexp)` *and the lookbehind assertion is denoted by* `(?<=fixed-regexp)`.

En español, operador de "trailing" o "mirar-adelante" positivo. Por ejemplo, `/\w+(?=\t)/` *solo casa una palabra si va seguida de un tabulador, pero el tabulador no formará parte de* `$&`. *Ejemplo:*

```
> cat -n lookahead.pl
   1  #!/usr/bin/perl
   2
   3   $a = "bugs the rabbit";
   4   $b = "bugs the frog";
   5   if ($a =~ m{bugs(?= the cat| the rabbit)}i) { print "$a matches. \$& = $&\n"; }
   6   else { print "$a does not match\n"; }
   7   if ($b =~ m{bugs(?= the cat| the rabbit)}i) { print "$b matches. \$& = $&\n"; }
   8   else { print "$b does not match\n"; }
```

Al ejecutar el programa obtenemos:

```
> lookahead.pl
bugs the rabbit matches. $& = bugs
bugs the frog does not match
>
```

Some examples using the debugger[4]*:*

```
  DB<1>      #012345678901234567890
  DB<2> $x = "I catch the housecat ’Tom-cat’ with catnip"
  DB<3> print "($&) (".pos($x).")\n" if $x  =~ /cat(?=\s)/g
(cat) (20)                   # matches ’cat’ in ’housecat’

  DB<5> $x = "I catch the housecat ’Tom-cat’ with catnip" # To reset pos
  DB<6> x @catwords = ($x =~ /(?<=\s)cat\w+/g)
0  ’catch’
1  ’catnip’

  DB<7>      #012345678901234567890123456789
  DB<8> $x = "I catch the housecat ’Tom-cat’ with catnip"
  DB<9> print "($&) (".pos($x).")\n" if $x =~ /\bcat\b/g
(cat) (29) # matches ’cat’ in ’Tom-cat’

  DB<10> $x = "I catch the housecat ’Tom-cat’ with catnip"
  DB<11> x  $x =~ /(?<=\s)cat(?=\s)/
  empty array
  DB<12> # doesn’t match; no isolated ’cat’ in middle of $x
```

A hard RegEx problem

Véase el nodo A hard RegEx problem en PerlMonks. Un monje solicita:

> *Hi Monks,*
> *I wanna to match this issues:*
>
> 1. *The string length is between 3 and 10*
> 2. *The string ONLY contains [0-9] or [a-z] or [A-Z], but*
> 3. *The string must contain a number AND a letter at least*
>
> *Pls help me check. Thanks*

Solución:

---

[4] catnip: La nepeta cataria, también llamada menta de los gatos, de la familia del tomillo y la lavanda. Su perfume desencadena un comportamiento en el animal, similar al del celo

```
casiano@millo:~$ perl -wde 0
main::(-e:1):   0
  DB<1> x 'aaa2a1' =~  /\A(?=.*[a-z])(?=.*\d)\w{3,10}\z/i
0  1
  DB<2> x 'aaaaaa' =~  /\A(?=.*[a-z])(?=.*\d)\w{3,10}\z/i
  empty array
  DB<3> x '1111111' =~  /\A(?=.*[a-z])(?=.*\d)\w{3,10}\z/i
  empty array
  DB<4> x '1111111bbbbb' =~  /\A(?=.*[a-z])(?=.*\d)\w{3,10}\z/i
  empty array
  DB<5> x '111bbbbb' =~  /\A(?=.*[a-z])(?=.*\d)\w{3,10}\z/i
0  1
```

Los paréntesis looakehaed and lookbehind no capturan

Note that the parentheses in `(?=regexp)` *and* `(?<=regexp)` *are non-capturing, since these are zero-width assertions.*

Limitaciones del lookbehind

Lookahead `(?=regexp)` *can match arbitrary regexps, but lookbehind* `(?<=fixed-regexp)` *only works for regexps of fixed width, i.e., a fixed number of characters long.*

Thus `(?<=(ab|bc))` *is fine, but* `(?<=(ab)*)` *is not.*

Negación de los operadores de lookahead y lookbehind

The negated versions of the lookahead and lookbehind assertions are denoted by `(?!regexp)` *and* `(?<!fixed-regexp)` *respectively. They evaluate true if the regexps do not match:*

```
    $x = "foobar";
    $x =~ /foo(?!bar)/;  # doesn't match, 'bar' follows 'foo'
    $x =~ /foo(?!baz)/;  # matches, 'baz' doesn't follow 'foo'
    $x =~ /(?<!\s)foo/;  # matches, there is no \s before 'foo'
```

Ejemplo: split con lookahead y lookbehind

Here is an example where a string containing blank-separated words, numbers and single dashes is to be split into its components.

Using `/\s+/` *alone won't work, because spaces are not required between dashes, or a word or a dash. Additional places for a split are established by looking ahead and behind:*

```
casiano@tonga:~$ perl5.10.1 -wdE 0
main::(-e:1):   0
  DB<1> $str = "one two - --6-8"
  DB<2> x @toks = split / \s+ | (?<=\S) (?=-) | (?<=-)  (?=\S)/x, $str
0  'one'
1  'two'
2  '-'
3  '-'
4  '-'
5  6
6  '-'
7  8
```

Look Around en perlre

El siguiente párrafo ha sido extraído la sección 'Look-Around-Assertions' en `pelre`. Usémoslo como texto de repaso:

Look-around assertions are zero width patterns which match a specific pattern without including it in `$&`*. Positive assertions match when their subpattern matches, negative assertions match when their subpattern fails. Look-behind matches text up to the current match position, look-ahead matches text following the current match position.*

- `(?=pattern)`

  A zero-width positive look-ahead assertion. For example, `/\w+(?=\t)/` *matches a word followed by a tab, without including the tab in* `$&`*.*
- `(?!pattern)`

  A zero-width negative look-ahead assertion. For example `/foo(?!bar)/` *matches any occurrence of* `foo` *that isn't followed by* `bar`*.*

  Note however that look-ahead and look-behind are NOT the same thing. You cannot use this for look-behind.

  If you are looking for a `bar` *that isn't preceded by a* `foo`*,* `/(?!foo)bar/` *will not do what you want.*

  That's because the `(?!foo)` *is just saying that the next thing cannot be* `foo` *–and it's not, it's a* `bar`*, so* `foobar` *will match.*

  You would have to do something like `/(?!foo)...bar/` *for that.*

  We say ”like”because there's the case of your `bar` *not having three characters before it.*

  You could cover that this way: `/(?:(?!foo)...|^.{0,2})bar/`*. Sometimes it's still easier just to say:*

  ```
      if (/bar/ && $` !~ /foo$/)
  ```

  For look-behind see below.
- `(?<=pattern)`

  A zero-width positive look-behind assertion.

  For example, `/(?<=\t)\w+/` *matches a word that follows a tab, without including the tab in* `$&`*. Works only for fixed-width look-behind.*
- `\K`

  There is a special form of this construct, called `\K` *, which causes the regex engine to 'keep' everything it had matched prior to the* `\K` *and not include it in* `$&`*. This effectively provides variable length look-behind. The use of* `\K` *inside of another look-around assertion is allowed, but the behaviour is currently not well defined.*

  For various reasons `\K` *may be significantly more efficient than the equivalent* `(?<=...)` *construct, and it is especially useful in situations where you want to efficiently remove something following something else in a string. For instance*

  ```
    s/(foo)bar/$1/g;
  ```

  can be rewritten as the much more efficient

  ```
    s/foo\Kbar//g;
  ```

  Sigue una sesión con el depurador que ilustra la semántica del operador:

  ```
  casiano@millo:~$ perl5.10.1 -wdE 0
  main::(-e:1):   0
    DB<1> print "& = <$&> 1 = <$1>\n" if "alphabet" =~ /([^aeiou][a-z][aeiou])[a-z]/
  & = <phab> 1 = <pha>
    DB<2> print "& = <$&> 1 = <$1>\n" if "alphabet" =~ /\K([^aeiou][a-z][aeiou])[a-z]/
  & = <phab> 1 = <pha>
    DB<3> print "& = <$&> 1 = <$1>\n" if "alphabet" =~ /([^aeiou]\K[a-z][aeiou])[a-z]/
  & = <hab> 1 = <pha>
  ```

```
  DB<4> print "& = <$&> 1 = <$1>\n" if "alphabet" =~ /([^aeiou][a-z]\K[aeiou])[a-z]/
& = <ab> 1 = <pha>
  DB<5> print "& = <$&> 1 = <$1>\n" if "alphabet" =~ /([^aeiou][a-z][aeiou])\K[a-z]/
& = <b> 1 = <pha>
  DB<6> print "& = <$&> 1 = <$1>\n" if "alphabet" =~ /([^aeiou][a-z][aeiou])[a-z]\K/
& = <> 1 = <pha>
  DB<7> @a = "alphabet" =~ /([aeiou]\K[^aeiou])/g; print "$&\n"
t
  DB<8> x @a
0  'al'
1  'ab'
2  'et'
```

Otro ejemplo: eliminamos los blancos del final en una cadena:

```
  DB<23> $x = '  cadena entre blancos  '
  DB<24> ($y = $x) =~ s/.*\b\K.*//g
  DB<25> p "<$y>"
<  cadena entre blancos>
```

- `(?<!pattern)`

  A zero-width negative look-behind assertion.

  For example `/(?<!bar)foo/` *matches any occurrence of* `foo` *that does not follow* `bar`*.*
  Works only for fixed-width look-behind.

Veamos un ejemplo de uso. Se quiere sustituir las extensiones `.something` por `.txt` en cadenas que contienen una ruta a un fichero:

```
casiano@millo:~$ perl5.10.1 -wdE 0
main::(-e:1):   0
  DB<1> ($b = $a = 'abc/xyz.something') =~ s{\.[^.]*$}{.txt}
  DB<2> p $b
abc/xyz.txt
  DB<3> ($b = $a = 'abc/xyz.something') =~ s/\.\K[^.]*$/txt/;
  DB<4> p $b
abc/xyz.txt
  DB<5> p $a
abc/xyz.something
```

Véase también:

- Regexp::Keep por Jeff Pinyan
- El nodo *positive look behind regexp mystery* en PerlMonks

Operador de predicción negativo: Última ocurrencia

Escriba una expresión regular que encuentre la última aparición de la cadena `foo` en una cadena dada.

```
  DB<6> x ($a = 'foo foo bar bar foo bar bar') =~ /foo(?!.*foo)/g; print pos($a)."\n"
19
  DB<7> x ($a = 'foo foo bar bar foo bar bar') =~ s/foo(?!.*foo)/\U$&/
0  1
  DB<8> x $a
0  'foo foo bar bar FOO bar bar'
```

Diferencias entre mirar adelante negativo y mirar adelante con clase negada

Aparentemente el operador “mirar-adelante” negativo es parecido a usar el operador “mirar-adelante” positivo con la negación de una clase.

`/regexp(?![abc])/`	`/regexp(?=[^abc])/`

Sin embargo existen al menos dos diferencias:

- Una negación de una clase debe casar algo para tener éxito. Un ‘mirar-adelante” negativo tiene éxito si, en particular no logra casar con algo. Por ejemplo:

  `\d+(?!\.)` casa con `$a = '452'`, mientras que `\d+(?=[^.])` lo hace, pero porque `452` es `45` seguido de un carácter que no es el punto:

  ```
  > cat lookaheadneg.pl
  #!/usr/bin/perl

   $a = "452";
   if ($a =~ m{\d+(?=[^.])}i) { print "$a casa clase negada. \$& = $&\n"; }
   else { print "$a no casa\n"; }
   if ($a =~ m{\d+(?!\.)}i) { print "$a casa predicción negativa. \$& = $&\n"; }
   else { print "$b no casa\n"; }
  nereida:~/perl/src> lookaheadneg.pl
  452 casa clase negada. $& = 45
  452 casa predicción negativa. $& = 452
  ```

- Una clase negada casa un único carácter. Un ‘mirar-adelante” negativo puede tener longitud arbitraria.

AND y AND NOT

Otros dos ejemplos:

- `^(?![A-Z]*$)[a-zA-Z]*$`

  casa con líneas formadas por secuencias de letras tales que no todas son mayúsculas. (Obsérvese el uso de las anclas).

- `^(?=.*?esto)(?=.*?eso)`

  casan con cualquier línea en la que aparezcan `esto` y `eso`. Ejemplo:

  ```
  > cat estoyeso.pl
  #!/usr/bin/perl

   my $a = shift;

   if ($a =~ m{^(?=.*?esto)(?=.*?eso)}i) { print "$a matches.\n"; }
   else { print "$a does not match\n"; }

  >estoyeso.pl 'hola eso y esto'
  hola eso y esto matches.
  > estoyeso.pl 'hola esto y eso'
  hola esto y eso matches.
  > estoyeso.pl 'hola aquello y eso'
  ```

```
hola aquello y eso does not match
> estoyeso.pl 'hola esto y aquello'
hola esto y aquello does not match
```

El ejemplo muestra que la interpretación es que cada operador mirar-adelante se interpreta siempre a partir de la posición actual de búsqueda. La expresión regular anterior es básicamente equivalente a `(/esto/ && /eso/)`.

- `(?!000)(\d\d\d)`

  casa con cualquier cadena de tres dígitos que no sea la cadena `000`.

Lookahead negativo versus lookbehind

Nótese que el "mirar-adelante" negativo no puede usarse fácilmente para imitar un "mirar-atrás", esto es, que no se puede imitar la conducta de `(?<!foo)bar` mediante algo como `(/?!foo)bar`. Tenga en cuenta que:

- Lo que dice `(?!foo)` es que los tres caracteres que siguen no puede ser `foo`.
- Así, `foo` no pertenece a `/(?!foo)bar/`, pero `foobar` pertenece a `(?!foo)bar/` porque `bar` es una cadena cuyos tres siguientes caracteres son `bar` y no son `foo`.
- Si quisieramos conseguir algo parecido a `(?<!foo)bar` usando un lookahead negativo tendríamos que escribir algo asi como `/(?!foo)...bar/` que casa con una cadena de tres caracteres que no sea `foo` seguida de `bar` (pero que tampoco es exactamente equivalente):

```
pl@nereida:~/Lperltesting$ cat -n foobar.pl
     1  use v5.10;
     2  use strict;
     3
     4  my $a = shift;
     5
     6  for my $r (q{(?<!foo)bar}, q{(?!foo)bar}, q{(?!foo)...bar}) {
     7    if ($a =~ /$r/) {
     8      say "$a casa con $r"
     9    }
    10    else {
    11      say "$a no casa con $r"
    12    }
    13  }
```

- Al ejecutar con diferentes entradas el programa anterior vemos que la solución `q{(?!foo)...bar}` se apróxima mas a `(q{(?<!foo)bar}`:

```
pl@nereida:~/Lperltesting$ perl5.10.1 foobar.pl foobar
foobar no casa con (?<!foo)bar
foobar casa con (?!foo)bar
foobar no casa con (?!foo)...bar

pl@nereida:~/Lperltesting$ perl5.10.1 foobar.pl bar
bar casa con (?<!foo)bar
bar casa con (?!foo)bar
bar no casa con (?!foo)...bar
```

Ejercicio 1.2.1. *Explique porqué* `bar` *casa con* `(?<!foo)bar` *pero no con* `(?!foo)...bar`. *¿Sabría encontrar una expresión regular mas apropiada usando lookahead negativo?*

- En realidad, posiblemente sea mas legible una solución como:

  ```
  if (/bar/ and $‘ !~ /foo$/)
  ```

  o aún mejor (véase 1.1.4):

  ```
  if (/bar/p && ${^PREMATCH} =~ /foo$/)
  ```

  El siguiente programa puede ser utilizado para ilustrar la equivalencia:

  ```
  pl@nereida:~/Lperltesting$ cat -n foobarprematch.pl
  use v5.10;
  use strict;

  $_ = shift;

  if (/bar/p && ${^PREMATCH} =~ /foo$/) {
    say "$_ no cumple ".q{/bar/p && ${^PREMATCH} =~ /foo$/};
  }
  else {
    say "$_ cumple ".q{/bar/p && ${^PREMATCH} =~ /foo$/};
  }
  if (/(?<!foo)bar/) {
    say "$_ casa con (?<!foo)bar"
  }
  else {
    say "$_ no casa con (?<!foo)bar"
  }
  ```

  Siguen dos ejecuciones:

  ```
  pl@nereida:~/Lperltesting$ perl5.10.1 foobarprematch.pl bar
  bar cumple /bar/p && ${^PREMATCH} =~ /foo$/
  bar casa con (?<!foo)bar
  pl@nereida:~/Lperltesting$ perl5.10.1 foobarprematch.pl foobar
  foobar no cumple /bar/p && ${^PREMATCH} =~ /foo$/
  foobar no casa con (?<!foo)bar
  ```

Ejercicios

Ejercicio 1.2.2.

- *Escriba una sustitución que reemplaze todas las apariciones de* `foo` *por* `foo,` *usando* `\K` *o lookbehind*
- *Escriba una sustitución que reemplaze todas las apariciones de* `lookahead` *por* `look-ahead` *usando lookaheads y lookbehinds*
- *Escriba una expresión regular que capture todo lo que hay entre las cadenas* `foo` *y* `bar` *siempre que no se incluya la palabra* `baz`
- *¿Cuál es la salida?*

  ```
  DB<1> x 'abc' =~ /(?=(.)(.)(.))a(b)/
  ```

- *Se quiere poner un espacio en blanco después de la aparición de cada coma:*

  ```
  s/,/, /g;
  ```

pero se quiere que la sustitución no tenga lugar si la coma esta incrustada entre dos dígitos.

- *Se quiere poner un espacio en blanco después de la aparición de cada coma:*

  ```
  s/,/, /g;
  ```

  pero se quiere que la sustitución no tenga lugar si la coma esta incrustada entre dos dígitos. Además se pide que si hay ya un espacio después de la coma, no se duplique

- *¿Cuál es la salida?*

  ```
  pl@nereida:~/Lperltesting$ cat -n ABC123.pl
  use warnings;
  use strict;

  my $c = 0;
  my @p = (’^(ABC)(?!123)’, ’^(\D*)(?!123)’,);

  for my $r (@p) {
    for my $s (qw{ABC123 ABC445}) {
      $c++;
      print "$c: ’$s’ =~ /$r/ : ";
      <>;
      if ($s =~ /$r/) {
        print " YES ($1)\n";
      }
      else {
        print " NO\n";
      }
    }
  }
  ```

1.2.4. Definición de Nombres de Patrones

Perl 5.10 introduce la posibilidad de definir subpatrones en una sección del patrón.

Lo que dice `perlretut` sobre la definición de nombres de patrones

Citando la sección *Defining named patterns* en el documento la sección ’Defining-named-patterns’ en `perlretut` para perl5.10:

> *Some regular expressions use identical subpatterns in several places. Starting with Perl 5.10, it is possible to define named subpatterns in a section of the pattern so that they can be called up by name anywhere in the pattern. This syntactic pattern for this definition group is* `"(?(DEFINE)(?<name>pattern)...)"` *An insertion of a named pattern is written as* `(?&name)`.

Veamos un ejemplo que define el lenguaje de los números en punto flotante:

```
pl@nereida:~/Lperltesting$ cat -n definingnamedpatterns.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1 -w
use v5.10;

my $regexp = qr{
   ^ (?<num>
             (?&osg)[\t\ ]* (?: (?&int)(?&dec)? | (?&dec) )
```

```
     )
     (?: [eE]
     (?<exp> (?&osg)(?&int)) )?
   $
      (?(DEFINE)
       (?<osg>[-+]?)         # optional sign
       (?<int>\d++)          # integer
       (?<dec>\.(?&int))     # decimal fraction
      )
}x;

my $input = <>;
chomp($input);
my @r;
if (@r = $input =~ $regexp) {
  my $exp = $+{exp} || '';
  say "$input matches: (num => '$+{num}', exp => '$exp')";
}
else {
  say "does not match";
}
```

`perlretut` comenta sobre este ejemplo:

> *The example above illustrates this feature. The three subpatterns that are used more than once are the optional sign, the digit sequence for an integer and the decimal fraction. The* `DEFINE` *group at the end of the pattern contains their definition. Notice that the decimal fraction pattern is the first place where we can reuse the integer pattern.*

Lo que dice `perlre` sobre la definición de patrones

Curiosamente, `(DEFINE)` se considera un caso particular de las expresiones regulares condicionales de la forma `(?(condition)yes-pattern)` (véase la sección 1.2.10). Esto es lo que dice la sección 'Extended-Patterns' en `perlre` al respecto:

> *A special form is the* `(DEFINE)` *predicate, which never executes directly its yes-pattern, and does not allow a no-pattern. This allows to define subpatterns which will be executed only by using the recursion mechanism. This way, you can define a set of regular expression rules that can be bundled into any pattern you choose.*
>
> *It is recommended that for this usage you put the* `DEFINE` *block at the end of the pattern, and that you name any subpatterns defined within it.*
>
> *Also, it's worth noting that patterns defined this way probably will not be as efficient, as the optimiser is not very clever about handling them.*
>
> *An example of how this might be used is as follows:*
>
> ```
> /(?<NAME>(?&NAME_PAT))(?<ADDR>(?&ADDRESS_PAT))
>        (?(DEFINE)
>          (?<NAME_PAT>....)
>          (?<ADRESS_PAT>....)
> )/x
> ```
>
> *Note that* **capture buffers matched inside of recursion are not accessible after the recursion returns**, *so the extra layer of capturing buffers is necessary. Thus* `$+{NAME_PAT}` *would not be defined even though* `$+{NAME}` *would be.*

Lo que dice `perlvar` sobre patrones con nombre Esto es lo que dice `perlvar` respecto a las variables implicadas `%+` y `%-`. Con respecto a el hash `%+`:

- `%LAST_PAREN_MATCH`, `%+`

  Similar to `@+` *, the* `%+` *hash allows access to the named capture buffers, should they exist, in the last successful match in the currently active dynamic scope.*

  For example, `$+{foo}` *is equivalent to* `$1` *after the following match:*

  ```
  ’foo’ =~ /(?<foo>foo)/;
  ```

  The keys of the `%+` *hash list only the names of buffers that have captured (and that are thus associated to defined values).*

  The underlying behaviour of `%+` *is provided by the Tie::Hash::NamedCapture module.*

  Note: `%-` *and* `%+` *are tied views into a common internal hash associated with the last successful regular expression. Therefore mixing iterative access to them via* `each` *may have unpredictable results. Likewise, if the last successful match changes, then the results may be surprising.*

- `%-`

  Similar to `%+` *, this variable allows access to the named capture buffers in the last successful match in the currently active dynamic scope.* **To each capture buffer name found in the regular expression, it associates a reference to an array containing the list of values captured by all buffers with that name (should there be several of them), in the order where they appear**.

  Here's an example:

  ```
  if (’1234’ =~ /(?<A>1)(?<B>2)(?<A>3)(?<B>4)/) {
    foreach my $bufname (sort keys %-) {
      my $ary = $-{$bufname};
      foreach my $idx (0..$#$ary) {
        print "\$-{$bufname}[$idx] : ",
              (defined($ary->[$idx]) ? "’$ary->[$idx]’" : "undef"),
              "\n";
      }
    }
  }
  ```

  would print out:

  ```
  $-{A}[0] : ’1’
  $-{A}[1] : ’3’
  $-{B}[0] : ’2’
  $-{B}[1] : ’4’
  ```

  The keys of the `%-` *hash correspond to all buffer names found in the regular expression.*

1.2.5. Patrones Recursivos

Perl 5.10 introduce la posibilidad de definir subpatrones en una sección del patrón. Citando la versión del documento `perlretut` para perl5.10:

> *This feature (introduced in Perl 5.10) significantly extends the power of Perl's pattern matching. By referring to some other capture group anywhere in the pattern with the construct* `(?group-ref)`*, the pattern within the referenced group is used as an independent subpattern in place of the group reference itself. Because the group reference may be contained within the group it refers to, it is now possible to apply pattern matching to tasks that hitherto required a recursive parser.*

> ...
>
> *In* `(?...)` *both absolute and relative backreferences may be used. The entire pattern can be reinserted with* `(?R)` *or* `(?0)`*. If you prefer to name your buffers, you can use* `(?&name)` *to recurse into that buffer.*

Palíndromos

Véase un ejemplo que reconoce los palabra-palíndromos (esto es, la lectura directa y la inversa de la cadena pueden diferir en los signos de puntuación):

```
casiano@millo:~/Lperltesting$ cat -n palindromos.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1 -w
use v5.10;

my $regexp = qr/^(\W*
                     (?:
                           (\w) (?1) \g{-1}  # palindromo estricto
                         |
                           \w?               # no recursiva
                     )
                  \W*)$/ix;

my $input = <>;
chomp($input);
if ($input =~ $regexp) {
  say "$input is a palindrome";
}
else {
  say "does not match";
}
```

Ejercicio 1.2.3. *¿Cuál es el efecto del modificador* `i` *en la regexp* `qr/^(\W* (?: (\w) (?1) \g{-1} | \w? ) \W*`

Siguen algunos ejemplos de ejecución[5]

```
pl@nereida:~/Lperltesting$ ./palindromos.pl
A man, a plan, a canal: Panama!
A man, a plan, a canal: Panama! is a palindrome
pl@nereida:~/Lperltesting$ ./palindromos.pl
A man, a plan, a cam, a yak, a yam, a canal { Panama!
A man, a plan, a cam, a yak, a yam, a canal { Panama! is a palindrome
pl@nereida:~/Lperltesting$ ./palindromos.pl
A man, a plan, a cat, a ham, a yak, a yam, a hat, a canal { Panama!
A man, a plan, a cat, a ham, a yak, a yam, a hat, a canal { Panama! is a palindrome
pl@nereida:~/Lperltesting$ ./palindromos.pl
saippuakauppias
saippuakauppias is a palindrome
pl@nereida:~/Lperltesting$ ./palindromos.pl
dfghjgfd
does not match
```

---

[5]

- saippuakauppias: Vendedor de jabón (suomi)
- yam: batata (inglés)
- cam: leva

```
pl@nereida:~/Lperltesting$ ./palindromos.pl
...,;;;;
...,;;;; is a palindrome
```

Lo que dice `perlre` sobre recursividad

> `(?PARNO) (?-PARNO) (?+PARNO) (?R) (?0)`
>
> *Similar to* `(??{ code })` *(véase la sección 1.2.9) except it does not involve compiling any code, instead it treats the contents of a capture buffer as an independent pattern that must match at the current position. Capture buffers contained by the pattern will have the value as determined by the outermost recursion.*
>
> `PARNO` *is a sequence of digits (not starting with 0) whose value reflects the paren-number of the capture buffer to recurse to.*
>
> `(?R)` *recurses to the beginning of the whole pattern.* `(?0)` *is an alternate syntax for* `(?R)`.
>
> *If* `PARNO` *is preceded by a plus or minus sign then it is assumed to be relative, with negative numbers indicating preceding capture buffers and positive ones following. Thus (?-1) refers to the most recently declared buffer, and (?+1) indicates the next buffer to be declared.*
>
> *Note that the counting for relative recursion differs from that of relative backreferences, in that with recursion unclosed buffers are included.*

Hay una diferencia fundamental entre `\g{-1}` y `(?-1)`. El primero significa *lo que casó con el último paréntesis*. El segundo significa que se debe *llamar a la expresión regular que define el último paréntesis*. Véase un ejemplo:

```
pl@nereida:~/Lperltesting$ perl5.10.1 -wdE 0
main::(-e:1):   0
  DB<1>  x ($a = "12 aAbB 34") =~ s/([aA])(?-1)(?+1)([bB])/-\1\2-/g
0  1
  DB<2> p $a
12 -aB- 34
```

En `perlre` también se comenta sobre este punto:

> *If there is no corresponding capture buffer defined, then it is a fatal error. Recursing deeper than 50 times without consuming any input string will also result in a fatal error. The maximum depth is compiled into perl, so changing it requires a custom build.*

Paréntesis Equilibrados

El siguiente programa (inspirado en uno que aparece en `perlre`) reconoce una llamada a una función `foo()` que puede contener una secuencia de expresiones con paréntesis equilibrados como argumento:

```
pl@nereida:~/Lperltesting$ cat perlrebalancedpar.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1  -w
use v5.10;
use strict;

my $regexp = qr{ (                      # paren group 1 (full function)
                foo
                   (                    # paren group 2 (parens)
                     \(
                        (               # paren group 3 (contents of parens)
                           (?:
```

```
                                [^()]+  # Non-parens
                              |
                                (?2) # Recurse to start of paren group 2
                           )*
                        )               # 3
                     \)
                    )                   # 2
              )                         # 1
    }x;

my $input = <>;
chomp($input);
my @res = ($input =~ /$regexp/);
if (@res) {
  say "<$&> is balanced\nParen: (@res)";
}
else {
  say "does not match";
}
```

Al ejecutar obtenemos:

```
pl@nereida:~/Lperltesting$ ./perlrebalancedpar.pl
foo(bar(baz)+baz(bop))
<foo(bar(baz)+baz(bop))> is balanced
Paren: (foo(bar(baz)+baz(bop)) (bar(baz)+baz(bop)) bar(baz)+baz(bop))
```

Como se comenta en `perlre` es conveniente usar índices relativos si se quiere tener una expresión regular reciclable:

> *The following shows how using negative indexing can make it easier to embed recursive patterns inside of a* `qr//` *construct for later use:*

```
my $parens = qr/(\((?:[^()]++|(?-1))*+\))/;
if (/foo $parens \s+ + \s+ bar $parens/x) {
  # do something here...
}
```

Véase la sección 1.2.6 para comprender el uso de los operadores posesivos como `++`.

Capturando los bloques de un programa

El siguiente programa presenta una heurística para determinar los bloques de un programa:

```
pl@nereida:~/Lperltesting$ cat blocks.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1 -w
use v5.10;
use strict;
#use re 'debug';

my $rb = qr{(?x)
    (
      \{                # llave abrir
         (?:
             [^{}]++    # no llaves
```

```
         |
              [^{}]*+  # no llaves
              (?1)     # recursivo
              [^{}]*+  # no llaves
         )*+
       \}              # llave cerrar
    )
  };

local $/ = undef;
my $input = <>;
my@blocks = $input =~ m{$rb}g;
my $i = 0;
say($i++.":\n$_\n===") for @blocks;
```

Veamos una ejecución. Le daremos como entrada el siguiente programa: Al ejecutar el programa con esta entrada obtenemos:

```
pl@nereida:~/Lperltesting$ cat -n blocks.c
     1  main() { /* 1 */
     2    { /* 2 */ }
     3    { /* 3 */ }
     4  }
     5
     6  f(){  /* 4 */
     7    {   /* 5 */
     8      { /* 6 */ }
     9    }
    10    {   /* 7 */
    11      { /* 8 */ }
    12    }
    13  }
    14
    15  g(){ /* 9 */
    16  }
    17
    18  h() {
    19  {{{}}}
    20  }
    21  /* end h */
```

```
pl@nereida:~/Lperltesting$ perl5.10.1 blocks.pl b
0:
{ /* 1 */
  { /* 2 */ }
  { /* 3 */ }
}
===
1:
{  /* 4 */
  {   /* 5 */
    { /* 6 */ }
  }
  {   /* 7 */
    { /* 8 */ }
  }
}
===
2:
{ /* 9 */
}
===
3:
{
{{{}}}
}
===
```

Reconocimiento de Lenguajes Recursivos: Un subconjunto de LaTeX

La posibilidad de combinar en las expresiones regulares Perl 5.10 la recursividad con los constructos `(?<name>...)` y `?&name)` así como las secciones `(?(DEFINE) ...)` permiten la escritura de expresiones regulares que reconocen lenguajes recursivos. El siguiente ejemplo muestra un reconocedor de un subconjunto del lenguaje LaTeX (véase la entrada LaTeX en la wikipedia):

```
pl@nereida:~/Lperltesting$ cat latex5_10.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1 -w
```

```
use strict;
use v5.10;

my $regexp = qr{
    \A(?&File)\z

    (?(DEFINE)
        (?<File>     (?&Element)*+\s*
        )

        (?<Element>  \s* (?&Command)
                  |  \s* (?&Literal)
        )

        (?<Command>  \\ \s* (?<L>(?&Literal)) \s* (?<Op>(?&Options)?) \s* (?<A>(?&Args))
           (?{
              say "command: <$+{L}> options: <$+{Op}> args: <$+{A}>"
           })
        )

        (?<Options>  \[ \s* (?:(?&Option) (?:\s*,\s* (?&Option) )*)? \s* \]
        )

        (?<Args>     (?: \{ \s* (?&Element)* \s* \} )*
        )

        (?<Option>   \s* [^][\$&%#_{}~^\s,]+
        )

        (?<Literal>  \s* ([^][\$&%#_{}~^\s]+)
        )
    )
}xms;

my $input = do{ local $/; <>};
if ($input =~ $regexp) {
  say "$@: matches:\n$&";
}
else {
  say "does not match";
}
```

Añadimos una acción semántica al final de la aceptación de un `<Command>`.

```
(?<Command>  \\ \s* (?<L>(?&Literal)) \s* (?<Op>(?&Options)?) \s* (?<A>(?&Args)?)
   (?{
      say "command: <$+{L}> options: <$+{Op}> args: <$+{A}>"
   })
)
```

Esta acción es ejecutada pero no afecta al proceso de análisis. (véase la sección 1.2.8 para mas información sobre las acciones semánticas en medio de una regexp). La acción se limita a mostrar que ha casado con cada una de las tres componentes: el comando, las opciones y los argumentos.

Los paréntesis adicionales, como en `(?<L>(?&Literal))` son necesarios para guardar lo que casó.

Cuando se ejecuta produce la siguiente salida[6]:

```
pl@nereida:~/Lperltesting$ cat prueba.tex
\documentclass[a4paper,11pt]{article}
\usepackage{latexsym}
\author{D. Conway}
\title{Parsing \LaTeX{}}
\begin{document}
\maketitle
\tableofcontents
\section{Description}
...is easy \footnote{But not\\ \emph{necessarily} simple}.
In fact it's easy peasy to do.
\end{document}

pl@nereida:~/Lperltesting$ ./latex5_10.pl prueba.tex
command: <documentclass> options: <[a4paper,11pt]> args: <{article}>
command: <usepackage> options: <> args: <{latexsym}>
command: <author> options: <> args: <{D. Conway}>
command: <LaTeX> options: <> args: <{}>
command: <title> options: <> args: <{Parsing \LaTeX{}}>
command: <begin> options: <> args: <{document}>
command: <maketitle> options: <> args: <>
command: <tableofcontents> options: <> args: <>
command: <section> options: <> args: <{Description}>
command: <emph> options: <> args: <{necessarily}>
command: <footnote> options: <> args: <{But not\\ \emph{necessarily} simple}>
command: <end> options: <> args: <{document}>
: matches:
\documentclass[a4paper,11pt]{article}
\usepackage{latexsym}
\author{D. Conway}
\title{Parsing \LaTeX{}}
\begin{document}
\maketitle
\tableofcontents
\section{Description}
...is easy \footnote{But not\\ \emph{necessarily} simple}.
In fact it's easy peasy to do.
```

---

[6]

- peasy:A disagreeable taste of very fresh green peas
- easy peasy:
  1. (uk) very easy (short for easy-peasy-lemon-squeezy)
  2. the first half of a rhyming phrase with several alternate second halves, all of which connote an activity or a result that is, respectively, simple to perform or achieve.

     Tie your shoes? Why that's easy peasy lemon squeezy!
     Beat your meat? Why that's easy peasy Japanesey!
     As a red-stater, condemn books and films without having read or seen them? Why that's easy peasy puddin'n'pie!
  3. It comes from a 1970's british TV commercial for Lemon Squeezy detergent. They were with a little girl who points out dirty greasy dishes to an adult (mom or relative) and then this adult produces Lemon Squeezy and they clean the dishes quickly. At the end of the commercial the girl says *Easy Peasy Lemon Squeezy.*

     Today it is a silly way to state something was or will be very easy.

```
\end{document}
```

La siguiente entrada `prueba3.tex` no pertenece al lenguaje definido por el patrón regular, debido a la presencia de la cadena `$In$` en la última línea:

```
pl@nereida:~/Lperltesting$ cat prueba3.tex
\documentclass[a4paper,11pt]{article}
\usepackage{latexsym}
\author{D. Conway}
\title{Parsing \LaTeX{}}
\begin{document}
\maketitle
\tableofcontents
\section{Description}
\comm{a}{b}
...is easy \footnote{But not\\ \emph{necessarily} simple}.
$In$ fact it's easy peasy to do.
\end{document}

pl@nereida:~/Lperltesting$ ./latex5_10.pl prueba3.tex
command: <documentclass> options: <[a4paper,11pt]> args: <{article}>
command: <usepackage> options: <> args: <{latexsym}>
command: <author> options: <> args: <{D. Conway}>
command: <LaTeX> options: <> args: <{}>
command: <title> options: <> args: <{Parsing \LaTeX{}}>
command: <begin> options: <> args: <{document}>
command: <maketitle> options: <> args: <>
command: <tableofcontents> options: <> args: <>
command: <section> options: <> args: <{Description}>
command: <comm> options: <> args: <{a}{b}>
command: <emph> options: <> args: <{necessarily}>
command: <footnote> options: <> args: <{But not\\ \emph{necessarily} simple}>
does not match
```

Ejercicio 1.2.4. *Obsérvese el uso del cuantificador posesivo en:*

```
        (?<File>     (?&Element)*+\s*
        )
```

¿Que ocurrre si se quita el posesivo y se vuelve a ejecutar `$ ./latex5_10.pl prueba3.tex`*?*

Reconocimiento de Expresiones Aritméticas

Véase el nodo Complex regex for maths formulas en perlmonks para la formulación del problema. Un monje pregunta:

> *Hiya monks,*
>
> *Im having trouble getting my head around a regular expression to match sequences. I need to catch all exceptions where a mathematical expression is illegal...*
>
> *There must be either a letter or a digit either side of an operator parenthesis must open and close next to letters or digits, not next to operators, and do not have to exist variables must not be more than one letter Nothing other than* `a-z,A-Z,0-9,+,-,*,/,(,)` *can be used*
>
> *Can anyone offer a hand on how best to tackle this problem?*
>
> *many thanks*

La solución parte de que una *expresión* es o bien un *término* o bien un *término* seguido de una operador y un *término*, esto es:

- `termino`
- `termino op termino op termino` ...

que puede ser unificado como `termino (op termino)*`.

Un *término* es un número o un identificador o una *expresión* entre paréntesis, esto es:

- `numero`
- `identificador`
- `( expresión )`

La siguiente expresión regular recursiva sigue esta idea:

```
pl@nereida:~/Lperltesting$ cat -n simpleexpressionsna.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1
use v5.10;
use strict;
use warnings;

local our ($skip, $term, $expr);
$skip = qr/\s*/;
$expr = qr{ (?<EXPR>
                   (?<TERM>              # An expression is a TERM  ...
                          $skip (?<ID>[a-zA-Z]+)
                        | $skip (?<INT>[1-9]\d*)
                        | $skip \(
                          $skip  (?&EXPR)
                          $skip \)
                   ) (?: $skip           # possibly followed by a sequence of ...
                         (?<OP>[-+*/])
                         (?&TERM)        # ... operand TERM pairs
                     )*
            )
          }x;
my $re = qr/^ $expr $skip \z/x;
sub is_valid { shift =~ /$re/o }

my @test = ( ’(a + 3)’, ’(3 * 4)+(b + x)’, ’(5 - a)*z’,
              ’((5 - a))*((((z)))+2)’, ’3 + 2’, ’!3 + 2’, ’3 + 2!’,
              ’3 a’, ’3 3’, ’3 * * 3’,
              ’2 - 3 * 4’,  ’2 - 3 + 4’,
            );
foreach (@test) {
  say("$_:");
  say(is_valid($_) ? "\n<$_> is valid" : "\n<$_> is not valid")
}
```

Podemos usar acciones semánticas empotradas para ver la forma en la que trabaja la expresión regular (véase la sección 1.2.8):

```
pl@nereida:~/Lperltesting$ cat -n simpleexpressions.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1
use v5.10;
use strict;
use warnings;

use re ’eval’; # to allow Eval-group at runtime

local our ($skip, $term, $expr);
$skip = qr/\s*/;
$expr = qr{ (?<EXPR>
                   (?<TERM>              # An expression is a TERM  ...
                          $skip (?<ID>[a-zA-Z]+)  (?{ print "[ID $+{ID}] "  })
                        | $skip (?<INT>[1-9]\d*)  (?{ print "[INT $+{INT}] " })
                        | $skip \(                (?{ print "[(] " })
                          $skip  (?&EXPR)
                          $skip \)                (?{ print "[)] " })
                   ) (?: $skip           # possibly followed by a sequence of ...
                         (?<OP>[-+*/])            (?{ print "[OP $+{OP}] " })
                         (?&TERM)        # ... operand TERM pairs
                     )*
            )
          }x;
my $re = qr/^ $expr $skip \z/x;
sub is_valid { shift =~ /$re/o }

my @test = ( ’(a + 3)’, ’(3 * 4)+(b + x)’, ’(5 - a)*z’,
              ’((5 - a))*((((z)))+2)’, ’3 + 2’, ’!3 + 2’, ’3 + 2!’,
              ’3 a’, ’3 3’, ’3 * * 3’,
              ’2 - 3 * 4’,  ’2 - 3 + 4’,
            );
foreach (@test) {
  say("$_:");
  say(is_valid($_) ? "\n<$_> is valid" : "\n<$_> is not valid")
}
```

Ejecución:

```
pl@nereida:~/Lperltesting$ ./simpleexpressions.pl
(a + 3):
[(] [ID a] [OP +] [INT 3] [)]
<(a + 3)> is valid
(3 * 4)+(b + x):
[(] [INT 3] [OP *] [INT 4] [)] [OP +] [(] [ID b] [OP +] [ID x] [)]
<(3 * 4)+(b + x)> is valid
(5 - a)*z:
[(] [INT 5] [OP -] [ID a] [)] [OP *] [ID z]
<(5 - a)*z> is valid
((5 - a))*((((z)))+2):
[(] [(] [INT 5] [OP -] [ID a] [)] [)] [OP *] [(] [(] [(] [(] [ID z] [)] [)] [)] [OP +] [INT 2]
<((5 - a))*((((z)))+2)> is valid
3 + 2:
[INT 3] [OP +] [INT 2]
<3 + 2> is valid
```

```
!3 + 2:

<!3 + 2> is not valid
3 + 2!:
[INT 3] [OP +] [INT 2]
<3 + 2!> is not valid
3 a:
[INT 3]
<3 a> is not valid
3 3:
[INT 3]
<3 3> is not valid
3 * * 3:
[INT 3] [OP *]
<3 * * 3> is not valid
2 - 3 * 4:
[INT 2] [OP -] [INT 3] [OP *] [INT 4]
<2 - 3 * 4> is valid
2 - 3 + 4:
[INT 2] [OP -] [INT 3] [OP +] [INT 4]
<2 - 3 + 4> is valid
```

1.2.6. Cuantificadores Posesivos

Por defecto, cuando un subpatrón con un cuantificador impide que el patrón global tenga éxito, se produce un backtrack. Hay ocasiones en las que esta conducta da lugar a ineficiencia.

Perl 5.10 provee los cuantificadores posesivos: Un cuantificador posesivo actúa como un cuantificador greedy pero no se produce backtracking.

`*+`	Casar 0 o mas veces y no retroceder
`++`	Casar 1 o mas veces y no retroceder
`?+`	Casar 0 o 1 veces y no retroceder
`{n}+`	Casar exactamente n veces y no retroceder (redundante)
`{n,}+`	Casar al menos n veces y no retroceder
`{n,m}+`	Casar al menos n veces y no mas de m veces y no retroceder

Por ejemplo, la cadena `’aaaa’` no casa con `/(a++a)/` porque no hay retroceso después de leer las 4 aes:

```
pl@nereida:˜/Lperltesting$ perl5.10.1 -wde 0
main::(-e:1):   0
  DB<1> x ’aaaa’ =˜ /(a+a)/
0  ’aaaa’
  DB<2> x ’aaaa’ =˜ /(a++a)/
  empty array
```

Cadenas Delimitadas por Comillas Dobles

Los operadores posesivos sirven para poder escribir expresiones regulares mas eficientes en aquellos casos en los que sabemos que el retroceso no conducirá a nuevas soluciones, como es el caso del reconocimiento de las cadenas delimitadas por comillas dobles:

```
pl@nereida:˜/Lperltesting$ cat -n ./quotedstrings.pl
     1  #!/usr/local/lib/perl/5.10.1/bin//perl5.10.1
     2  use v5.10;
     3
     4  my $regexp = qr/
     5    "              # double quote
```

```
  (?:           # no memory
      [^"\\]++  # no " or escape: Don't backtrack
    | \\.       # escaped character
  )*+
  "             # end double quote
/x;

my $input = <>;
chomp($input);
if ($input =~ $regexp) {
  say "$& is a string";
}
else {
  say "does not match";
}
```

Paréntesis Posesivos

Los paréntesis posesivos `(?> ...)` dan lugar a un reconocedor que rechaza las demandas de retroceso. De hecho, los operadores posesivos pueden ser reescritos en términos de los paréntesis posesivos: La notación `X++` es equivalente a `(?>X+)`.

Paréntesis Balanceados

El siguiente ejemplo reconoce el lenguaje de los paréntesis balanceados:

```
pl@nereida:~/Lperltesting$ cat -n ./balancedparenthesis.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1
use v5.10;

my $regexp =
    qr/^(
           [^()]*+ # no hay paréntesis, no backtrack
          \(
              (?>        # subgrupo posesivo
                 [^()]++ # no hay paréntesis, + posesivo, no backtrack
                |(?1)    # o es un paréntesis equilibrado
              )*
           \)
           [^()]*+ # no hay paréntesis
         )$/x;

my $input = <>;
chomp($input);
if ($input =~ $regexp) {
  say "$& is a balanced parenthesis";
}
else {
  say "does not match";
}
```

Cuando se ejecuta produce una salida como:

```
pl@nereida:~/Lperltesting$ ./balancedparenthesis.pl
(2*(3+4)-5)*2
(2*(3+4)-5)*2 is a balanced parenthesis
```

```
pl@nereida:~/Lperltesting$ ./balancedparenthesis.pl
(2*(3+4)-5))*2
does not match
pl@nereida:~/Lperltesting$ ./balancedparenthesis.pl
2*(3+4
does not match
pl@nereida:~/Lperltesting$ ./balancedparenthesis.pl
4*(2*(3+4)-5)*2
4*(2*(3+4)-5)*2 is a balanced parenthesis
```

Encontrando los bloques de un programa

El uso de los operadores posesivos nos permite reescribir la solución al problema de encontrar los bloques maximales de un código dada en la sección 1.2.5 de la siguiente manera:

```
pl@nereida:~/Lperltesting$ cat blocksopti.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1 -w
use v5.10;
use strict;
#use re 'debug';

my $rb = qr{(?x)
    (
      \{               # llave abrir
         (?:
             [^{}]++   # no llaves
         |
             (?1)      # recursivo
             [^{}]*+   # no llaves
         )*+
       \}              # llave cerrar
    )
  };

local $/ = undef;
my $input = <>;
my@blocks = $input =~ m{$rb}g;
my $i = 0;
say($i++.":\n$_\n===") for @blocks;
```

Véase también

- Possessive Quantifiers en http://www.regular-expressions.info/
- Nodo *Possessive Quantifiers in Perl 5.10 regexps* en PerlMonks
- perldoc `perlre`

1.2.7. Perl 5.10: Numeración de los Grupos en Alternativas

A veces conviene tener una forma de acceso uniforme a la lista proporcionada por los paréntesis con memoria. Por ejemplo, la siguiente expresión regular reconoce el lenguaje de las horas en notaciones civil y militar:

```
pl@nereida:~/Lperltesting$ perl5.10.1 -wde 0
main::(-e:1):   0
```

```
  DB<1> ’23:12’ =~ /(\d\d|\d):(\d\d)|(\d\d)(\d\d)/; print "1->$1 2->$2\n"
1->23 2->12

  DB<2> ’2312’ =~ /(\d\d|\d):(\d\d)|(\d\d)(\d\d)/; print "3->$3 4->$4\n"
3->23 4->12
```

Parece inconveniente tener los resultados en variables distintas. El constructo `(?| ...)` hace que los paréntesis se enumeren relativos a las alternativas:

```
  DB<3> ’2312’ =~ /(?|(\d\d|\d):(\d\d)|(\d\d)(\d\d))/; print "1->$1 2->$2\n"
1->23 2->12

  DB<4> ’23:12’ =~ /(?|(\d\d|\d):(\d\d)|(\d\d)(\d\d))/; print "1->$1 2->$2\n"
1->23 2->12
```

Ahora en ambos casos `$1` y `$2` contienen las horas y minutos.

1.2.8. Ejecución de Código dentro de una Expresión Regular

Es posible introducir código Perl dentro de una expresión regular. Para ello se usa la notación `(?{code})`.

El siguiente texto esta tomado de la sección ’A-bit-of-magic:-executing-Perl-code-in-a-regular-expression’ en `perlretut`:

> *Normally, regexps are a part of Perl expressions. Code evaluation expressions turn that around by allowing arbitrary Perl code to be a part of a regexp. A code evaluation expression is denoted (?code), with code a string of Perl statements.*
>
> *Be warned that this feature is considered experimental, and may be changed without notice.*
>
> *Code expressions are zero-width assertions, and the value they return depends on their environment.*
>
> *There are two possibilities: either the code expression is used as a conditional in a conditional expression* `(?(condition)...)`*, or it is not.*
>
> - *If the code expression is a conditional, the code is evaluated and the result (i.e., the result of the last statement) is used to determine truth or falsehood.*
> - *If the code expression is not used as a conditional, the assertion always evaluates true and the result is put into the special variable* `$^R` *. The variable* `$^R` *can then be used in code expressions later in the regexp*

Resultado de la última ejecución

Las expresiones de código son *zero-width assertions*: no consumen entrada. El resultado de la ejecución se salva en la variable especial `$^R`.

Veamos un ejemplo:

```
pl@nereida:~/Lperltesting$ perl5.10.1 -wde 0
main::(-e:1):   0
  DB<1> $x = "abcdef"
  DB<2> $x =~ /abc(?{ "Hi mom\n" })def(?{ print $^R })$/
Hi mom
  DB<3> $x =~ /abc(?{ print "Hi mom\n"; 4 })def(?{ print "$^R\n" })/
Hi mom
4
  DB<4> $x =~ /abc(?{ print "Hi mom\n"; 4 })ddd(?{ print "$^R\n" })/ # does not match
  DB<5>
```

En el último ejemplo (línea `DB<4>`) ninguno de los `print` se ejecuta dado que no hay matching.

El Código empotrado no es interpolado
Tomado de la sección 'Extended-Patterns' en `perlre`:

> *This zero-width assertion evaluates any embedded Perl code. It always succeeds, and its code is not interpolated. Currently, the rules to determine where the code ends are somewhat convoluted.*

Contenido del último paréntesis y la variable por defecto en acciones empotradas
Tomado de la sección 'Extended-Patterns' en `perlre`:

> *. . . can be used with the special variable* `$^N` *to capture the results of submatches in variables without having to keep track of the number of nested parentheses. For example:*

```
pl@nereida:~/Lperltesting$ perl5.10.1 -wdE 0
main::(-e:1):   0
  DB<1> $x = "The brown fox jumps over the lazy dog"
  DB<2> x $x =~ /the (\S+)(?{ $color = $^N }) (\S+)(?{ $animal = $^N })/i
0  'brown'
1  'fox'
  DB<3> p "color=$color animal=$animal\n"
color=brown animal=fox
  DB<4> $x =~ /the (\S+)(?{ print (substr($_,0,pos($_)))."\n" }) (\S+)/i
The brown
```

> *Inside the* `(?{...})` *block,* `$_` *refers to the string the regular expression is matching against. You can also use* `pos()` *to know what is the current position of matching within this string.*

Los cuantificadores y el código empotrado
Si se usa un cuantificador sobre un código empotrado, actúa como un bucle:

```
pl@nereida:~/Lperltesting$ perl5.10.1 -wde 0
main::(-e:1):   0
  DB<1> $x = "aaaa"
  DB<2>  $x =~ /(a(?{ $c++ }))*/
  DB<3> p $c
4
  DB<4> $y = "abcd"
  DB<5> $y =~ /(?:(.)(?{ print "-$1-\n" }))*/
-a-
-b-
-c-
-d-
```

Ámbito
Tomado (y modificado el ejemplo) de la sección 'Extended-Patterns' en `perlre`:

> *. . . The code is properly scoped in the following sense: If the assertion is backtracked (compare la sección 'Backtracking' en* `perlre`*), all changes introduced after localization are undone, so that*

```
pl@nereida:~/Lperltesting$ cat embededcodescope.pl
  use strict;

  our ($cnt, $res);
```

```
 sub echo {
local our $pre = substr($_,0,pos($_));
local our $post = (pos($_) < length)? (substr($_,1+pos($_))) : '';

print("$pre(count = $cnt)$post\n");
 }

 $_ = 'a' x 8;
 m<
(?{ $cnt = 0 }) # Initialize $cnt.
(
  a
  (?{
    local $cnt = $cnt + 1; # Update $cnt, backtracking-safe.
    echo();
  })
)*
aaaa
(?{ $res = $cnt }) # On success copy to non-localized
# location.
 >x;

 print "FINAL RESULT: cnt = $cnt res =$res\n";
```

will set `$res = 4` *. Note that after the match,* `$cnt` *returns to the globally introduced value, because the scopes that restrict local operators are unwound.*

```
pl@nereida:~/Lperltesting$ perl5.8.8 -w embededcodescope.pl
a(count = 1)aaaaaa
aa(count = 2)aaaaa
aaa(count = 3)aaaa
aaaa(count = 4)aaa
aaaaa(count = 5)aa
aaaaaa(count = 6)a
aaaaaaa(count = 7)
aaaaaaaa(count = 8)
FINAL RESULT: cnt = 0 res =4
```

Caveats

- *Due to an unfortunate implementation issue, the Perl code contained in these blocks is treated as a compile time closure that can have seemingly bizarre consequences when used with lexically scoped variables inside of subroutines or loops. There are various workarounds for this, including simply using global variables instead. If you are using this construct and strange results occur then check for the use of lexically scoped variables.*
- *For reasons of security, this construct is forbidden if the regular expression involves run-time interpolation of variables, unless the perilous* `use re 'eval'` *pragma has been used (see re), or the variables contain results of* `qr//` *operator (see* `"qr/STRING/imosx"` *in* `perlop`*).*

  This restriction is due to the wide-spread and remarkably convenient custom of using run-time determined strings as patterns. For example:

```
1. $re = <>;
2. chomp $re;
3. $string =~ /$re/;
```

Before Perl knew how to execute interpolated code within a pattern, this operation was completely safe from a security point of view, although it could raise an exception from an illegal pattern. If you turn on the `use re 'eval'` *, though, it is no longer secure, so you should only do so if you are also using taint checking. Better yet, use the carefully constrained evaluation within a Safe compartment. See perlsec for details about both these mechanisms. (Véase la sección 'Taint-mode' en* `perlsec`*)*

- *Because Perl's regex engine is currently* not re-entrant*, interpolated code may not invoke the regex engine either directly with* `m//` *or* `s///`*, or indirectly with functions such as* `split`*.*

Depurando con código empotrado Colisiones en los Nombres de las Subexpresiones Regulares

Las acciones empotradas pueden utilizarse como mecanismo de depuración y de descubrimiento del comportamiento de nuestras expresiones regulares.

En el siguiente programa se produce una colisión entre los nombres `<i>` y `<j>` de los patrones que ocurren en el patrón `<expr>` y en el patrón principal:

```
pl@nereida:~/Lperltesting$ cat -n clashofnamedofssets.pl
  1     #!/usr/local/lib/perl/5.10.1/bin//perl5.10.1
  2     use v5.10;
  3
  4     my $input;
  5
  6     local $" = ", ";
  7
  8     my $parser = qr{
  9         ^ (?<i> (?&expr)) (?<j> (?&expr)) \z
 10           (?{
 11                say "main \$+ hash:";
 12                say " ($_ => $+{$_}) " for sort keys %+;
 13            })
 14
 15         (?(DEFINE)
 16             (?<expr>
 17                 (?<i> . )
 18                 (?<j> . )
 19                   (?{
 20                       say "expr \$+ hash:";
 21                       say " ($_ => $+{$_}) " for sort keys %+;
 22                   })
 23             )
 24         )
 25     }x;
 26
 27     $input = <>;
 28     chomp($input);
 29     if ($input =~ $parser) {
 30       say "matches: ($&)";
 31     }
```

La colisión hace que la salida sea esta:

```
pl@nereida:~/Lperltesting$ ./clashofnamedofssets.pl
abab
expr $+ hash:
 (i => a)
 (j => b)
expr $+ hash:
 (i => ab)
 (j => b)
main $+ hash:
 (i => ab)
 (j => ab)
matches: (abab)
```

Si se evitan las colisiones, se evita la pérdida de información:

```
pl@nereida:~/Lperltesting$ cat -n namedoffsets.pl
  1    #!/usr/local/lib/perl/5.10.1/bin//perl5.10.1
  2    use v5.10;
  3
  4    my $input;
  5
  6    local $" = ", ";
  7
  8    my $parser = qr{
  9        ^ (?<i> (?&expr)) (?<j> (?&expr)) \z
 10          (?{
 11               say "main \$+ hash:";
 12               say " ($_ => $+{$_}) " for sort keys %+;
 13           })
 14
 15        (?(DEFINE)
 16            (?<expr>
 17                (?<i_e> . )
 18                (?<j_e> . )
 19                  (?{
 20                      say "expr \$+ hash:";
 21                      say " ($_ => $+{$_}) " for sort keys %+;
 22                  })
 23            )
 24        )
 25    }x;
 26
 27    $input = <>;
 28    chomp($input);
 29    if ($input =~ $parser) {
 30      say "matches: ($&)";
 31    }
```

que al ejecutarse produce:

```
pl@nereida:~/Lperltesting$ ./namedoffsets.pl
abab
expr $+ hash:
```

```
 (i_e => a)
 (j_e => b)
expr $+ hash:
 (i => ab)
 (i_e => a)
 (j_e => b)
main $+ hash:
 (i => ab)
 (j => ab)
matches: (abab)
```

1.2.9. Expresiones Regulares en tiempo de matching

Los paréntesis especiales:

```
(??{ Código Perl })
```

hacen que el `Código Perl` sea evaluado durante el tiempo de matching. El resultado de la evaluación se trata como una expresión regular. El match continuará intentando casar con la expresión regular retornada.

Paréntesis con memoria dentro de una *pattern code expression*

Los paréntesis en la expresión regular retornada no cuentan en el patrón exterior. Véase el siguiente ejemplo:

```
pl@nereida:~/Lperltesting$ cat -n postponedregexp.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1 -w
use v5.10;
use strict;

my $r = qr{(?x)                # ignore spaces
            ([ab])             # save ’a’ or ’b’ in \$1
            (??{ "($^N)"x3 })  # 3 more of the same as in \$1
          };
say "<$&> lastpar = $#-" if ’bbbb’ =~ $r;
say "<$&> lastpar = $#-" if ’aaaa’ =~ $r;
say "<abab> didn’t match" unless ’abab’ =~ $r;
say "<aaab> didn’t match" unless ’aaab’ =~ $r;
```

Como se ve, hemos accedido desde el código interior al último paréntesis usando `$^N`. Sigue una ejecución:

```
pl@nereida:~/Lperltesting$ ./postponedregexp.pl
<bbbb> lastpar = 1
<aaaa> lastpar = 1
<abab> didn’t match
<aaab> didn’t match
```

Ejemplo: Secuencias de dígitos de longitud especificada por el primer dígito

Consideremos el problema de escribir una expresión regular que reconoce secuencias no vacías de dígitos tales que la longitud de la secuencia restante viene determinada por el primer dígito. Esta es una solución:

```
pl@nereida:~/Lperltesting$ cat -n intints.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1 -w
```

```
use v5.10;
use strict;

my $r = qr{(?x)                # ignore spaces
           (\d)                # a digit
           ( (??{
               "\\d{$^N}"      # as many as the former
             })                # digit says
           )
          };
say "<$&> <$1> <$2>" if '3428' =~ $r;
say "<$&> <$1> <$2>" if '228' =~ $r;
say "<$&> <$1> <$2>" if '14' =~ $r;
say "24 does not match" unless '24' =~ $r;
say "4324 does not match" unless '4324' =~ $r;
```

Cuando se ejecuta se obtiene:

```
pl@nereida:~/Lperltesting$ ./intints.pl
<3428> <3> <428>
<228> <2> <28>
<14> <1> <4>
24 does not match
4324 does not match
```

Ejemplo: Secuencias de dígitos no repetidos

Otro ejemplo: queremos escribir una expresión regular que reconozca secuencias de `$n` dígitos en las que no todos los dígitos se repiten. Donde quizá `$n` es capturado de un paréntesis anterior en la expresión regular. Para simplificar la ilustración de la técnica supongamos que `$n = 7`:

```
pl@nereida:~$ perl5.10.1 -wdE 0
main::(-e:1):   0
  DB<1> x join '', map { "(?!".$_."{7})" } 0..9
0  '(?!0{7})(?!1{7})(?!2{7})(?!3{7})(?!4{7})(?!5{7})(?!6{7})(?!7{7})(?!8{7})(?!9{7})'
  DB<2> x '7777777' =~ /(??{join '', map { "(?!".$_."{7})" } 0..9})(\d{7})/
  empty array
  DB<3> x '7777778' =~ /(??{join '', map { "(?!".$_."{7})" } 0..9})(\d{7})/
0  7777778
  DB<4> x '4444444' =~ /(??{join '', map { "(?!".$_."{7})" } 0..9})(\d{7})/
  empty array
  DB<5> x '4422444' =~ /(??{join '', map { "(?!".$_."{7})" } 0..9})(\d{7})/
0  4422444
```

Palíndromos con independencia del acento

Se trata en este ejercicio de generalizar la expresión regular introducida en la sección 1.2.5 para reconocer los palabra-palíndromos.

Se trata de encontrar una regexp que acepte que la lectura derecha e inversa de una frase en Español pueda diferir en la acentuación (como es el caso del clásico palíndromo *dábale arroz a la zorra el abad*). Una solución trivial es preprocesar la cadena eliminando los acentos. Supondremos sin embargo que se quiere trabajar sobre la cadena original. He aquí una solucion:

```
pl@nereida:~/Lperltesting$ cat actionspanishpalin.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1 -w -CIOEioA
use v5.10;
```

```
use strict;
use utf8;
use re ’eval’;
use Switch;

sub f {
  my $char = shift;

  switch($char) {
    case [ qw{a á} ] { return ’[aá]’ }
    case [ qw{e é} ] { return ’[eé]’ }
    case [ qw{i í} ] { return ’[ií]’ }
    case [ qw{o ó} ] { return ’[oó]’ }
    case [ qw{u ú} ] { return ’[uú]’ }
    else             { return $char  };
  }
}

my $regexp = qr/ˆ(\W* (?:
                            (\w) (?-2)(??{ f($ˆN) })
                          | \w?
                      ) \W*
                  )
                $
               /ix;

my $input = <>; # Try: ’dábale arroz a la zorra el abad’;
chomp($input);
if ($input =˜ $regexp) {
  say "$input is a palindrome";
}
else {
  say "$input does not match";
}
```

Sigue un ejemplo de ejecución:

```
pl@nereida:˜/Lperltesting$ ./actionspanishpalin.pl
dábale arroz a la zorra el abad
dábale arroz a la zorra el abad is a palindrome
pl@nereida:˜/Lperltesting$ ./actionspanishpalin.pl
éoíúaáuioé
éoíúaáuioé is a palindrome
pl@nereida:˜/Lperltesting$ ./actionspanishpalin.pl
dáed
dáed does not match
```

Postponiendo para conseguir recursividad

Véase el nodo Complex regex for maths formulas para la formulación del problema:

> *Hiya monks,*
>
> *Im having trouble getting my head around a regular expression to match sequences. I need to catch all exceptions where a mathematical expression is illegal...*

> *There must be either a letter or a digit either side of an operator parenthesis must open and close next to letters or digits, not next to operators, and do not have to exist variables must not be more than one letter Nothing other than* `a-z,A-Z,0-9,+,-,*,/,(,)` *can be used*
>
> *Can anyone offer a hand on how best to tackle this problem?*
>
> *many thanks*

La respuesta dada por ikegami usa `(?{{ ... })` para conseguir una conducta recursiva en versiones de perl anteriores a la 5.10:

```
pl@nereida:˜/Lperltesting$ cat -n complexformula.pl
  1    #!/usr/bin/perl
  2    use strict;
  3    use warnings;
  4
  5    sub is_valid_expr {
  6       use re ’eval’; # to allow Eval-group at runtime
  7
  8       local our ($skip, $term, $expr);
  9       $skip = qr! \s* !x;
 10       $term = qr! $skip [a-zA-Z]+              # A term is an identifier
 11                 | $skip [1-9][0-9]*            # or a number
 12                 | $skip \( (??{ $expr }) $skip # or an expression
 13                         \)
 14                 !x;
 15       $expr = qr! $term                         # A expr is a term
 16                   (?: $skip [-+*/] $term )*     # or a term + a term ...
 17                 !x;
 18
 19       return $_[0] =˜ / ˆ $expr $skip \z /x;
 20    }
 21
 22    print(is_valid_expr($_) ? "$_ is valid\n" : "$_ is not valid\n") foreach (
 23     ’(a + 3)’,
 24     ’(3 * 4)+(b + x)’,
 25     ’(5 - a)*z’,
 26     ’3 + 2’,
 27
 28     ’!3 + 2’,
 29     ’3 + 2!’,
 30
 31     ’3 a’,
 32     ’3 3’,
 33     ’3 * * 3’,
 34
 35     ’2 - 3 * 4’,
 36     ’2 - 3 + 4’,
 37    );
```

Sigue el resultado de la ejecución:

```
pl@nereida:˜/Lperltesting$ perl complexformula.pl
(a + 3) is valid
(3 * 4)+(b + x) is valid
```

```
(5 - a)*z is valid
3 + 2 is valid
!3 + 2 is not valid
3 + 2! is not valid
3 a is not valid
3 3 is not valid
3 * * 3 is not valid
2 - 3 * 4 is valid
2 - 3 + 4 is valid
```

Caveats

Estos son algunos puntos a tener en cuenta cuando se usan patrones postpuestos. Véase la entrada `(??{ code })` en la sección 'Extended-Patterns' en `perlre`:

> *WARNING: This extended regular expression feature is considered experimental, and may be changed without notice. Code executed that has side effects may not perform identically from version to version due to the effect of future optimisations in the regex engine.*
>
> *This is a* postponed regular subexpression*. The code is evaluated at run time, at the moment this subexpression may match. The result of evaluation is considered as a regular expression and matched as if it were inserted instead of this construct.*
>
> *The code is not interpolated.*
>
> *As before, the rules to determine where the code ends are currently somewhat convoluted.*
>
> *Because perl's regex engine is not currently re-entrant, delayed code may not invoke the regex engine either directly with* `m//` *or* `s///`*), or indirectly with functions such as* `split`*.*
>
> *Recursing deeper than 50 times without consuming any input string will result in a fatal error. The maximum depth is compiled into perl, so changing it requires a custom build.*

1.2.10. Expresiones Condicionales

Citando a `perlre`:

> *A conditional expression is a form of if-then-else statement that allows one to choose which patterns are to be matched, based on some condition.*
>
> *There are two types of conditional expression:* `(?(condition)yes-regexp)` *and* `(?(condition)yes-rege`
>
> `(?(condition)yes-regexp)` *is like an* `if () {}` *statement in Perl. If the condition is true, the yes-regexp will be matched. If the condition is false, the yes-regexp will be skipped and Perl will move onto the next regexp element.*
>
> *The second form is like an* `if () {} else {}` *statement in Perl. If the condition is true, the yes-regexp will be matched, otherwise the no-regexp will be matched.*
>
> *The condition can have several forms.*
>
> - *The first form is simply an integer in parentheses (integer). It is true if the corresponding backreference* `\integer` *matched earlier in the regexp. The same thing can be done with a name associated with a capture buffer, written as* `(<name>)` *or* `('name')`*.*
> - *The second form is a bare zero width assertion* `(?...)`*, either a lookahead, a lookbehind, or a code assertion.*
> - *The third set of forms provides tests that return true if the expression is executed within a recursion* `(R)` *or is being called from some capturing group, referenced either by number* `(R1`*, or by name* `(R&name)`*.*

Condiciones: número de paréntesis

Una expresión condicional puede adoptar diversas formas. La mas simple es un entero en paréntesis. Es cierta si la correspondiente referencia `\integer` casó (también se puede usar un nombre si se trata de un paréntesis con nombre).

En la expresión regular `/^(.)(..)?(?(2)a|b)/` si el segundo paréntesis casa, la cadena debe ir seguida de una `a`, si no casa deberá ir seguida de una `b`:

```
  DB<1> x 'hola' =~ /^(.)(..)?(?(2)a|b)/
0  'h'
1  'ol'
  DB<2> x 'ha' =~ /^(.)(..)?(?(2)a|b)/
  empty array
  DB<3> x 'hb' =~ /^(.)(..)?(?(2)a|b)/
0  'h'
1  undef
```

Ejemplo: cadenas de la forma *una*-otra-otra-*una*

La siguiente búsqueda casa con patrones de la forma `$x$x` o `$x$y$y$x`:

```
pl@nereida:~/Lperltesting$ perl5.10.1 -wde 0
main::(-e:1):   0
  DB<1> x 'aa' =~ m{^(\w+)(\w+)?(?(2)\2\1|\1)$}
0  'a'
1  undef
  DB<2> x 'abba' =~ m{^(\w+)(\w+)?(?(2)\2\1|\1)$}
0  'a'
1  'b'
  DB<3> x 'abbc' =~ m{^(\w+)(\w+)?(?(2)\2\1|\1)$}
  empty array
  DB<4> x 'juanpedropedrojuan' =~ m{^(\w+)(\w+)?(?(2)\2\1|\1)$}
0  'juan'
1  'pedro'
```

Condiciones: Código

Una expresión condicional también puede ser un código:

```
  DB<1> $a = 0; print "$&" if 'hola' =~ m{(?(?{$a})hola|adios)} # No hay matching

  DB<2> $a = 1; print "$&" if 'hola' =~ m{(?(?{$a})hola|adios)}
hola
```

Ejemplo: Cadenas con posible paréntesis inicial (no anidados)

La siguiente expresión regular utiliza un condicional para forzar a que si una cadena comienza por un paréntesis abrir termina con un paréntesis cerrar. Si la cadena no comienza por paréntesis abrir no debe existir un paréntesis final de cierre:

```
pl@nereida:~/Lperltesting$ cat -n conditionalregexp.pl
   1    #!/usr/local/lib/perl/5.10.1/bin//perl5.10.1 -w
   2    use v5.10;
   3    use strict;
   4
   5    my $r = qr{(?x)                # ignore spaces
   6                ^
   7                ( \( )?            # may be it comes an open par
```

```
            [^()]+                 # no parenthesis
            (?(1)                  # did we sart with par?
              \)                   # if yes then close par
            )
            $
          };
say "<$&>" if ’(abcd)’ =~ $r;
say "<$&>" if ’abc’ =~ $r;
say "<(abc> does not match" unless ’(abc’ =~ $r;
say "<abc)> does not match" unless ’abc)’ =~ $r;
```

Al ejecutar este programa se obtiene:

```
pl@nereida:~/Lperltesting$ ./conditionalregexp.pl
<(abcd)>
<abc>
<(abc> does not match
<abc)> does not match
```

Expresiones Condicionales con `(R)`

El siguiente ejemplo muestra el uso de la condición `(R)`, la cual comprueba si la expresión ha sido evaluada dentro de una recursión:

```
pl@nereida:~/Lperltesting$ perl5.10.1 -wdE 0
main::(-e:1):   0
  DB<1> x ’bbaaaabb’ =~ /(b(?(R)a+|(?0))b)/
0  ’bbaaaabb’
  DB<2> x ’bb’ =~ /(b(?(R)a+|(?0))b)/
  empty array
  DB<3> x ’bab’ =~ /(b(?(R)a+|(?0))b)/
  empty array
  DB<4> x ’bbabb’ =~ /(b(?(R)a+|(?0))b)/
0  ’bbabb’
```

La sub-expresión regular `(?(R)a+|(?0))` dice: si esta siendo evaluada recursivamente admite `a+` si no, evalúa la regexp completa recursivamente.

Ejemplo: Palíndromos con Equivalencia de Acentos Españoles

Se trata en este ejercicio de generalizar la expresión regular introducida en la sección 1.2.5 para reconocer los palabra-palíndromos[7]. Se trata de encontrar una regexp que acepte que la lectura derecha e inversa de una frase en Español pueda diferir en la acentuación (como es el caso del clásico palíndromo *dábale arroz a la zorra el abad*). Una solución trivial es preprocesar la cadena eliminando los acentos. Supondremos sin embargo que se quiere trabajar sobre la cadena original. He aquí una solucion parcial (por consideraciones de legibilidad sólo se consideran las vocales `a` y `o`:

```
pl@nereida:~/Lperltesting$ cat spanishpalin.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1 -w -CIOEioA
use v5.10;
use strict;
use utf8;

my $regexp = qr/^(?<pal>\W* (?:
                            (?<L>(?<a>[áa])|(?<e>[ée])|\w) # letter
```

---

[7] No sé si existe el término. Significa que la lectura directa y la inversa pueden diferir en los signos de puntuación

```
                            (?&pal)                        # nested palindrome
                            (?(<a>)[áa]                    # if is an "a" group
                                  |(?:((?<e>)[ée]          # if is an "e" group
                                            |\g{L}         # exact match
                                      )                    # end if [ée]
                                   )                       # end group
                            )                              # end if [áa]
                          | \w?                            # non rec. case
                      ) \W*                                # punctuation symbols
                  )
                $
               /ix;

my $input = <>; # Try: ’dábale arroz a la zorra el abad’;
chomp($input);
if ($input =˜ $regexp) {
  say "$input is a palindrome";
}
else {
  say "$input does not match";
}
```

Ejecución:

```
pl@nereida:˜/Lperltesting$ ./spanishpalin.pl
dábale arroz a la zorra el abad
dábale arroz a la zorra el abad is a palindrome
pl@nereida:˜/Lperltesting$ ./spanishpalin.pl
óuuo
óuuo does not match
pl@nereida:˜/Lperltesting$ ./spanishpalin.pl
éaáe
éaáe is a palindrome
```

Hemos usado la opción `-CIOEioA` para asegurarnos que los ficheros de entrada/saldia y error y la línea de comandos estan en modo UTF-8. (Véase la sección 1.3)

Esto es lo que dice la documentación de `perlrun` al respecto:

> *The* `-C` *flag controls some of the Perl Unicode features.*
>
> *As of 5.8.1, the* `-C` *can be followed either by a number or a list of option letters. The letters, their numeric values, and effects are as follows; listing the letters is equal to summing the numbers.*

```
I 1 STDIN is assumed to be in UTF-8
O 2 STDOUT will be in UTF-8
E 4 STDERR will be in UTF-8
S 7 I + O + E
i 8 UTF-8 is the default PerlIO layer for input streams
o 16 UTF-8 is the default PerlIO layer for output streams
D 24 i + o
A 32 the @ARGV elements are expected to be strings encoded
in UTF-8
L 64 normally the "IOEioA" are unconditional,
the L makes them conditional on the locale environment
```

```
variables (the LC_ALL, LC_TYPE, and LANG, in the order
of decreasing precedence) -- if the variables indicate
UTF-8, then the selected "IOEioA" are in effect
a 256 Set ${^UTF8CACHE} to -1, to run the UTF-8 caching code in
debugging mode.
```

> *For example,* `-COE` *and* `-C6` *will both turn on UTF-8-ness on both* `STDOUT` *and* `STDERR`*. Repeating letters is just redundant, not cumulative nor toggling.*
>
> *The io options mean that any subsequent* `open()` *(or similar I/O operations) will have the* `:utf8` *PerlIO layer implicitly applied to them, in other words,* `UTF-8` *is expected from any input stream, and* `UTF-8` *is produced to any output stream. This is just the default, with explicit layers in* `open()` *and with* `binmode()` *one can manipulate streams as usual.*
>
> `-C` *on its own (not followed by any number or option list), or the empty string* `""` *for the* `PERL_UNICODE` *environment variable, has the same effect as* `-CSDL` *. In other words, the standard I/O handles and the default*`open()` *layer are UTF-8-fied but only if the locale environment variables indicate a UTF-8 locale. This behaviour follows the implicit (and problematic) UTF-8 behaviour of Perl 5.8.0.*
>
> *You can use* `-C0` *(or* `0` *for* `PERL_UNICODE` *) to explicitly disable all the above Unicode features.*

El pragma `use utf8` hace que se utilice una semántica de carácteres (por ejemplo, la regexp `/./` casará con un carácter unicode), el pragma `use bytes` cambia de semántica de caracteres a semántica de bytes (la regexp `.` casará con un byte).

1.2.11. Verbos que controlan el retroceso

El verbo de control `(*FAIL)`

Tomado de la sección ’Backtracking-control-verbs’ en `perlretut`:

> *The control verb* `(*FAIL)` *may be abbreviated as* `(*F)`*. If this is inserted in a regexp it will cause to fail, just like at some mismatch between the pattern and the string. Processing of the regexp continues like after any ”normal”failure, so that the next position in the string or another alternative will be tried. As failing to match doesn’t preserve capture buffers or produce results, it may be necessary to use this in combination with embedded code.*

```
pl@nereida:~/Lperltesting$ cat -n vowelcount.pl
     1  #!/usr/local/lib/perl/5.10.1/bin//perl5.10.1  -w
     2  use strict;
     3
     4  my $input = shift() || <STDIN>;
     5  my %count = ();
     6  $input =~ /([aeiou])(?{ $count{$1}++; })(*FAIL)/i;
     7  printf("’%s’ => %3d\n", $_, $count{$_})  for (sort keys %count);
```

Al ejecutarse con entrada `supercalifragilistico` produce la salida:

```
pl@nereida:~/Lperltesting$ ./vowelcount.pl
supercalifragilistico
’a’ =>   2
’e’ =>   1
’i’ =>   4
’o’ =>   1
’u’ =>   1
```

Ejercicio 1.2.5. *¿Que queda en* `$1` *depués de ejecutado el matching* `$input =~ /([aeiou])(?{ $count{$1}++;`

Véase también:

- El nodo en PerlMonks *The Oldest Plays the Piano*
- Véase el ejercicio *Las tres hijas* en la sección 1.5.4

El verbo de control `(*ACCEPT)`

Tomado de `perlretut`:

> *This pattern matches nothing and causes the end of successful matching at the point at which the* `(*ACCEPT)` *pattern was encountered, regardless of whether there is actually more to match in the string. When inside of a nested pattern, such as recursion, or in a subpattern dynamically generated via* `(??{})`*, only the innermost pattern is ended immediately.*
>
> *If the* `(*ACCEPT)` *is inside of capturing buffers then the buffers are marked as ended at the point at which the* `(*ACCEPT)` *was encountered. For instance:*

```
  DB<1> x ’AB’ =˜ /(A (A|B(*ACCEPT)|C) D)(E)/x
0  ’AB’
1  ’B’
2  undef
  DB<2> x ’ACDE’  =˜ /(A (A|B(*ACCEPT)|C) D)(E)/x
0  ’ACD’
1  ’C’
2  ’E’
```

El verbo `SKIP`

> *This zero-width pattern prunes the backtracking tree at the current point when backtracked into on failure. Consider the pattern* `A (*SKIP) B`*, where* `A` *and* `B` *are complex patterns. Until the* `(*SKIP)` *verb is reached,* `A` *may backtrack as necessary to match. Once it is reached, matching continues in* `B`*, which may also backtrack as necessary; however, should* `B` *not match, then no further backtracking will take place, and the pattern will fail outright at the current starting position.*
>
> *It also signifies that whatever text that was matched leading up to the* `(*SKIP)` *pattern being executed cannot be part of any match of this pattern. This effectively means that the regex engine* `skips` *forward to this position on failure and tries to match again, (assuming that there is sufficient room to match).*
>
> *The name of the* `(*SKIP:NAME)` *pattern has special significance. If a* `(*MARK:NAME)` *was encountered while matching, then it is that position which is used as the "skip point". If no* `(*MARK)` *of that name was encountered, then the* `(*SKIP)` *operator has no effect. When used without a name the "skip point"is where the match point was when executing the* `(*SKIP)` *pattern.*

Ejemplo:

```
pl@nereida:˜/Lperltesting$ cat -n SKIP.pl
     1  #!/soft/perl5lib/bin/perl5.10.1 -w
     2  use strict;
     3  use v5.10;
     4
     5  say "NO SKIP: /a+b?(*FAIL)/";
     6  our $count = 0;
     7  ’aaab’ =˜ /a+b?(?{print "$&\n"; $count++})(*FAIL)/;
     8  say "Count=$count\n";
     9
```

```
say "WITH SKIP: a+b?(*SKIP)(*FAIL)/";
$count = 0;
’aaab’ =~ /a+b?(*SKIP)(?{print "$&\n"; $count++})(*FAIL)/;
say "WITH SKIP: Count=$count\n";

say "WITH SKIP /a+(*SKIP)b?(*FAIL)/:";
$count = 0;
’aaab’ =~ /a+(*SKIP)b?(?{print "$&\n"; $count++})(*FAIL)/;
say "Count=$count\n";

say "WITH SKIP /(*SKIP)a+b?(*FAIL): ";
$count = 0;
’aaab’ =~ /(*SKIP)a+b?(?{print "$&\n"; $count++})(*FAIL)/;
say "Count=$count\n";
```

Ejecución:

```
pl@nereida:~/Lperltesting$ perl5.10.1 SKIP.pl
NO SKIP: /a+b?(*FAIL)/
aaab
aaa
aa
a
aab
aa
a
ab
a
Count=9

WITH SKIP: a+b?(*SKIP)(*FAIL)/
aaab
WITH SKIP: Count=1

WITH SKIP /a+(*SKIP)b?(*FAIL)/:
aaab
aaa
Count=2

WITH SKIP /(*SKIP)a+b?(*FAIL):
aaab
aaa
aa
a
aab
aa
a
ab
a
Count=9
```

Marcas

Tomado de la sección ’Backtracking-control-verbs’ en `perlretut`:

> `(*MARK:NAME) (*:NAME)`
>
> *This zero-width pattern can be used to mark the point reached in a string when a certain part of the pattern has been successfully matched. This mark may be given a name. A later* `(*SKIP)` *pattern will then skip forward to that point if backtracked into on failure. Any number of* `(*MARK)` *patterns are allowed, and the* `NAME` *portion is optional and may be duplicated.*
>
> *In addition to interacting with the* `(*SKIP)` *pattern,* `(*MARK:NAME)` *can be used to* `label` *a pattern branch, so that after matching, the program can determine which branches of the pattern were involved in the match.*
>
> *When a match is successful, the* `$REGMARK` *variable will be set to the name of the most recently executed* `(*MARK:NAME)` *that was involved in the match.*
>
> *This can be used to determine which branch of a pattern was matched without using a separate capture buffer for each branch, which in turn can result in a performance improvement.*
>
> *When a match has failed, and unless another verb has been involved in failing the match and has provided its own name to use, the* `$REGERROR` *variable will be set to the name of the most recently executed* `(*MARK:NAME)`.

```
pl@nereida:~/Lperltesting$ cat -n mark.pl
 1  use v5.10;
 2  use strict;
 3
 4  our $REGMARK;
 5
 6  $_ = shift;
 7  say $REGMARK if /(?:x(*MARK:mx)|y(*MARK:my)|z(*MARK:mz))/;
 8  say $REGMARK if /(?:x(*:xx)|y(*:yy)|z(*:zz))/;
```

Cuando se ejecuta produce:

```
pl@nereida:~/Lperltesting$ perl5.10.1 mark.pl y
my
yy
pl@nereida:~/Lperltesting$ perl5.10.1 mark.pl z
mz
zz
```

Poniendo un espacio después de cada signo de puntuación

Se quiere poner un espacio en blanco después de la aparición de cada coma:

```
s/,/, /g;
```

pero se quiere que la sustitución no tenga lugar si la coma esta incrustada entre dos dígitos. Además se pide que si hay ya un espacio después de la coma, no se duplique. Sigue una solución que usa marcas:

```
pl@nereida:~/Lperltesting$ perl5.10.1 -wdE 0
main::(-e:1):   0
  DB<1> $a = ’ab,cd, ef,12,34,efg,56,78,df, ef,’
  DB<2> x ($b = $a) =~ s/\d,\d(*:d)|,(?!\s)/($REGMARK eq ’d’)? $& : ’, ’/ge
0  8
  DB<3> p "<$b>"
<ab, cd, ef, 12,34, efg, 56,78, df, ef, >
```

1.3. Unicode

Introducción

Ejercicio 1.3.1. *Antes de comenzar esta sección, lea los siguientes documentos:*

- *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)*

La siguiente introducción esta tomada de la sección la sección 'Unicode' en `perluniintro`:

> Unicode *is a character set standard which plans to codify all of the writing systems of the world, plus many other symbols.*
>
> *Unicode and ISO/IEC 10646 are coordinated standards that provide* code points *for characters in almost all modern character set standards, covering more than 30 writing systems and hundreds of languages, including all commercially-important modern languages.*
>
> *All characters in the largest Chinese, Japanese, and Korean dictionaries are also encoded. The standards will eventually cover almost all characters in more than 250 writing systems and thousands of languages. Unicode 1.0 was released in October 1991, and 4.0 in April 2003.*
>
> *A* Unicode character *is an abstract entity. It is not bound to any particular integer width, especially not to the C language char .*
>
> *Unicode is language-neutral and display-neutral: it does not encode the language of the text and it does not generally define fonts or other graphical layout details. Unicode operates on characters and on text built from those characters.*
>
> *Unicode defines characters like* `LATIN CAPITAL LETTER A` *or* `GREEK SMALL LETTER ALPHA` *and unique numbers for the characters, in this case* `0x0041` *and* `0x03B1`*, respectively. These unique numbers are called* code points*.*
>
> *The Unicode standard prefers using hexadecimal notation for the code points.*
>
> *The Unicode standard uses the* **notation** `U+0041 LATIN CAPITAL LETTER A`*, to give the hexadecimal code point and the normative name of the character.*
>
> *Unicode also defines various* Unicode properties *for the characters, like* `uppercase` *or* `lowercase`*,* `decimal digit`*, or* `punctuation`*; these properties are independent of the names of the characters.*
>
> *Furthermore, various operations on the characters like uppercasing, lowercasing, and collating (sorting) are defined.*
>
> *A* Unicode character *consists either of a single code point, or a base character (like* `LATIN CAPITAL LETTER A` *), followed by one or more* **modifiers** *(like* `COMBINING ACUTE ACCENT` *). This sequence of base character and modifiers is called a* combining character sequence.
>
> *Whether to call these combining character sequences çharacters"depends on your point of view. If you are a programmer, you probably would tend towards seeing each element in the sequences as one unit, or çharacter". The whole sequence could be seen as one çharacter", however, from the user's point of view, since that's probably what it looks like in the context of the user's language.*
>
> *With this "whole sequence"view of characters, the total number of characters is open-ended. But in the programmer's .ºne unit is one character"point of view, the concept of çharacters"is more deterministic.*
>
> **In this document, we take that second point of view: one çharacter"is one Unicode code point, be it a base character or a combining character**.
>
> *For some combinations, there are* precomposed characters. `LATIN CAPITAL LETTER A WITH ACUTE` *, for example, is defined as a single code point.*

These precomposed characters are, however, only available for some combinations, and are mainly meant to support round-trip conversions between Unicode and legacy standards (like the ISO 8859).

In the general case, the composing method is more extensible. To support conversion between different compositions of the characters, various normalization forms to standardize representations are also defined.

Because of backward compatibility with legacy encodings, the .^{a} unique number for every character"idea breaks down a bit: instead, there is .^{a}t least one number for every character".

The same character could be represented differently in several legacy encodings.

The converse is also not true: some code points do not have an assigned character.

- *Firstly, there are unallocated code points within otherwise used blocks.*
- *Secondly, there are special Unicode control characters that do not represent true characters.*

A common myth about Unicode is that it would be "16-bit", that is, Unicode is only represented as 0x10000 (or 65536) characters from 0x0000 to 0xFFFF . This is untrue. Since Unicode 2.0 (July 1996), Unicode has been defined all the way up to 21 bits (0x10FFFF), and since Unicode 3.1 (March 2001), characters have been defined beyond 0xFFFF . The first 0x10000 characters are called the Plane 0, or the Basic Multilingual Plane *(*BMP*). With Unicode 3.1, 17 (yes, seventeen)* planes *in all were defined–but they are nowhere near full of defined characters, yet.*

Another myth is that the 256-character blocks have something to do with languages–that each block would define the characters used by a language or a set of languages. This is also untrue. The division into blocks exists, but it is almost completely accidental–an artifact of how the characters have been and still are allocated. Instead, there is a concept called scripts, which is more useful: there is

- *Latin script,*
- *Greek script,*

and so on. Scripts usually span varied parts of several blocks. For further information see `Unicode::UCD`*:*

```
pl@nereida:~/Lperltesting$ perl5.10.1 -wdE 0
main::(-e:1):   0
  DB<1> use Unicode::UCD qw{charinfo charscripts}
  DB<2> x charinfo(0x41)
0  HASH(0xc69a88)
   'bidi' => 'L'
   'block' => 'Basic Latin'
   'category' => 'Lu'
   'code' => 0041
   'combining' => 0
   'comment' => ''
   'decimal' => ''
   'decomposition' => ''
   'digit' => ''
   'lower' => 0061
   'mirrored' => 'N'
   'name' => 'LATIN CAPITAL LETTER A'
   'numeric' => ''
```

```
        'script' => 'Latin'
        'title' => ''
        'unicode10' => ''
        'upper' => ''
      DB<3> x @{charscripts()->{Greek}}[0..3]
    0  ARRAY(0xd676a8)
       0  880
       1  883
       2  'Greek'
    1  ARRAY(0xd86300)
       0  885
       1  885
       2  'Greek'
    2  ARRAY(0xd6c718)
       0  886
       1  887
       2  'Greek'
    3  ARRAY(0xd6c790)
       0  890
       1  890
       2  'Greek'
```

The Unicode code points are just abstract numbers. To input and output these abstract numbers, the numbers must be encoded or serialised somehow. Unicode defines several character encoding forms, of which UTF-8 is perhaps the most popular. UTF-8 is a variable length encoding that encodes Unicode characters as 1 to 6 bytes (only 4 with the currently defined characters). Other encodings include UTF-16 and UTF-32 and their big- and little-endian variants (UTF-8 is byte-order independent) The ISO/IEC 10646 defines the UCS-2 and UCS-4 encoding forms.

1.4. Expresiones Regulares en Otros Lenguajes

Vim

- Learn vi/vim in 50 lines and 15 minutes
- VIM Regular Expressions
- Editing features for advanced users
- Vim documentation: pattern
- Vim Regular Expressions Chart

Java

El siguiente ejemplo muestra un programa estilo `grep`: solicita una expresión regular para aplicarla luego a una serie de entradas leídas desde la entrada estandar.

```
casiano@nereida:~/projects/PA/regexp$ cat -n Application.java
/**
 * javac Application.java
 * java Application
 */

import java.io.*;
```

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class Application {

    public static void main(String[] args){
        String regexp = "";
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        try {
            System.out.print("Enter your regex: ");
            regexp = br.readLine();
        } catch (IOException e) { System.exit(1); };
        while (true) {

            String input = "";
            try {
              System.out.print("Enter input string to search: ");
              input = br.readLine();
            } catch (IOException e) { System.exit(1); };

            Pattern pattern = Pattern.compile(regexp);
            Matcher matcher = pattern.matcher(input);

            boolean found = false;
            while (matcher.find()) {
                System.out.println("I found the text "
                                   + matcher.group()
                                   + " starting at index "
                                   + matcher.start()
                                   + " and ending at index "
                                   +matcher.end()
                );
                found = true;
            }
            if(!found){
                System.out.println("No match found.");
            }
        }
    }
}
```

Ejecución:

```
casiano@nereida:~/Ljavatesting$ java Application
Enter your regex: (\d+).(\d+)
Enter input string to search: a4b5d6c7efg
I found the text 4b5 starting at index 1 and ending at index 4
I found the text 6c7 starting at index 5 and ending at index 8
Enter input string to search: abc
No match found.
Enter input string to search:
```

Véase también Java Regular Expressions

bash

Esta es una versión en `bash` del conversor de temperaturas visto en las secciones anteriores:

```
pl@nereida:~/src/bash$ cat -n f2c
     1  #!/bin/bash
     2  echo "Enter a temperature (i.e. 32F, 100C):";
     3  read input;
     4
     5  if [ -z "$(echo $input | grep -i ’^[-+]\?[0-9]\+\(\.[0-9]*\)\?\ *[CF]$’)" ]
     6  then
     7    echo "Expecting a temperature, so don’t understand \"$input\"." 1>&2;
     8  else
     9    input=$(echo $input | tr -d ’ ’);
    10    InputNum=${input:0:${#input}-1};
    11    Type=${input: -1}
    12
    13    if [ $Type = "c" -o $Type = "C" ]
    14    then
    15     celsius=$InputNum;
    16     fahrenheit=$(echo "scale=2; ($celsius * 9/5)+32" | bc -l);
    17    else
    18     fahrenheit=$InputNum;
    19     celsius=$(echo "scale=2; ($fahrenheit -32)*5/9" | bc -l);
    20    fi
    21
    22    echo "$celsius C = $fahrenheit F";
    23  fi
```

C

```
pl@nereida:~/src/regexpr$ cat -n pcregrep.c
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <assert.h>
 5 #include <pcre.h>
 6
 7 char enter_reverse_mode[] = "\33[7m";
 8 char exit_reverse_mode[] = "\33[0m";
 9
10 int main(int argc, char **argv)
11 {
12   const char *pattern;
13   const char *errstr;
14   int erroffset;
15   pcre *expr;
16   char line[512];
17   assert(argc == 2); /* XXX fixme */
18   pattern = argv[1];
19   if (!(expr = pcre_compile(pattern, 0, &errstr, &erroffset, 0))) {
20     fprintf(stderr, "%s: %s\n", pattern, errstr);
21     return EXIT_FAILURE;
22   }
23   while (fgets(line, sizeof line, stdin)) {
24     size_t len = strcspn(line, "\n");
```

```
    int matches[2];
    int offset = 0;
    int flags = 0;
    line[len] = '\0';
    while (0 < pcre_exec(expr, 0, line, len, offset, flags, matches, 2)) {
      printf("%.*s%s%.*s%s",
        matches[0] - offset, line + offset,
        enter_reverse_mode,
        matches[1] - matches[0], line + matches[0],
        exit_reverse_mode);
      offset = matches[1];
      flags |= PCRE_NOTBOL;
    }
    printf("%s\n", line + offset);
  }
  return EXIT_SUCCESS;
}
```

Compilación:

```
pl@nereida:~/src/regexpr$ gcc -lpcre pcregrep.c -o pcregrep
```

Cuando se ejecuta espera un patrón en la línea de comandos y pasa a leer desde la entrada estandar. Las cadenas que casan se muestran resaltadas:

```
pl@nereida:~/src/regexpr$ ./pcregrep '\d+'
435 otro 23

435 otro 23

hola
hola
```

Python

```
pl@nereida:~/src/python$ cat -n c2f.py
#!/usr/local/bin/python
import re

temp = raw_input( ' Introduzca una temperatura (i.e. 32F, 100C): ' )
pattern = re.compile( "^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$", re.IGNORECASE )
mo = pattern.match( temp )

if mo:
  inputNum = float(mo.group( 1 ))
  type = mo.group( 3 )
  celsius = 0.0
  fahrenheit = 0.0
  if ( type == "C" or type == "c" ) :
    celsius = inputNum
    fahrenheit = ( celsius * 9/5 ) + 32
  else :
    fahrenheit = inputNum
    celsius = ( fahrenheit - 32 ) * 5/9
  print " ", '%.2f'%(celsius), " C = ", '%.2f'%(fahrenheit), " F\n"
else :
  print " Se experaba una temperatura, no se entiende", temp, "\n"
```

Ruby

```
pl@nereida:˜/src/ruby$ cat -n f2c_b
 1  #!/usr/bin/ruby
 2
 3  # Primero leemos una temperatura
 4  class Temperature_calculator
 5    def initialize temp
 6    comp = Regexp.new(’ˆ([-+]?\d+(\.\d*)?)\s*([CFcf])$’)
 7    if temp =˜ comp
 8    begin
 9      cifra = Float($1)
10      @C,@F = ( $3 == "F" or $3 == "f")? [(cifra -32) * 5/9, cifra] : [cifra , cifra * 9/5 +
11    end
12    else
13      raise("Entrada incorrecta")
14    end
15  end
16
17    def show
18      puts "Temperatura en Celsius: #{@C}, temperatura en Fahrenheit: #{@F}"
19    end
20  end
21
22  temperatura = Temperature_calculator.new(readline.chop)
23  temperatura.show
```

Javascript

```
<SCRIPT LANGUAGE="JavaScript"><!--
function demoMatchClick() {
  var re = new RegExp(document.demoMatch.regex.value);
  if (document.demoMatch.subject.value.match(re)) {
    alert("Successful match");
  } else {
    alert("No match");
  }
}

function demoShowMatchClick() {
  var re = new RegExp(document.demoMatch.regex.value);
  var m = re.exec(document.demoMatch.subject.value);
  if (m == null) {
    alert("No match");
  } else {
    var s = "Match at position " + m.index + ":\n";
    for (i = 0; i < m.length; i++) {
      s = s + m[i] + "\n";
    }
    alert(s);
  }
}

function demoReplaceClick() {
```

```
  var re = new RegExp(document.demoMatch.regex.value, "g");
  document.demoMatch.result.value =
    document.demoMatch.subject.value.replace(re,
      document.demoMatch.replacement.value);
}
// -->
</SCRIPT>

<FORM ID="demoMatch" NAME="demoMatch" METHOD=POST ACTION="javascript:void(0)">
<P>Regexp: <INPUT TYPE=TEXT NAME="regex" VALUE="\bt[a-z]+\b" SIZE=50></P>
<P>Subject string: <INPUT TYPE=TEXT NAME="subject"
   VALUE="This is a test of the JavaScript RegExp object" SIZE=50></P>
<P><INPUT TYPE=SUBMIT VALUE="Test Match" ONCLICK="demoMatchClick()">
<INPUT TYPE=SUBMIT VALUE="Show Match" ONCLICK="demoShowMatchClick()"></P>

<P>Replacement text: <INPUT TYPE=TEXT NAME="replacement" VALUE="replaced" SIZE=50></P>
<P>Result: <INPUT TYPE=TEXT NAME="result"
   VALUE="click the button to see the result" SIZE=50></P>
<P><INPUT TYPE=SUBMIT VALUE="Replace" ONCLICK="demoReplaceClick()"></P>
</FORM>
```

1.5. Casos de Estudio

1.5.1. Secuencias de números de tamaño fijo

El siguiente problema y sus soluciones se describen en el libro de J.E.F. Friedl [1]. Supongamos que tenemos un texto conteniendo códigos que son números de tamaño fijo, digamos seis dígitos, todos pegados, sin separadores entre ellos, como sigue:

012345678901**123334**234567890123**125934**890123345126

El problema es encontrar los códigos que comienzan por `12`. En negrita se han resaltado las soluciones. Son soluciones sólo aquellas que, comienzan por `12` en una posición múltiplo de seis. Una solución es:

```
@nums = grep {m/^12/} m/\d{6}/g;
```

que genera una lista con los números y luego selecciona los que comienzan por `12`. Otra solución es:

```
@nums = grep { defined } m/(12\d{4})|\d{6}/g;
```

que aprovecha que la expresión regular devolverá una lista vacía cuando el número no empieza por `12`:

```
DB<1> $x = '012345678901123334234567890123125934890123345126'
DB<2> x  ($x =~ m/(12\d{4})|\d{6}/g)
0  undef
1  undef
2  123334
3  undef
4  undef
5  125934
6  undef
7  undef
```

Obsérvese que se esta utilizando también que el operador `|` no es *greedy*.

¿Se puede resolver el problema usando sólamente una expresión regular? Obsérvese que esta solución "casi funciona":

```
 DB<3> x @nums = $x =~ m/(?:\d{6})*?(12\d{4})/g;
0  123334
1  125934
2  123345
```

recoge la secuencia mas corta de grupos de seis dígitos que no casan, seguida de una secuencia que casa. El problema que tiene esta solución es al final, cuando se han casado todas las soluciones, entonces la búsqueda exhaustiva hará que nos muestre soluciones que no comienzan en posiciones múltiplo de seis. Por eso encuentra `123345`:

012345678901**123334**234567890123**125934**890**123345**126

Por eso, Friedl propone esta solución:

```
@nums = m/(?:\d{6})*?(12\d{4})(?:(?!12)\d{6})*/g;
```

Se asume que existe al menos un éxito en la entrada inicial. Que es un extraordinario ejemplo de como el uso de paréntesis de agrupamiento simplifica y mejora la legibilidad de la solución. Es fantástico también el uso del operador de predicción negativo.

Solución usando el ancla \ `G`

El ancla `\G` ha sido concebida para su uso con la opción `/g`. Casa con el punto en la cadena en el que terminó el último emparejamiento. Cuando se trata del primer intento o no se está usando `/g`, usar `\G` es lo mismo que usar `\A`.

Mediante el uso de este ancla es posible formular la siguiente solución al problema planteado:

```
pl@nereida:~/Lperltesting$ perl -wde 0
main::(-e:1):   0
DB<1> $_ = '012345678901123334234567890123125934890123345126'
DB<2> x m/\G(?:\d{6})*?(12\d{4})/g
0  123334
1  125934
```

Sustitución

Si lo que se quiere es sustituir las secuencias deseadas es poisble hacerlo con la siguiente expresión regular:

```
casiano@nereida:~/docs/curriculums/CV_MEC$ perl -wde 0
DB<1> x $x = '012345678901123334234567890123125934890123345126'
0  012345678901123334234567890123125934890123345126
DB<2> x  ($y = $x) =~ s/(12\d{4})|\d{6}/$1? "-$1-":$& /ge
0  8
DB<3> p $y
012345678901-123334-234567890123-125934-890123345126
```

1.5.2. Palabras Repetidas

Su jefe le pide una herramienta que compruebe la aparición de duplicaciones consecutivas en un texto texto (como esta esta y la anterior anterior). La solución debe cumplir las siguientes especificaciones:

- Aceptar cualquier número de ficheros. Resaltar las apariciones de duplicaciones. Cada línea del informe debe estar precedida del nombre del fichero.

- Funcionar no sólo cuando la duplicación ocurre en la misma línea.
- Funcionar independientemente del *case* y de los blancos usados en medio de ambas palabras.
- Las palabras en cuestión pueden estar separadas por *tags* `HTML`.

```
#!/usr/bin/perl -w
use strict;
use Term::ANSIScreen qw/:constants/;

my $bold = BOLD();
my $clear = CLEAR();
my $line = 1;

# read paragraph
local $/ = ".\n";
while (my $par = <>) {
  next unless $par =~ s{
        \b                 # start word ...
        ([a-z]+)           # grab word in $1 and \1
        (                  # save the tags and spaces in $2
        (\s|<[^>]+>)+      # spaces or HTML tags
        )
        (\1\b)             # repeated word in $4
  }!$bold$1$clear$2$bold$4$clear!igx;

  $par =~ s/^/"$ARGV(".$line++."): "/meg;   # insert filename and line number

  print $par;
}
```

1.5.3. Análisis de cadenas con datos separados por comas

Supongamos que tenemos cierto texto en `$text` proveniente de un fichero `CSV` (*Comma Separated Values*). Esto es el fichero contiene líneas con el formato:

```
"earth",1,"moon",9.374
```

Esta línea representa cinco campos. Es razonable querer guardar esta información en un *array*, digamos `@field`, de manera que `$field[0] == 'earth'`, `$field[1] == '1'`, etc. Esto no sólo implica descomponer la cadena en campos sino también quitar las comillas de los campos entrecomillados. La primera solución que se nos ocurre es hacer uso de la función `split`:

```
@fields = split(/,/,$text);
```

Pero esta solución deja las comillas dobles en los campos entrecomillados. Peor aún, los campos entrecomillados pueden contener comas, en cuyo caso la división proporcionada por `split` sería errónea.

```
#!/usr/bin/perl -w
use Text::ParseWords;

sub parse_csv {
  my $text = shift;
  my @fields = (); # initialize @fields to be empty
```

```
 7
 8   while ($text =~
 9     m/"(([^"\\]|\\.)*)",? # quoted fields
10       |
11       ([^,]+),?           # $3 = non quoted fields
12       |
13       ,                   # allows empty fields
14     /gx
15     )
16   {
17     push(@fields, defined($1)? $1:$3); # add the just matched field
18   }
19   push(@fields, undef) if $text =~ m/,$/; #account for an empty last field
20   return @fields;
21 }
22
23 $test = '"earth",1,"a1, a2","moon",9.374';
24 print "string = \'$test\'\n";
25 print "Using parse_csv\n:";
26 @fields = parse_csv($test);
27 foreach $i (@fields) {
28   print "$i\n";
29 }
30
31 print "Using Text::ParseWords\n:";
32 #  @words = &quotewords($delim, $keep, @lines);
33 #The $keep argument is a boolean flag.  If true, then the
34 #tokens are split on the specified delimiter, but all other
35 #characters (quotes, backslashes, etc.) are kept in the
36 #tokens.  If $keep is false then the &*quotewords()
37 #functions remove all quotes and backslashes that are not
38 #themselves backslash-escaped or inside of single quotes
39 #(i.e., &quotewords() tries to interpret these characters
40 #just like the Bourne shell).
41
42 @fields = quotewords(',',0,$test);
43 foreach $i (@fields) {
44   print "$i\n";
45 }
```

Las subrutinas en Perl reciben sus argumentos en el *array* `@_`. Si la lista de argumentos contiene listas, estas son “aplanadas” en una única lista. Si, como es el caso, la subrutina ha sido declarada antes de la llamada, los argumentos pueden escribirse sin paréntesis que les rodeen:

```
@fields = parse_csv $test;
```

Otro modo de llamar una subrutina es usando el prefijo `&`, pero sin proporcionar lista de argumentos.

```
@fields = &parse_csv;
```

En este caso se le pasa a la rutina el valor actual del *array* `@_`.

Los operadores `push` (usado en la línea 17) y `pop` trabajan sobre el final del *array*. De manera análoga los operadores `shift` y `unshift` lo hacen sobre el comienzo. El operador ternario `?` trabaja de manera análoga como lo hace en C.

El código del `push` podría sustituirse por este otro:

```
push(@fields, $+);
```

Puesto que la variable `$+` contiene la cadena que ha casado con el último paréntesis que haya casado en el ultimo “matching”.

La segunda parte del código muestra que existe un módulo en Perl, el módulo `Text::Parsewords` que proporciona la rutina `quotewords` que hace la misma función que nuestra subrutina.

Sigue un ejemplo de ejecución:

```
> csv.pl
string = '"earth",1,"a1, a2","moon",9.374'
Using parse_csv
:earth
1
a1, a2
moon
9.374
Using Text::ParseWords
:earth
1
a1, a2
moon
9.374
```

1.5.4. Las Expresiones Regulares como Exploradores de un Árbol de Soluciones

Números Primos

El siguiente programa evalúa si un número es primo o no:

```
pl@nereida:~/Lperltesting$ cat -n isprime.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3
 4  my $num = shift;
 5  die "Usage: $0 integer\n" unless (defined($num) && $num =~ /^\d+$/);
 6
 7  if (("1" x $num) =~ /^(11+)\1+$/) {
 8    my $factor = length($1);
 9    print "$num is $factor x ".$num/$factor."\n";
10  }
11  else {
12    print "$num is prime\n";
13  }
```

Siguen varias ejecuciones:

```
pl@nereida:~/Lperltesting$ ./isprime.pl 35.32
Usage: ./isprime.pl integer
pl@nereida:~/Lperltesting$ ./isprime.pl 47
47 is prime
pl@nereida:~/Lperltesting$ ./isprime.pl 137
137 is prime
pl@nereida:~/Lperltesting$ ./isprime.pl 147
147 is 49 x 3
pl@nereida:~/Lperltesting$ ./isprime.pl 137
137 is prime
pl@nereida:~/Lperltesting$ ./isprime.pl 49
```

```
49 is 7 x 7
pl@nereida:~/Lperltesting$ ./isprime.pl 47
47 is prime
```

Ecuaciones Diofánticas: Una solución
Según dice la entrada Diophantine_equationen la wikipedia:

> *In mathematics, a Diophantine equation is an indeterminate polynomial equation that allows the variables to be integers only.*

La siguiente sesión con el depurador muestra como se puede resolver una ecuación lineal diofántica con coeficientes positivos usando una expresión regular:

```
  DB<1> # Resolvamos 3x + 2y + 5z = 40
DB<2> x ('a'x40) =~  /^((?:...)+)((?:..)+)((?:.....)+)$/
0  'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
1  'aa'
2  'aaaaa'
DB<3> x map { length }  ('a'x40) =~  /^((?:...)+)((?:..)+)((?:.....)+)$/
0  33
1  2
2  5
DB<4> @c = (3, 2, 5)
DB<5> x map { length($_) / $c[$i++] }  ('a'x40) =~  /^((?:...)+)((?:..)+)((?:.....)+)$/
0  11
1  1
2  1
DB<6> p 3*11+2*1+5*1
40
```

Ecuaciones Diofánticas: Todas las soluciones
Usando el verbo `(*FAIL)` es posible obtener todas las soluciones:

```
main::(-e:1):   0
DB<1>  sub equ { my @c = @_; print "\t3*$c[0]+2*$c[1]+5*$c[2] = ",3*$c[0]+2*$c[1]+5*$c[2],"\n"
DB<2> sub f { my @c = ((length($1)/3), (length($2)/2), (length($3)/5)); equ(@c); }
DB<3> x ('a'x40) =~  /^((?:...)+)((?:..)+)((?:.....)+)$(?{ f() })(*FAIL)/x
        3*11+2*1+5*1 = 40
        3*9+2*4+5*1 = 40
        3*8+2*3+5*2 = 40
        3*7+2*7+5*1 = 40
        3*7+2*2+5*3 = 40
        3*6+2*6+5*2 = 40
        3*6+2*1+5*4 = 40
        3*5+2*10+5*1 = 40
        3*5+2*5+5*3 = 40
        3*4+2*9+5*2 = 40
        3*4+2*4+5*4 = 40
        3*3+2*13+5*1 = 40
        3*3+2*8+5*3 = 40
        3*3+2*3+5*5 = 40
        3*2+2*12+5*2 = 40
        3*2+2*7+5*4 = 40
        3*2+2*2+5*6 = 40
```

```
            3*1+2*16+5*1 = 40
            3*1+2*11+5*3 = 40
            3*1+2*6+5*5 = 40
            3*1+2*1+5*7 = 40
  empty array
DB<4>
```

Ecuaciones Diofánticas: Resolutor general

El siguiente programa recibe en línea de comandos los coeficientes y término inependeinte de una ecuación lineal diofántica con coeficientes positivos y muestra todas las soluciones. El algoritmo primero crea una cadena conteniendo el código Perl que contiene la expresión regular adecuada para pasar luego a evaluarlo:

```
pl@nereida:~/Lperltesting$ cat -n diophantinesolvergen.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1 -w
use v5.10;
use strict;

# Writes a Perl solver for
# a1 x1 + a2 x2 + ... + an xn = b
# a_i and b integers > 0
#

my $b = pop;
my @a = @ARGV;
my $debug = 1;

my $b1 = '1'x$b;
my @a1 = map { '1'x$_ } @a;
my @index = map { 'length($'.$_.")/".$a[$_-1] } 1..(@a);
my $aux = join ",", @index;

my $regexp = '^';
$regexp .= "((?:$_)+)" for @a1;

$regexp .= '$(?{ f() })(*FAIL)';

my $solver = <<"SOLVER";
my \@stack;
sub f {
  my \@s = ($aux);
  push \@stack, [ \@s ];
}

q{$b1} =~ m{$regexp}x;

return \@stack;
SOLVER

print "Solver:\n--------\n$solver\n--------\n" if $debug;

my @stack = eval $solver;

say("@$_") for @stack
```

Sigue un ejemplo de ejecución:

```
pl@nereida:~/Lperltesting$ ./diophantinesolvergen.pl 3 2 5 40
Solver:
--------
my @stack;
sub f {
  my @s = (length($1)/3,length($2)/2,length($3)/5);
  push @stack, [ @s ];
}

q{1111111111111111111111111111111111111111} =~ m{^((?:111)+)((?:11)+)((?:11111)+)$(?{ f() })(*

return @stack;

--------
11 1 1
9 4 1
8 3 2
7 7 1
7 2 3
6 6 2
6 1 4
5 10 1
5 5 3
4 9 2
4 4 4
3 13 1
3 8 3
3 3 5
2 12 2
2 7 4
2 2 6
1 16 1
1 11 3
1 6 5
1 1 7
```

Las Tres Hijas

En la páginas de Retos Matemáticos de DIVULGAMAT puede encontrarse el siguiente problema:

Ejercicio 1.5.1. *Dos matemáticos se vieron en la calle después de muchos años sin coincidir.*

- *¡Hola!, ¿qué tal?, ¿te casaste?, y... ¿cuántos hijos tienes?*
- *Pues tengo tres hijas.*
- *¿y qué años tienen?*
- *¡A ver si lo adivinas!: el producto de las edades de las tres es 36, y su suma es el número del portal que ves enfrente...*
- *¡Me falta un dato!*

- *¡Ah, sí!, ¡la mayor toca el piano!*

¿Qué edad tendrán las tres hijas?

¿Podemos ayudarnos de una expresión regular para resolver el problema? Al ejecutar el siguiente programa:

```
pl@nereida:~/Lperltesting$ cat -n playspiano.pl
 1  #!/usr/local/lib/perl/5.10.1/bin//perl5.10.1  -w
 2  use v5.10;
 3  use strict;
 4  use List::Util qw{sum};
 5
 6  local our %u;
 7  sub f {
 8    my @a = @_;
 9    @a = sort { $b <=> $a } (length($a[1]), length($a[0])/length($a[1]), 36/length($a[0]) );
10
11    local $" = ", ";
12    say "(@a)\t ".sum(@a) unless exists($u{"@a"});
13    $u{"@a"} = undef;
14  }
15
16  say "SOL\t\tNUMBER";
17  my @a =  (’1’x36) =~
18           /^((1+)\2+)(\1+)$
19                      (?{ f($1, $2, $3)
20                       })
21             (*FAIL)
22           /x;
```

obtenemos la salida:

```
pl@nereida:~/Lperltesting$ ./playspiano.pl
SOL             NUMBER
(9, 2, 2)        13
(6, 3, 2)        11
(4, 3, 3)        10
(18, 2, 1)       21
(12, 3, 1)       16
(9, 4, 1)        14
(6, 6, 1)        13
```

Explique el funcionamiento del programa. A la vista de la salida ¿Cuáles eran las edades de las hijas?

Mochila 0-1

Para una definición del problema vea la sección El Problema de la Mochila 0-1 en los apuntes de LHP

Ejercicio 1.5.2. *¿Sería capaz de resolver usando expresiones regulares el problema de la mochila 0-1? ¡Si lo logra merece el premio a la solución mas freak que se haya encontrado para dicho problema!*

Véase también

Véase también:

- Véase el nodo en PerlMonks *The Oldest Plays the Piano*
- Solving Algebraic Equations Using Regular Expressions

1.5.5. Número de substituciones realizadas

El operador de substitución devuelve el número de substituciones realizadas, que puede ser mayor que uno si se usa la opción `/g`. En cualquier otro caso retorna el valor falso.

```
#!/usr/bin/perl -w
undef($/);
$paragraph = <STDIN>;
$count = 0;
$count = ($paragraph =˜ s/Mister\b/Mr./ig);
print "$paragraph";
print "\n$count\n";
```

El resultado de la ejecución es el siguiente:

```
> numsust.pl
Dear Mister Bean,
Is a pleasure for me and Mister Pluto
to invite you to the Opening Session
Official dinner that will be chaired by
Mister Goofy.

Yours sincerely
  Mister Mickey Mouse
Dear Mr. Bean,
Is a pleasure for me and Mr. Pluto
to invite you to the Opening Session
Official dinner that will be chaired by
Mr. Goofy.

Yours sincerely
  Mr. Mickey Mouse

4
```

1.5.6. Expandiendo y comprimiendo tabs

Este programa convierte los tabs en el número apropiado de blancos.

```
pl@nereida:˜/Lperltesting$ cat -n expandtabs.pl
   1  #!/usr/bin/perl -w
   2  use strict;
   3
   4  my @string = <>;
   5
   6  for (@string) {
   7    while (s/\t+/’ ’ x (length($&)*8 - length($‘)%8)/e) {}
   8    print $_;
   9  }
```

Sigue un ejemplo de ejecución:

```
pl@nereida:˜/Lperltesting$ cat -nt tabs.in
     1  012345670123456701234567012345670
     2  one^Itwo^I^Ithree
     3  four^I^I^I^Ifive
     4  ^I^Itwo
```

```
pl@nereida:~/Lperltesting$ ./expandtabs.pl tabs.in | cat -tn
     1  012345670123456701234567012345670
     2  one     two             three
     3  four                            five
     4                  two
```

Ejercicio 1.5.3. *¿Funciona igual si se cambia el bucle* `while` *por una opción* `/g`*?*

```
pl@nereida:~/Lperltesting$ cat -n ./expandtabs2.pl
   1    #!/usr/bin/perl -w
   2    use strict;
   3
   4    my @string = <>;
   5
   6    for (@string) {
   7      s/\t+/’ ’ x (length($&)*8 - length($‘)%8)/ge;
   8      print $_;
   9    }
```

¿Porqué?

1.5.7. Modificación de Múltiples Ficheros: one liner

Aunque no es la forma de uso habitual, Perl puede ser utilizado en “modo sed” para modificar el texto en múltiples ficheros:

`perl -e ’s/nereida\.deioc\.ull\.es/miranda.deioc.ull.es/gi’ -p -i.bak *.html`

Este programa sustituye la palabra original (g)lobalmente e i)gnorando el “case”) en todos los ficheros `*.html` y para cada uno de ellos crea una copia de seguridad `*.html.bak`.

Otro ejemplo: la sustitución que sigue ocurre en todos los ficheros `info.txt` en todos los subdirectorios de los subdirectorios que comiencen por `alu`:

```
perl -e ’s/\|hyperpage//gi’ -p -i.bak  alu*/*/info.txt
```

Las *opciones de línea* de comandos significan lo siguiente:

-e puede usarse para definir el script en la línea de comandos. Multiples -e te permiten escribir un multi-script. Cuando se usa -e, perl no busca por un fichero de script entre la lista de argumentos.

-p La opción `-p` hace que perl incluya un bucle alrededor de tu “script” al estilo sed:

```
while (<>) {
        ...             # your script goes here
} continue {
        print;
}
```

-n Nótese que las líneas se imprimen automáticamente. Para suprimir la impresión usa la opción `-n`

-i[ext] La opción `-i` Expresa que los ficheros procesados serán modificados. Se renombra el fichero de entrada `file.in` a `file.in.ext`, abriendo el de salida con el mismo nombre del fichero de entrada `file.in`. Se selecciona dicho fichero como de salida por defecto para las sentencias `print`. Si se proporciona una extensión se hace una copia de seguridad. Si no, no se hace copia de seguridad.

En general las opciones pueden ponerse en la primera línea del "script", donde se indica el intérprete. Asi pues, decir

`perl -p -i.bak -e "s/foo/bar/;"`

es equivalente a usar el "script":

```
#!/usr/bin/perl -pi.bak
s/foo/bar/;
```

1.6. tr y split

El operador de traducción permite la conversión de unos caracteres por otros. Tiene la sintáxis:

`tr/SEARCHLIST/REPLACEMENTLIST/cds`
`y/SEARCHLIST/REPLACEMENTLIST/cds`

El operador permite el reemplazo carácter a carácter, por ejemplo:

```
$ perl -de 0
  DB<1> $a = 'fiboncacci'
  DB<2> $a =~ tr/aeiou/AEIOU/
  DB<3> print $a
fIbOncAccI
  DB<4> $a =~ y/fbnc/FBNC/
  DB<5> print $a
FIBONCACCI
```

El operador devuelve el número de carácteres reeemplazados o suprimidos.

```
$cnt = $sky =~ tr/*/*/; # count the stars in $sky
```

Si se especifica el modificador `/d`, cualquier carácter en `SEARCHLIST` que no figure en `REPLACEMENTLIST` es eliminado.

```
  DB<6> print $a
FIBONCACCI
  DB<7> $a =~ y/OA//d
  DB<8> print $a
FIBNCCCI
```

Si se especifica el modificador `/s`, las secuencias de carácteres consecutivos que serían traducidas al mismo carácter son comprimidas a una sola:

```
  DB<1> $b = 'aaghhh!'
  DB<2> $b =~ tr/ah//s
  DB<3> p $b
agh!
```

Observa que si la cadena `REPLACEMENTLIST` es vacía, no se introduce ninguna modificación.

Si se especifica el modificador `/c`, se complementa `SEARCHLIST`; esto es, se buscan los caracteres que no están en `SEARCHLIST`.

```
tr/a-zA-Z/ /cs; # change non-alphas to single space
```

Cuando se dan múltiples traducciones para un mismo carácter, solo la primera es utilizada:

```
tr/AAA/XYZ/
```

traducirá `A` por `X`.

El siguiente *script* busca una expresión regular en el fichero de `passwords` e imprime los *login* de los usuarios que casan con dicha cadena. Para evitar posibles confusiones con las vocales acentuadas se usa el operador `tr`.

```
#!/usr/bin/perl -w
$search = shift(@ARGV) or die("you must provide a regexpr\n");
$search =~ y/ÁÉÍÓÚáéíóú/AEIOUaeiou/;
open(FILE,"/etc/passwd");
while ($line = <FILE>) {
  $line =~ y/ÁÉÍÓÚáéíóú/AEIOUaeiou/;
  if ($line =~ /$search/io) {
    @fields = split(":",$line);
    $login = $fields[0];
    if ($line !~ /^#/) {
      print "$login\n";
    }
    else {
      print "#$login\n";
    }
  }
}
```

Ejecución (suponemos que el nombre del fichero anterior es `split.pl`):

```
> split.pl Rodriguez
##direccion
call
casiano
alu5
alu6
##doctorado
paco
falmeida
##ihiu07
```

Para familiarizarte con este operador, codifica y prueba el siguiente código:

```
 1 #!/usr/bin/perl -w
 2 $searchlist = shift @ARGV;
 3 $replacelist = shift @ARGV;
 4 $option = "";
 5 $option = shift @ARGV if @ARGV;
 6
 7 while (<>) {
 8   $num = eval "tr/$searchlist/$replacelist/$option";
 9   die $@ if $@;
10   print "$num: $_";
11 }
```

Perl construye la tabla de traducción en “tiempo de compilación”. Por ello ni `SEARCHLIST` ni `REPLACEMENTLIST` son susceptibles de ser interpolados. Esto significa que si queremos usar variables tenemos que recurrir a la función `eval`.

La expresión pasada como parámetro a `eval` en la línea 8 es analizada y ejecutada como si se tratara de un pequeño programa Perl. Cualquier asignación a variables permanece después del `eval`, asi como cualquier definición de subrutina. El código dentro de `eval` se trata como si fuera un bloque, de manera que cualesquiera variables locales (declaradas con `my`) desaparecen al final del bloque.

La variable `$@` contiene el mensaje de error asociado con la última ejecución del comando `eval`. Si es nula es que el último comando se ejecuto correctamente. Aqui tienes un ejemplo de llamada:

```
> tr.pl 'a-z' 'A-Z' s
jose hernandez
13: JOSE HERNANDEZ
joosee hernnandez
16: JOSE HERNANDEZ
```

1.7. Pack y Unpack

El operador `pack` trabaja de forma parecida a `sprintf`. Su primer argumento es una cadena, seguida de una lista de valores a formatear y devuelve una cadena:

```
pack("CCC", 65, 66, 67, 68) # empaquetamos A B C D
```

el inverso es el operador `unpack`

```
unpack("CCC", "ABCD")
```

La cadena de formato es una lista de especificadores que indican el tipo del dato que se va a empaquetar/desempaquetar. Cada especificador puede opcionalmente seguirse de un contador de repetición que indica el número de elementos a formatear. Si se pone un asterisco (`*`) se indica que la especificación se aplica a todos los elementos restantes de la lista.

Formato	Descripción
A	Una cadena completada con blancos
a	Una cadena completada con ceros
B	Una cadena binaria en orden descendente
b	Una cadena binaria en orden ascendente
H	Una cadena hexadecimal, los nibble altos primero
h	Una cadena hexadecimal, los nibble bajos primero

Ejemplo de uso del formato `A`:

```
DB<1> $a = pack "A2A3", "Pea","rl"
DB<2> p $a
Perl
DB<3> @b = unpack "A2A3", "Perl"
DB<4> p "@b"
Pe rl
```

La variable `@b` tiene ahora dos cadenas. Una es `Pe` la otra es `rl`. Veamos un ejemplo con el formato `B`:

```
p ord('A')
65
DB<22> $x = pack "B8", "01000001"
DB<23> p $x
A
DB<24> @y = unpack "B8", "A"
DB<25> p "@y"
01000001
DB<26> $x = pack "b8", "10000010"
DB<27> p $x
```

1.8. Práctica: Un lenguaje para Componer Invitaciones

En el capítulo 6 (sección 6.4.2.2) del libro The LaTex Web Companion se define un lenguaje para componer textos para enviar invitaciones.

Para escribir una invitación en ese lenguaje escribiríamos algo así:

```
pl@nereida:~/Lpl0910/Practicas/161009/src$ cat -n invitation.xml
 1  <?xml version="1.0"?>
 2  <!DOCTYPE invitation SYSTEM "invitation.dtd">
 3  <invitation>
 4  <!-- ++++ The header part of the document ++++ -->
 5  <front>
 6  <to>Anna, Bernard, Didier, Johanna</to>
 7  <date>Next Friday Evening at 8 pm</date>
 8  <where>The Web Cafe</where>
 9  <why>My first XML baby</why>
10  </front>
11  <!-- +++++ The main part of the document +++++ -->
12  <body>
13  <par>
14  I would like to invite you all to celebrate
15  the birth of <emph>Invitation</emph>, my
16  first XML document child.
17  </par>
18  <par>
19  Please do your best to come and join me next Friday
20  evening. And, do not forget to bring your friends.
21  </par>
22  <par>
23  I <emph>really</emph> look forward to see you soon!
24  </par>
25  </body>
26  <!-- +++ The closing part of the document ++++ -->
27  <back>
28  <signature>Michel</signature>
29  </back>
30  </invitation>
```

La sintáxis del lenguaje queda reflejada en la siguiente *Document Type Definition* (*DTD*) que aparece en la sección 6.4.3 del libro de Goosens:

```
pl@nereida:~/Lpl0910/Practicas/161009/src$ cat -n invitation.dtd
 1  <!-- invitation DTD   -->
 2  <!-- May 26th 1998 mg -->
 3  <!ELEMENT invitation (front, body, back) >
 4  <!ELEMENT front      (to, date, where, why?) >
 5  <!ELEMENT date       (#PCDATA) >
 6  <!ELEMENT to         (#PCDATA) >
 7  <!ELEMENT where      (#PCDATA) >
 8  <!ELEMENT why        (#PCDATA) >
 9  <!ELEMENT body       (par+) >
10  <!ELEMENT par        (#PCDATA|emph)* >
11  <!ELEMENT emph       (#PCDATA) >
12  <!ELEMENT back       (signature) >
13  <!ELEMENT signature  (#PCDATA) >
```

El objetivo de esta práctica es escribir un programa Perl que usando las extensiones para expresiones regulares presentes en la versión 5.10 reconozca el lenguaje anterior.

Véase también:

- The LaTex Web Companion

- Examples from The LaTeX Web Companion (véanse los subdirectorios correspondietnes a los capítulos 6 y 7)

1.9. Analisis Sintáctico con Expresiones Regulares Perl

1.9.1. Introducción al Analísis Sintáctico con Expresiones Regulares

Como se ha comentado en la sección 1.2.5 Perl 5.10 permite el reconocimiento de expresiones definidas mediante gramáticas recursivas, siempre que estas puedan ser analizadas por un analizador recursivo descendente. Sin embargo, las expresiones regulares Perl 5.10 hace difícil construir una representación del árbol de análisis sintáctico abstracto. Además, la necesidad de explicitar en la regexp los blancos existentes entre los símbolos hace que la descripción sea menos robusta y menos legible.

Ejemplo: Traducción de expresiones aritméticas en infijo a postfijo

El siguiente ejemplo muestra una expresión regular que traduce expresiones de diferencias en infijo a postfijo.

Se usa una variable `$tran` para calcular la traducción de la subexpresión vista hasta el momento.

La gramática original que consideramos es recursiva a izquierdas:

```
 exp ->   exp ’-’ digits
        | digits
```

aplicando las técnicas explicadas en 2.8.1 y en el nodo de perlmonks 553889 transformamos la gramática en:

```
exp ->   digits rest
rest ->   ’-’ rest
        | # empty
```

Sigue el código:

```
pl@nereida:~/Lperltesting$ cat -n infixtopostfix.pl
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1
use v5.10;

# Infix to postfix translator using 5.10 regexp
# original grammar:
#  exp ->   exp ’-’ digits
#         | digits
#
# Applying left-recursion elimination we have:
#  exp ->   digits rest
#  rest ->   ’-’ rest
#          | # empty
#
my $input;
local our $tran = ’’;

my $regexp = qr{
    (?&exp)

    (?(DEFINE)
        (?<exp>    ((?&digits)) \s* (?{ $tran .= "$^N "; say "tran=$tran"; }) (?&rest)
                      (?{
                         say "exp -> digits($^N) rest";
```

```
                      })
        )

        (?<rest>     \s* - ((?&digits)) (?{ $tran .= "$^N - "; say "tran=$tran"; }) (?&
                        (?{
                           say "rest -> - digits($^N) rest";
                        })
                  |  # empty
                        (?{
                           say "rest -> empty";
                        })
        )

        (?<digits>  \s* (\d+)
        )
    )
}xms;

$input = <>;
chomp($input);
if ($input =~ $regexp) {
  say "matches: $&\ntran=$tran";
}
else {
  say "does not match";
}
```

La variable `$^N` contiene el valor que casó con el último paréntesis. Al ejecutar el código anterior obtenemos:

Véase la ejecución:

```
pl@nereida:~/Lperltesting$ ./infixtopostfix.pl
ab 5 - 3 -2 cd;
tran= 5
tran= 5  3 -
tran= 5  3 - 2 -
rest -> empty
rest -> - digits(2) rest
rest -> - digits( 3) rest
exp -> digits( 5) rest
matches:  5 - 3 -2
tran= 5  3 - 2 -
```

Como se ve, el recorrido primero profundo se traduce en la reconstrucción de una derivación a derechas.

Accediendo a los atributos de paréntesis anteriores mediante acciones intermedias

Es difícil extender el ejemplo anterior a lenguajes mas complejos debido a la limitación de que sólo se dispone de acceso al último paréntesis vía `$^N`. En muchos casos es necesario poder acceder a paréntesis/atributos anteriores.

El siguiente código considera el caso de expresiones con sumas, restas, multiplicaciones y divisiones. Utiliza la variable `op` y una acción intermedia (líneas 51-53) para almacenar el segundo paréntesis necesitado:

```
pl@nereida:~/Lperltesting$ cat -n ./calc510withactions3.pl
```

```
#!/usr/local/lib/perl/5.10.1/bin//perl5.10.1
use v5.10;

# Infix to postfix translator using 5.10 regexp
# Original grammar:

#  exp ->   exp [-+] term
#         | term
#  term ->   term [*/] digits
#          | digits

# Applying left-recursion elimination we have:

#  exp  ->   term re
#  re   ->   [+-] term re
#          | # empty
#  term ->   digits rt
#  rt   ->   [*/] rt
#          | # empty


my $input;
my @stack;

local our $op = ’’;
my $regexp = qr{
    (?&exp)

    (?(DEFINE)
        (?<exp>    (?&term) (?&re)
                     (?{ say "exp -> term re" })
        )

        (?<re>     \s* ([+-]) (?&term) \s* (?{ push @stack, $^N }) (?&re)
                     (?{ say "re -> [+-] term re" })
                 | # empty
                     (?{ say "re -> empty" })
        )

        (?<term>   ((?&digits))
                      (?{ # intermediate action
                          push @stack, $^N
                      })
                   (?&rt)
                      (?{
                          say "term-> digits($^N) rt";
                      })
        )

        (?<rt>     \s*([*/])
                           (?{ # intermediate action
                              local $op = $^N;
                           })
```

```
                   ((?&digits)) \s*
                           (?{  # intermediate action
                                push @stack, $^N, $op
                            })
                   (?&rt) # end of <rt> definition
                           (?{
                                say "rt -> [*/] digits($^N) rt"
                            })
                  |  # empty
                     (?{ say "rt -> empty" })
        )

        (?<digits>  \s* \d+
        )
    )
}xms;

$input = <>;
chomp($input);
if ($input =~ $regexp) {
  say "matches: $&\nStack=(@stack)";
}
else {
  say "does not match";
}
```

Sigue una ejecución:

```
pl@nereida:~/Lperltesting$ ./calc510withactions3.pl
5-8/4/2-1
rt -> empty
term-> digits(5) rt
rt -> empty
rt -> [*/] digits(2) rt
rt -> [*/] digits(4) rt
term-> digits(8) rt
rt -> empty
term-> digits(1) rt
re -> empty
re -> [+-] term re
re -> [+-] term re
exp -> term re
matches: 5-8/4/2-1
Stack=(5 8 4 / 2 / - 1 -)
```

Accediendo a los atributos de paréntesis anteriores mediante `@-`

Sigue una solución alternativa que obvia la necesidad de introducir incómodas acciones intermedias. Utilizamos las variables `@-` y `@+`:

> *Since Perl 5.6.1 the special variables* `@-` *and* `@+` *can functionally replace* `` $` ``, `$&` *and* `$'`. *These arrays contain pointers to the beginning and end of each match (see* `perlvar` *for the full story), so they give you essentially the same information, but without the risk of excessive string copying.*

Véanse los párrafos en las páginas 44, 44) y 45 para mas información sobre `@-` y `@+`.

Nótese la función `rc` en las líneas 21-28. `rc(1)` nos retorna lo que casó con el último paréntesis, `rc(2)` lo que casó con el penúltimo, etc.

```
pl@nereida:~/Lperltesting$ cat -n calc510withactions4.pl
 1    #!/usr/local/lib/perl/5.10.1/bin//perl5.10.1
 2    use v5.10;
 3
 4    # Infix to postfix translator using 5.10 regexp
 5    # Original grammar:
 6
 7    #  exp ->   exp [-+] term
 8    #         | term
 9    #  term ->   term [*/] digits
10    #          | digits
11
12    # Applying left-recursion elimination we have:
13
14    #  exp  ->   term re
15    #  re   ->   [+-] term re
16    #          | # empty
17    #  term ->   digits rt
18    #  rt   ->   [*/] rt
19    #          | # empty
20
21    sub rc {
22      my $ofs = - shift;
23
24      # Number of parenthesis that matched
25      my $np = @-;
26      #      string, ofsset, length
27      substr($_, $-[$ofs], $+[$np+$ofs] - $-[$ofs])
28    }
29
30    my $input;
31    my @stack;
32
33    my $regexp = qr{
34        (?&exp)
35
36        (?(DEFINE)
37            (?<exp>    (?&term) (?&re)
38                         (?{ say "exp -> term re" })
39            )
40
41            (?<re>     \s* ([+-]) (?&term) \s* (?{ push @stack, rc(1) }) (?&re)
42                         (?{ say "re -> [+-] term re" })
43                     | # empty
44                         (?{ say "re -> empty" })
45            )
46
47            (?<term>   ((?&digits))
48                          (?{ # intermediate action
49                              push @stack, rc(1)
50                          })
```

```
                   (?&rt)
                      (?{
                          say "term-> digits(".rc(1).") rt";
                      })
    )

        (?<rt>     \s*([*/]) ((?&digits)) \s*
                           (?{  # intermediate action
                                push @stack, rc(1), rc(2)
                            })
                   (?&rt) # end of <rt> definition
                           (?{
                                say "rt -> [*/] digits(".rc(1).") rt"
                            })
                  |  # empty
                     (?{ say "rt -> empty" })
        )

        (?<digits>  \s* \d+
        )
    )
}xms;

$input = <>;
chomp($input);
if ($input =~ $regexp) {
  say "matches: $&\nStack=(@stack)";
}
else {
  say "does not match";
}
```

Ahora accedemos a los atributos asociados con los dos paréntesis, en la regla de `<rt>` usando la función `rc`:

```
(?<rt>     \s*([*/]) ((?&digits)) \s*
                   (?{  # intermediate action
                        push @stack, rc(1), rc(2)
                    })
```

Sigue una ejecución del programa:

```
pl@nereida:~/Lperltesting$ ./calc510withactions4.pl
5-8/4/2-1
rt -> empty
term-> digits(5) rt
rt -> empty
rt -> [*/] digits(2) rt
rt -> [*/] digits(4) rt
term-> digits(8) rt
rt -> empty
term-> digits(1) rt
re -> empty
re -> [+-] term re
```

```
re -> [+-] term re
exp -> term re
matches: 5-8/4/2-1
Stack=(5 8 4 / 2 / - 1 -)
pl@nereida:~/Lperltesting$
```

Accediendo a los atributos de paréntesis anteriores mediante paréntesis con nombre

Una nueva solución: dar nombre a los paréntesis y acceder a los mismos:

```
(?<rt>     \s*(?<op>[*/]) (?<num>(?&digits)) \s*
                   (?{  # intermediate action
                        push @stack, $+{num}, $+{op}
                    })
```

Sigue el código completo:

```
pl@nereida:~/Lperltesting$ cat -n ./calc510withnamedpar.pl
 1    #!/usr/local/lib/perl/5.10.1/bin//perl5.10.1
 2    use v5.10;
 3
 4    # Infix to postfix translator using 5.10 regexp
 5    # Original grammar:
 6
 7    #  exp ->   exp [-+] term
 8    #        | term
 9    #  term ->   term [*/] digits
10    #         | digits
11
12    # Applying left-recursion elimination we have:
13
14    #  exp  ->   term re
15    #  re   ->   [+-] term re
16    #         | # empty
17    #  term ->   digits rt
18    #  rt   ->   [*/] rt
19    #         | # empty
20
21    my @stack;
22
23    my $regexp = qr{
24        (?&exp)
25
26        (?(DEFINE)
27            (?<exp>    (?&term) (?&re)
28                         (?{ say "exp -> term re" })
29            )
30
31            (?<re>     \s* ([+-]) (?&term) \s* (?{ push @stack, $^N }) (?&re)
32                         (?{ say "re -> [+-] term re" })
33                     | # empty
34                         (?{ say "re -> empty" })
35            )
36
37            (?<term>   ((?&digits))
```

```
                      (?{ # intermediate action
                          push @stack, $^N
                      })
                   (?&rt)
                      (?{
                          say "term-> digits($^N) rt";
                      })
        )

        (?<rt>     \s*(?<op>[*/]) (?<num>(?&digits)) \s*
                           (?{  # intermediate action
                                push @stack, $+{num}, $+{op}
                            })
                   (?&rt) # end of <rt> definition
                           (?{
                                say "rt -> [*/] digits($^N) rt"
                            })
                  |  # empty
                     (?{ say "rt -> empty" })
        )

        (?<digits>  \s* \d+
        )
    )
}xms;

my $input = <>;
chomp($input);
if ($input =~ $regexp) {
  say "matches: $&\nStack=(@stack)";
}
else {
  say "does not match";
}
```

Ejecución:

```
pl@nereida:~/Lperltesting$ ./calc510withnamedpar.pl
5-8/4/2-1
rt -> empty
term-> digits(5) rt
rt -> empty
rt -> [*/] digits(2) rt
rt -> [*/] digits(4) rt
term-> digits(8) rt
rt -> empty
term-> digits(1) rt
re -> empty
re -> [+-] term re
re -> [+-] term re
exp -> term re
matches: 5-8/4/2-1
Stack=(5 8 4 / 2 / - 1 -)
```

Véase También

- El nodo *Backreference variables in code embedded inside Perl 5.10 regexps* en PerlMonks
- El nodo *Strange behavior of @- and @+ in perl5.10 regexps* en PerlMonks

1.9.2. Construyendo el AST con Expresiones Regulares 5.10

Construiremos en esta sección un traductor de infijo a postfijo utilizando una aproximación general: construiremos una representación del Abstract Syntax Tree o AST (véase la sección 2.9 Árbol de Análisis Abstracto para una definición detallada de que es un árbol sintáctico).

Como la aplicación es un poco mas compleja la hemos dividido en varios ficheros. Esta es la estructura:

```
.
|-- ASTandtrans3.pl    # programa principal
|-- BinaryOp.pm        # clases para el manejo de los nodos del AST
|-- testreegxpparen.pl # prueba para Regexp::Paren
‘-- Regexp
    ‘-- Paren.pm       # módulo de extensión de $^N
```

La salida del programa puede ser dividida en tres partes. La primera muestra una antiderivación a derechas inversa:

```
pl@nereida:~/Lperltesting$ ./ASTandtrans3.pl
2*(3-4)
factor -> NUM(2)
factor -> NUM(3)
rt -> empty
term-> factor rt
factor -> NUM(4)
rt -> empty
term-> factor rt
re -> empty
re -> [+-] term re
exp -> term re
factor -> ( exp )
rt -> empty
rt -> [*/] factor rt
term-> factor rt
re -> empty
exp -> term re
matches: 2*(3-4)
```

Que leída de abajo a arriba nos da una derivación a derechas de la cadena `2*(3-4)`:

```
exp => term re => term => factor rt =>
factor [*/](*) factor rt => factor [*/](*) factor =>
factor [*/](*) ( exp ) => factor [*/](*) ( term re ) =>
factor [*/](*) ( term [+-](-) term re ) =>
factor [*/](*) ( term [+-](-) term ) =>
factor [*/](*) ( term [+-](-) factor rt ) =>
factor [*/](*) ( term [+-](-) factor ) =>
factor [*/](*) ( term [+-](-) NUM(4) ) =>
factor [*/](*) ( factor rt [+-](-) NUM(4) ) =>
factor [*/](*) ( factor [+-](-) NUM(4) ) =>
factor [*/](*) ( NUM(3) [+-](-) NUM(4) )  =>
NUM(2) [*/](*) ( NUM(3) [+-](-) NUM(4) )
```

La segunda parte nos muestra la representación del AST para la entrada dada (`2*(3-4)`):

```
AST:
$VAR1 = bless( {
  ’left’ => bless( {
  ’right’ => bless(
    ’left’ => bless(
    ’right’ => bless
    ’op’ => ’-’
  }, ’ADD’ ),
  ’op’ => ’*’
}, ’MULT’ );
```

La última parte de la salida nos muestra la traducción a postfijo de la expresión en infijo suministrada en la entrada (`2*(3-4)`):

```
2 3 4 - *
```

Programa Principal: usando la pila de atributos

La gramática original que consideramos es recursiva a izquierdas:

```
exp    ->   exp [-+] term
          | term
term   ->   term [*/] factor
          | factor
factor ->  \( exp \)
          | \d+
```

aplicando las técnicas explicadas en 2.8.2 es posible transformar la gramática en una no recursiva por la izquierda:

```
exp       ->   term restoexp
restoexp  ->   [-+] term restoexp
             | # vacío
term      ->   term restoterm
restoterm ->   [*/] factor restoterm
             | # vacío
factor    ->   \( exp \)
             | \d+
```

Ahora bien, no basta con transformar la gramática en una equivalente. Lo que tenemos como punto de partida no es una gramática sino un *esquema de traducción* (véase la sección 2.7) que construye el AST asociado con la expresión. Nuestro esquema de traducción conceptual es algo así:

```
exp    ->   exp ([-+]) term       { ADD->new(left => $exp, right => $term, op => $1) }
          | term                  { $term }
term   ->   term ([*/]) factor    { MULT->new(left => $exp, right => $term, op => $1) }
          | factor                { $factor }
factor ->  \( exp \)              { $exp }
          | (\d+)                 { NUM->new(val => $1) }
```

Lo que queremos conseguir un conjunto de acciones semánticas asociadas para gramática no recursiva que sea equivalente a este.

Este es el programa resultante una vez aplicadas las transformaciones. La implementación de la asociación entre símbolos y atributos la realizamos manualmente mediante una pila de atributos:

```
pl@nereida:~/Lperltesting$ cat -n ./ASTandtrans3.pl
 1  #!/usr/local/lib/perl/5.10.1/bin//perl5.10.1
 2  use v5.10;
 3  use strict;
 4  use Regexp::Paren qw{g};
 5  use BinaryOp;
 6
 7  use Data::Dumper;
 8  $Data::Dumper::Indent = 1;
 9
10  # Builds AST
11  my @stack;
12  my $regexp = qr{
13      (?&exp)
14
15      (?(DEFINE)
16          (?<exp>    (?&term) (?&re)
17                       (?{ say "exp -> term re" })
18          )
19
20          (?<re>     \s* ([+-]) (?&term)
21                        (?{  # intermediate action
22                            local our ($ch1, $term) = splice @stack, -2;
23
24                            push @stack, ADD->new( {left => $ch1, right => $term, op => g(1)
25                        })
26                     (?&re)
27                       (?{ say "re -> [+-] term re" })
28                   | # empty
29                       (?{ say "re -> empty" })
30          )
31
32          (?<term>   ((?&factor)) (?&rt)
33                        (?{
34                            say "term-> factor rt";
35                        })
36          )
37
38          (?<rt>     \s*([*/]) (?&factor)
39                         (?{  # intermediate action
40                              local our ($ch1, $ch2) = splice @stack, -2;
41
42                              push @stack, MULT->new({left => $ch1, right => $ch2, op => g(1
43                          })
44                     (?&rt) # end of <rt> definition
45                         (?{
46                              say "rt -> [*/] factor rt"
47                          })
48                   | # empty
49                         (?{ say "rt -> empty" })
50          )
51
52          (?<factor> \s* (\d+)
```

```
                        (?{
                           say "factor -> NUM($^N)";
                           push @stack, bless { 'val' => g(1) }, 'NUM';
                        })
                   | \s* \( (?&exp) \s* \)
                        (?{ say "factor -> ( exp )" })
        )
    )
}xms;

my $input = <>;
chomp($input);
if ($input =~ $regexp) {
  say "matches: $&";
  my $ast = pop @stack;
  say "AST:\n", Dumper $ast;

  say $ast->translate;
}
else {
  say "does not match";
}
```

Las Clases representando a los AST

Cada nodo del AST es un objeto. La clase del nodo nos dice que tipo de nodo es. Así los nodos de la clase `MULT` agrupan a los nódos de multiplicación y división. Los nodos de la clase `ADD` agrupan a los nódos de suma y resta. El procedimiento general es asociar un método `translate` con cada clase de nodo. De esta forma se logra el polimorfismo necesario: cada clase de nodo sabe como traducirse y el método `translate` de cada clase puede escribirse como

- Obtener los resultados de llamar a `$child->translate` para cada uno de los nodos hijos `$child`. Por ejemplo, si el nodo fuera un nodo `IF_ELSE` de un hipotético lenguaje de programación, se llamaría a los métodos `translate` sobre sus tres hijos `boolexpr`, `ifstatement` y `elsestatement`.
- Combinar los resultados para producir la traducción adecuada del nodo actual.

Es esta combinación la que mas puede cambiar según el tipo de nodo. Así, en el caso de el nodo `IF_ELSE` el seudocódigo para la traducción sería algo parecido a esto:

```
my $self = shift;
my $etiqueta1 = generar_nueva_etiqueta;
my $etiqueta2 = generar_nueva_etiqueta;

my $boolexpr      = $self->boolexpr->translate;
my $ifstatement   = $self->ifstatement->translate,
my $elsestatement = $self->elsestatement->translate,
return << "ENDTRANS";
    $boolexpr
    JUMPZERO $etiqueta1:
    $ifstatement
    JUMP     $etiqueta2:
  $etiqueta1:
    $elsestatement
  $etiqueta2:
ENDTRANS
```

Siguiendo estas observaciones el código de `BinaryOp.pm` queda así:

```
pl@nereida:~/Lperltesting$ cat -n BinaryOp.pm
package BinaryOp;
use strict;
use base qw(Class::Accessor);

BinaryOp->mk_accessors(qw{left right op});

sub translate {
  my $self = shift;

  return $self->left->translate." ".$self->right->translate." ".$self->op;
}

package ADD;
use base qw{BinaryOp};

package MULT;
use base qw{BinaryOp};

package NUM;

sub translate {
  my $self = shift;

  return $self->{val};
}

1;
```

Véase también:

- Class::Accessor

Accediendo a los paréntesis lejanos: El módulo `Regexp::Paren`

En esta solución utilizamos las variables `@-` y `@+` para construir una función que nos permite acceder a lo que casó con los últimos paréntesis con memoria:

> *Since Perl 5.6.1 the special variables* `@-` *and* `@+` *can functionally replace* `` $` ``, `$&` *and* `$'`*. These arrays contain pointers to the beginning and end of each match (see* `perlvar` *for the full story), so they give you essentially the same information, but without the risk of excessive string copying.*

Véanse los párrafos en las páginas 44, 44) y 45 para mas información sobre `@-` y `@+`.

`g(1)` nos retorna lo que casó con el último paréntesis, `g(2)` lo que casó con el penúltimo, etc.

```
pl@nereida:~/Lperltesting$ cat -n Regexp/Paren.pm
package Regexp::Paren;
use strict;

use base qw{Exporter};

our @EXPORT_OK = qw{g};

```

```
sub g {
  die "Error in ’Regexp::Paren::g’. Not used inside (?{ code }) construct\n" unless define
  my $ofs = - shift;

  # Number of parenthesis that matched
  my $np = @-;
  die "Error. Illegal ’Regexp::Paren::g’ ref inside (?{ code }) construct\n" unless ($np >
  # $_ contains the string being matched
  substr($_, $-[$ofs], $+[$np+$ofs] - $-[$ofs])
}

1;

=head1 NAME

Regexp::Paren - Extends $ˆN inside (?{ ... }) constructs

=head1 SYNOPSIS

  use Regexp::Paren qw{g};

  ’abcde’ =˜ qr{(.)(.)(.)
                       (?{ print g(1)." ".g(2)." ".g(3)."\n" })                   # c b a
               (.)     (?{ print g(1)." ".g(2)." ".g(3)." ".g(4)."\n" })          # d c b
               (.)     (?{ print g(1)." ".g(2)." ".g(3)." ".g(4)." ".g(5)."\n" }) # e d c
              }x;

  print g(1)." ".g(2)." ".g(3)." ".g(4)." ".g(5)."\n"; # error!

=head1 DESCRIPTION

Inside a C<(?{ ... })> construct, C<g(1)> refers to what matched the last parenthesis
(like C<$ˆN>), C<g(2)> refers to the string that matched with the parenthesis before
the last, C<g(3)> refers to the string that matched with the parenthesis at distance 3,
etc.

=head1 SEE ALSO

=over 2

=item * L<perlre>

=item * L<perlretut>

=item * PerlMonks node I<Strange behavior o> C<@-> I<and> C<@+> I<in perl5.10 regexps> L<h

=item * PerlMonks node I<Backreference variables in code embedded inside Perl 5.10 regexps

=back

=head1 AUTHOR

Casiano Rodriguez-Leon (casiano@ull.es)
```

```

=head1 ACKNOWLEDGMENTS

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

=head1 LICENCE AND COPYRIGHT

Copyright (c) 2009- Casiano Rodriguez-Leon (casiano@ull.es). All rights reserved.

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

Al ejecutar `perldoc Regexp::Paren` podemos ver la documentación incluida (véase la documentación en `perlpod` y `perlpodspec` así como la sección La Documentación en Perl para mas detalles):

```
NAME
    Regexp::Paren - Extends $^N inside (?{ ... }) constructs

SYNOPSIS
      use Regexp::Paren qw{g};

      ’abcde’ =~ qr{(.)(.)(.)
                           (?{ print g(1)." ".g(2)." ".g(3)."\n" })                   # c
                   (.)     (?{ print g(1)." ".g(2)." ".g(3)." ".g(4)."\n" })          # d
                   (.)     (?{ print g(1)." ".g(2)." ".g(3)." ".g(4)." ".g(5)."\n" }) # e
                  }x;

      print g(1)." ".g(2)." ".g(3)." ".g(4)." ".g(5)."\n"; # error!

DESCRIPTION
    Inside a "(?{ ... })" construct, g(1) refers to what matched the last
    parenthesis (like $^N), g(2) refers to the string that matched with the
    parenthesis before the last, g(3) refers to the string that matched with
    the parenthesis at distance 3, etc.

SEE ALSO
    * perlre
    * perlretut
    * PerlMonks node *Strange behavior o* "@-" *and* "@+" *in perl5.10
      regexps* <http://www.perlmonks.org/?node_id=794736>
    * PerlMonks node *Backreference variables in code embedded inside Perl
      5.10 regexps* <http://www.perlmonks.org/?node_id=794424>

AUTHOR
    Casiano Rodriguez-Leon (casiano@ull.es)

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.

LICENCE AND COPYRIGHT
    Copyright (c) 2009- Casiano Rodriguez-Leon (casiano@ull.es). All rights
```

1.10. Práctica: Traducción de `invitation` a `HTML`

Esta práctica es continuación de la práctica *un lenguaje para componer invitaciones* especificada en la sección 1.8.

El objetivo es traducir la entrada escrita en el lenguaje de invitaciones a HTML. La traducción del ejemplo anterior debería ser parecida a esta:

```
pl@nereida:~/Lpl0910/Practicas/161009/src$ cat -n invitat
 1  <?xml version="1.0"?>
 2  <!DOCTYPE invitation SYSTEM "invitation.dtd">
 3  <invitation>
 4  <!-- ++++ The header part of the document ++++ -->
 5  <front>
 6  <to>Anna, Bernard, Didier, Johanna</to>
 7  <date>Next Friday Evening at 8 pm</date>
 8  <where>The Web Cafe</where>
 9  <why>My first XML baby</why>
10  </front>
11  <!-- +++++ The main part of the document +++++ -->
12  <body>
13  <par>
14  I would like to invite you all to celebrate
15  the birth of <emph>Invitation</emph>, my
16  first XML document child.
17  </par>
18  <par>
19  Please do your best to come and join me next Friday
20  evening. And, do not forget to bring your friends.
21  </par>
22  <par>
23  I <emph>really</emph> look forward to see you soon!
24  </par>
25  </body>
26  <!-- +++ The closing part of the document ++++ -->
27  <back>
28  <signature>Michel</signature>
29  </back>
30  </invitation>
```

Para ver el resultado en su navegador visite el fichero invitation.html

Su programa deberá producir un Abstract Syntax Tree. Los nodos serán objetos. Cada clase (`FRONT`, `TO`, etc.) deberá de disponer de un método `translate`.

Para simplificar el proceso de traducción a `HTML` se sugiere utilizar una hoja de estilo parecida a la siguiente (tomada de la seción 7.4.4 del citado libro de Goosens):

```
pl@nereida:~/Lpl0910/Practicas/161009/src$ cat -n invit.css
 1  /* CSS stylesheet for invitation1 in HTML */
 2  BODY {margin-top: 1em;     /* global page parameters */
 3        margin-bottom: 1em;
 4        margin-left: 1em;
 5        margin-right: 1em;
 6        font-family: serif;
 7        line-height: 1.1;
 8        color: black;
 9  }
```

```
H1   {text-align: center;  /* for global title   */
      font-size: x-large;
}
P    {text-align: justify; /* paragraphs in body */
      margin-top: 1em;
}
TABLE { border-width: 0pt }
TBODY { border-width: 0pt }
TD[class="front"] {        /* table data in front matter */
      text-align: left;
      font-weight: bold;
}
TD.front {        /* table data in front matter */
      text-align: left;
      font-weight: bold;
}
EM   {font-style: italic;  /* emphasis in body   */
}
P.signature {     /* signature          */
      text-align: right;
      font-weight: bold;
}
```

Véase también:

- The LaTex Web Companion
- Examples from The LaTeX Web Companion (véanse los subdirectorios correspondietnes a los capítulos 6 y 7)
- CSS Tutorial
- Edición extremadamente simple de HTML
- Perl-XML Frequently Asked Questions

1.11. Análisis Sintáctico con `Regexp::Grammars`

El módulo Regexp::Grammars escrito por Damian Conway extiende las expresiones regulares Perl con la capacidad de generar representaciones del árbol de análisis sintáctico abstracto y obviando la necesidad de explicitar los blancos. El módulo necesita para funcionar una versión de Perl superior o igual a la 5.10.

1.11.1. Introducción

El Problema

La documentación de Regexp::Grammars establece cual es el problema que aborda el módulo:

> *. . . Perl5.10 makes possible to use regexes to recognize complex, hierarchical–and even recursive–textual structures. The problem is that Perl 5.10 doesn't provide any support for extracting that hierarchical data into nested data structures. In other words, using Perl 5.10 you can match complex data, but not parse it into an internally useful form.*
>
> *An additional problem when using Perl 5.10 regexes to match complex data formats is that you have to make sure you remember to insert whitespace- matching constructs (such as* `\s*`*) at every possible position where the data might contain ignorable whitespace. This reduces the readability of such patterns, and increases the chance of errors (typically caused by overlooking a location where whitespace might appear).*

> **Una solución:** `Regexp::Grammars`
>
> *The Regexp::Grammars module solves both those problems.*
>
> *If you import the module into a particular lexical scope, it preprocesses any regex in that scope, so as to implement a number of extensions to the standard Perl 5.10 regex syntax. These extensions simplify the task of defining and calling subrules within a grammar, and allow those subrule calls to capture and retain the components of they match in a proper hierarchical manner.*

La sintaxis de una expresión regular `Regexp::Grammars`

Las expresiones regulares Regexp::Grammars aumentan las regexp Perl 5.10. La sintáxis se expande y se modifica:

> *A Regexp::Grammars specification consists of a pattern (which may include both standard Perl 5.10 regex syntax, as well as special Regexp::Grammars directives), followed by one or more rule or token definitions.*

Sigue un ejemplo:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n balanced_brackets.pl
use strict;
use warnings;
use 5.010;
use Data::Dumper;

my $rbb = do {
    use Regexp::Grammars;
    qr{
      (<pp>)

      <rule: pp>   \( (?: [^()]*+ | <escape> | <pp> )* \)

      <token: escape> \\.

    }xs;
};

while (my $input = <>) {
    while ($input =~ m{$rbb}g) {
        say("matches: <$&>");
        say Dumper \%/;
    }
}
```

> *Note that there is no need to explicitly place* `\s*` *subpatterns throughout the rules; that is taken care of automatically.*
>
> *...*
>
> *The initial pattern (*`(<pp>)`*) acts like the* top *rule of the grammar, and must be matched completely for the grammar to match.*
>
> *The rules and tokens are declarations only and they are not directly matched. Instead, they act like subroutines, and are invoked by name from the initial pattern (or from within a rule or token).*
>
> *Each rule or token extends from the directive that introduces it up to either the next rule or token directive, or (in the case of the final rule or token) to the end of the grammar.*

El hash %/: Una representación del AST Al ejecutar el programa anterior con entrada `(2*(3+5))*4+(2-3)` produce:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 balanced_brackets.pl
(2*(3+5))*4+(2-3)
matches: <(2*(3+5))>
$VAR1 = {
          ’’ => ’(2*(3+5))’,
          ’pp’ => {
                    ’’ => ’(2*(3+5))’,
                    ’pp’ => ’(3+5)’
                  }
        };

matches: <(2-3)>
$VAR1 = {
          ’’ => ’(2-3)’,
          ’pp’ => ’(2-3)’
        };
```

> *Each rule calls the subrules specified within it, and then return a hash containing whatever result each of those subrules returned, with each result indexed by the subrule’s name.*
>
> *In this way, each level of the hierarchical regex can generate hashes recording everything its own subrules matched, so when the entire pattern matches, it produces a tree of nested hashes that represent the structured data the pattern matched.*
>
> *...*
>
> *In addition each result-hash has one extra key: the empty string. The value for this key is whatever string the entire subrule call matched.*

Diferencias entre `token` y `rule`

> *The difference between a token and a rule is that a token treats any whitespace within it exactly as a normal Perl regular expression would. That is, a sequence of whitespace in a token is ignored if the `/x` modifier is in effect, or else matches the same literal sequence of whitespace characters (if `/x` is not in effect).*

En el ejemplo anterior el comportamiento es el mismo si se reescribe la regla para el token `escape` como:

```
    13        <rule: escape> \\.
```

En este otro ejemplo mostramos que la diferencia entre token y rule es significativa:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n tokenvsrule.pl
     1  use strict;
     2  use warnings;
     3  use 5.010;
     4  use Data::Dumper;
     5
     6  my $rbb = do {
     7      use Regexp::Grammars;
     8      qr{
     9        <s>
    10
    11        <rule: s> <a> <c>
```

```

      <rule: c>  c d

      <token: a>  a b

    }xs;
};

while (my $input = <>) {
    if ($input =˜ m{$rbb}) {
        say("matches: <$&>");
        say Dumper \%/;
    }
    else {
        say "Does not match";
    }
}
```

Al ejecutar este programa vemos la diferencia en la interpretación de los blancos:

```
pl@nereida:˜/Lregexpgrammars/demo$ perl5.10.1 tokenvsrule.pl
ab c d
matches: <ab c d>
$VAR1 = {
          ’’ => ’ab c d’,
          ’s’ => {
                   ’’ => ’ab c d’,
                   ’c’ => ’c d’,
                   ’a’ => ’ab’
                 }
        };

a b c d
Does not match
ab cd
matches: <ab cd>
$VAR1 = {
          ’’ => ’ab cd’,
          ’s’ => {
                   ’’ => ’ab cd’,
                   ’c’ => ’cd’,
                   ’a’ => ’ab’
                 }
        };
```

Obsérvese como la entrada `a b c d` es rechazada mientras que la entrada `ab c d` es aceptada.

Redefinición de los espacios en blanco

> *In a rule, any sequence of whitespace (except those at the very start and the very end of the rule) is treated as matching the implicit subrule* `<.ws>`*, which is automatically predefined to match optional whitespace (i.e.* `\s*`*).*
>
> *You can explicitly define a* `<ws>` *token to change that default behaviour. For example, you could alter the definition of whitespace to include Perlish comments, by adding an explicit* `<token: ws>`*:*

```
<token: ws>
   (?: \s+ | #[^\n]* )*
```

> *But be careful not to define* `<ws>` *as a rule, as this will lead to all kinds of infinitely recursive unpleasantness.*

El siguiente ejemplo ilustra como redefinir `<ws>`:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n tokenvsruleandws.pl
use strict;
use warnings;
use 5.010;
use Data::Dumper;

my $rbb = do {
    use Regexp::Grammars;
    no warnings 'uninitialized';
    qr{
      <s>

      <token: ws> (?: \s+ | /\* .*? \*/)*+

      <rule: s> <a> <c>

      <rule: c>  c d

      <token: a>  a b

    }xs;
};

while (my $input = <>) {
    if ($input =~ m{$rbb}) {
        say Dumper \%/;
    }
    else {
        say "Does not match";
    }
}
```

Ahora podemos introducir comentarios en la entrada:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 -w tokenvsruleandws.pl
ab /* 1 */ c d
$VAR1 = {
          '' => 'ab /* 1 */ c d',
          's' => {
                   '' => 'ab /* 1 */ c d',
                   'c' => 'c d',
                   'a' => 'ab'
                 }
        };
```

Llamando a las subreglas

> *To invoke a rule to match at any point, just enclose the rule's name in angle brackets (like in Perl 6). There must be no space between the opening bracket and the rulename. For example:*
>
> ```
> qr{
>     file:             # Match literal sequence ’f’ ’i’ ’l’ ’e’ ’:’
>     <name>            # Call <rule: name>
>     <options>?        # Call <rule: options> (it’s okay if it fails)
>
>     <rule: name>
>         # etc.
> }x;
> ```
>
> *If you need to match a literal pattern that would otherwise look like a subrule call, just backslash-escape the leading angle:*
>
> ```
> qr{
>     file:             # Match literal sequence ’f’ ’i’ ’l’ ’e’ ’:’
>     \<name>           # Match literal sequence ’<’ ’n’ ’a’ ’m’ ’e’ ’>’
>     <options>?        # Call <rule: options> (it’s okay if it fails)
>
>     <rule: name>
>         # etc.
> }x;
> ```

El siguiente programa ilustra algunos puntos discutidos en la cita anterior:

```
casiano@millo:~/src/perl/regexp-grammar-examples$ cat -n badbracket.pl
 1  use strict;
 2  use warnings;
 3  use 5.010;
 4  use Data::Dumper;
 5
 6  my $rbb = do {
 7      use Regexp::Grammars;
 8      qr{
 9        (<pp>)
10
11        <rule: pp>   \( (?: <b  > | \< | < escape> | <pp> )* \)
12
13        <token: b  > b
14
15        <token: escape> \\.
16
17      }xs;
18  };
19
20  while (my $input = <>) {
21      while ($input =~ m{$rbb}g) {
22          say("matches: <$&>");
23          say Dumper \%/;
24      }
25  }
```

Obsérvense los blancos en `< escape>` y en `<token: b   > b`. Pese a ello el programa funciona:

```
casiano@millo:~/src/perl/regexp-grammar-examples$ perl5.10.1 badbracket.pl
(\(\))
matches: <(\(\))>
$VAR1 = {
          ’’ => ’(\\(\\))’,
          ’pp’ => {
                    ’’ => ’(\\(\\))’,
                    ’escape’ => ’\\)’
                  }
        };

(b)
matches: <(b)>
$VAR1 = {
          ’’ => ’(b)’,
          ’pp’ => {
                    ’’ => ’(b)’,
                    ’b’ => ’b’
                  }
        };

(<)
matches: <(<)>
$VAR1 = {
          ’’ => ’(<)’,
          ’pp’ => ’(<)’
        };

(c)

casiano@millo:
```

Eliminación del anidamiento de ramas unarias en %/

> *. . . Note, however, that if the result-hash at any level contains only the empty-string key (i.e. the subrule did not call any sub-subrules or save any of their nested result-hashes), then the hash is unpacked and just the matched substring itself if returned.*
>
> *For example, if* `<rule: sentence>` *had been defined:*
>
> ```
> <rule: sentence>
>     I see dead people
> ```
>
> *then a successful call to the rule would only add:*
>
> ```
> sentence => ’I see dead people’
> ```
>
> *to the current result-hash.*
>
> *This is a useful feature because it prevents a series of nested subrule calls from producing very unwieldy data structures. For example, without this automatic unpacking, even the simple earlier example:*
>
> ```
> <rule: sentence>
>     <noun> <verb> <object>
> ```

would produce something needlessly complex, such as:

```
sentence => {
    ""     => ’I saw a dog’,
    noun   => {
        "" => ’I’,
    },
    verb   => {
        "" => ’saw’,
    },
    object => {
        ""      => ’a dog’,
        article => {
            "" => ’a’,
        },
        noun    => {
            "" => ’dog’,
        },
    },
}
```

El siguiente ejemplo ilustra este punto:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n unaryproductions.pl
use strict;
use warnings;
use 5.010;
use Data::Dumper;

my $rbb = do {
    use Regexp::Grammars;
    qr{
      <s>

      <rule: s> <noun> <verb> <object>

      <token: noun> he | she | Peter | Jane

      <token: verb> saw | sees

      <token: object> a\s+dog | a\s+cat

    }x;
};

while (my $input = <>) {
    while ($input =~ m{$rbb}g) {
        say("matches: <$&>");
        say Dumper \%/;
    }
}
```

Sigue una ejecución del programa anterior:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 unaryproductions.pl
```

```
he saw a dog
matches: <he saw a dog>
$VAR1 = {
          '' => 'he saw a dog',
          's' => {
                   '' => 'he saw a dog',
                   'object' => 'a dog',
                   'verb' => 'saw',
                   'noun' => 'he'
                 }
        };

Jane sees a cat
matches: <Jane sees a cat>
$VAR1 = {
          '' => 'Jane sees a cat',
          's' => {
                   '' => 'Jane sees a cat',
                   'object' => 'a cat',
                   'verb' => 'sees',
                   'noun' => 'Jane'
                 }
        };
```

Ámbito de uso de `Regexp::Grammars`

Cuando se usa Regexp::Grammars como parte de un programa que utiliza otras regexes hay que evitar que Regexp::Grammars procese las mismas. Regexp::Grammars reescribe las expresiones regulares durante la fase de preproceso. Esta por ello presenta las mismas limitaciones que cualquier otra forma de 'source filtering' (véase `perlfilter`). Por ello es una buena idea declarar la gramática en un bloque `do` restringiendo de esta forma el ámbito de acción del módulo.

```
my $calculator = do{
    use Regexp::Grammars;
    qr{
         ........
    }xms
};
```

1.11.2. Objetos

> *When a grammar has parsed successfully, the* `%/` *variable will contain a series of nested hashes (and possibly arrays) representing the hierarchical structure of the parsed data.*
>
> *Typically, the next step is to walk that tree, extracting or converting or otherwise processing that information. If the tree has nodes of many different types, it can be difficult to build a recursive subroutine that can navigate it easily.*
>
> *A much cleaner solution is possible if the nodes of the tree are proper objects. In that case, you just define a* `trasnlate()` *method for each of the classes, and have every node call that method on each of its children. The chain of* `translate()` *calls would cascade down the nodes of the tree, each one invoking the appropriate* `translate()` *method according to the type of node encountered.*
>
> *The only problem is that, by default, Regexp::Grammars returns a tree of plain-old hashes, not* `Class::Whatever` *objects. Fortunately, it's easy to request that the result has-*

hes be automatically blessed into the appropriate classes, using the `<objrule:...>` *and* `<objtoken:...>` *directives.*

These directives are identical to the `<rule:...>` *and* `<token:...>` *directives (respectively), except that the rule or token they create will also bless the hash it normally returns, converting it to an object of a class whose name is the same as the rule or token itself.*

For example:

```
<objrule: Element>
    # ...Defines a rule that can be called as <Element>
    # ...and which returns a hash-based Element object
```

The `IDENTIFIER` *of the rule or token may also be fully qualified. In such cases, the rule or token is defined using only the final* short name*, but the result object is blessed using the fully qualified* long name*. For example:*

```
<objrule: LaTeX::Element>
    # ...Defines a rule that can be called as <Element>
    # ...and which returns a hash-based LaTeX::Element object
```

This can be useful to ensure that returned objects don't collide with other namespaces in your program.

Note that you can freely mix object-returning and plain-old-hash-returning rules and tokens within a single grammar, though you have to be careful not to subsequently try to call a method on any of the unblessed nodes.

1.11.3. Renombrando los resultados de una subregla

Nombre de la regla versus Nombre del Resultado

No siempre el nombre de la regla es el mas apropiado para ser el nombre del resultado:

> *It is not always convenient to have subrule results stored under the same name as the rule itself. Rule names should be optimized for understanding the behaviour of the parser, whereas result names should be optimized for understanding the structure of the data. Often those two goals are identical, but not always; sometimes rule names need to describe what the data looks like, while result names need to describe what the data means.*

Colisión de nombres de reglas

For example, sometimes you need to call the same rule twice, to match two syntactically identical components whose positions give then semantically distinct meanings:

```
<rule: copy_cmd>
    copy <file> <file>
```

The problem here is that, if the second call to `<file>` *succeeds, its result-hash will be stored under the key* `file`*, clobbering the data that was returned from the first call to* `<file>`*.*

Aliasing

To avoid such problems, Regexp::Grammars allows you to alias any subrule call, so that it is still invoked by the original name, but its result-hash is stored under a different key. The syntax for that is: `<alias=rulename>`*. For example:*

```
<rule: copy_cmd>
    copy <from=file> <to=file>
```

Here, `<rule: file>` *is called twice, with the first result-hash being stored under the key* `from`*, and the second result-hash being stored under the key* `to`*.*

Note, however, that the alias before the `=` *must be a proper identifier (i.e. a letter or underscore, followed by letters, digits, and/or underscores). Aliases that start with an underscore and aliases named* `MATCH` *have special meaning.*

Normalización de los resultados mediante aliasing

Aliases can also be useful for normalizing data that may appear in different formats and sequences. For example:

```
<rule: copy_cmd>
    copy <from=file>        <to=file>
  | dup    <to=file>  as  <from=file>
  |      <from=file>  ->    <to=file>
  |        <to=file>  <-  <from=file>
```

Here, regardless of which order the old and new files are specified, the result-hash always gets:

```
copy_cmd => {
    from => 'oldfile',
      to => 'newfile',
}
```

Ejemplo

El siguiente programa ilustra los comentarios de la documentación:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n copygrammar.pl
     1  use strict;
     2  use warnings;
     3  use 5.010;
     4  use Data::Dumper;
     5
     6  my $rbb = do {
     7      use Regexp::Grammars;
     8      qr{
     9        <copy_cmd>
    10
    11        <rule: copy_cmd>
    12              copy <from=file> <to=file>
    13          |   <from=file> ->   <to=file>
    14          |   <to=file>   <- <from=file>
    15
    16        <token: file> [\w./\\]+
    17      }x;
    18  };
    19
    20  while (my $input = <>) {
    21      while ($input =~ m{$rbb}g) {
    22          say("matches: <$&>");
    23          say Dumper \%/;
    24      }
    25  }
```

Cuando lo ejecutamos obtenemos:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 copygrammar.pl
copy a b
matches: <copy a b>
$VAR1 = {
          '' => 'copy a b',
          'copy_cmd' => {
                          '' => 'copy a b',
                          'to' => 'b',
                          'from' => 'a'
                        }
        };

b <- a
matches: <b <- a>
$VAR1 = {
          '' => 'b <- a',
          'copy_cmd' => {
                          '' => 'b <- a',
                          'to' => 'b',
                          'from' => 'a'
                        }
        };

a -> b
matches: <a -> b>
$VAR1 = {
          '' => 'a -> b',
          'copy_cmd' => {
                          '' => 'a -> b',
                          'to' => 'b',
                          'from' => 'a'
                        }
        };
```

1.11.4. Listas

El operador de cierre positivo

> *If a subrule call is quantified with a repetition specifier:*
>
> ```
> <rule: file_sequence>
>     <file>+
> ```
>
> *then each repeated match overwrites the corresponding entry in the surrounding rule's result-hash, so only the result of the final repetition will be retained. That is, if the above example matched the string* `foo.pl bar.py baz.php`*, then the result-hash would contain:*
>
> ```
> file_sequence {
>     ""   => 'foo.pl bar.py baz.php',
>     file => 'baz.php',
> }
> ```

Operadores de listas y espacios en blanco

Existe un caveat con el uso de los operadores de repetición y el manejo de los blancos. Véase el siguiente programa:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n numbers3.pl
 1  use strict;
 2  use warnings;
 3  use 5.010;
 4  use Data::Dumper;
 5
 6  my $rbb = do {
 7      use Regexp::Grammars;
 8
 9      qr{
10        <numbers>
11
12        <rule: numbers>
13          (<number>)+
14
15        <token: number> \s*\d+
16      }xms;
17  };
18
19  while (my $input = <>) {
20      if ($input =~ m{$rbb}) {
21          say("matches: <$&>");
22          say Dumper \%/;
23      }
24  }
```

Obsérvese el uso explícito de espacios `\s*\d+` en la definición de `number`.

Sigue un ejemplo de ejecución:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5_10_1 numbers3.pl
1 2 3 4
matches: <1 2 3 4>
$VAR1 = {
          '' => '1 2 3 4',
          'numbers' => {
                         '' => '1 2 3 4',
                         'number' => ' 4'
                       }
        };
```

Si se eliminan los blancos de la definición de `number`:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n numbers.pl
     1  use strict;
     2  use warnings;
     3  use 5.010;
     4  use Data::Dumper;
     5
     6  my $rbb = do {
     7      use Regexp::Grammars;
     8
```

```
    qr{
      <numbers>

      <rule: numbers>
        (<number>)+

      <token: number> \d+
    }xms;
};

while (my $input = <>) {
    if ($input =~ m{$rbb}) {
        say("matches: <$&>");
        say Dumper \%/;
    }
}
```

se obtiene una conducta que puede sorprender:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 numbers.pl
12 34 56
matches: <12>
$VAR1 = {
          '' => '12',
          'numbers' => {
                         '' => '12',
                         'number' => '12'
                       }
        };
```

La explicación está en la documentación: véase la sección Grammar Syntax:

> `<rule: IDENTIFIER>`
>
> *Define a rule whose name is specified by the supplied identifier.*
>
> *Everything following the* `<rule:...>` *directive (up to the next* `<rule:...>` *or* `<token:...>` *directive) is treated as part of the rule being defined.*
>
> Any whitespace in the rule is replaced by a call to the `<.ws>` *subrule (which defaults to matching* `\s*`*, but may be explicitly redefined).*

También podríamos haber resuelto el problema introduciendo un blanco explícito dentro del cierre positivo:

```
<rule: numbers>
  (<number> )+

<token: number> \d+
```

Una Solución al problema de recordar los resultados de una lista: El uso de brackets

> *Usually, that's not the desired outcome, so Regexp::Grammars provides another mechanism by which to call a subrule; one that saves all repetitions of its results.*
>
> *A regular subrule call consists of the rule's name surrounded by angle brackets. If, instead, you surround the rule's name with* `<[...]>` *(angle and square brackets) like so:*
>
> ```
> <rule: file_sequence>
>     <[file]>+
> ```

> *then the rule is invoked in exactly the same way, but the result of that submatch is pushed onto an array nested inside the appropriate result-hash entry. In other words, if the above example matched the same* `foo.pl bar.py baz.php` *string, the result-hash would contain:*
>
> ```
> file_sequence {
>     ""   => ’foo.pl bar.py baz.php’,
>     file => [ ’foo.pl’, ’bar.py’, ’baz.php’ ],
> }
> ```

Teniendo en cuenta lo dicho anteriormente sobre los blancos dentro de los cuantificadores, es necesario introducir blancos dentro del operador de repetición:

```
pl@nereida:˜/Lregexpgrammars/demo$ cat -n numbers4.pl
use strict;
use warnings;
use 5.010;
use Data::Dumper;

my $rbb = do {
    use Regexp::Grammars;

    qr{
      <numbers>

      <rule: numbers>
        (?:  <[number]> )+

      <token: number> \d+
    }xms;
};

while (my $input = <>) {
    if ($input =˜ m{$rbb}) {
        say("matches: <$&>");
        say Dumper \%/;
    }
}
```

Al ejecutar este programa obtenemos:

```
pl@nereida:˜/Lregexpgrammars/demo$ perl5_10_1 numbers4.pl
1 2 3 4
matches: <1 2 3 4
>
$VAR1 = {
          ’’ => ’1 2 3 4
’,
          ’numbers’ => {
                         ’’ => ’1 2 3 4
’,
                         ’number’ => [ ’1’, ’2’, ’3’, ’4’ ]
                       }
        };
```

Otra forma de resolver las colisiones de nombres: salvarlos en una lista

> *This* listifying subrule call *can also be useful for non-repeated subrule calls, if the same subrule is invoked in several places in a grammar. For example if a cmdline option could be given either one or two values, you might parse it:*
>
> ```
> <rule: size_option>
>     -size <[size]> (?: x <[size]> )?
> ```
>
> *The result-hash entry for* `size` *would then always contain an array, with either one or two elements, depending on the input being parsed.*

Sigue un ejemplo:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n sizes.pl
 1  use strict;
 2  use warnings;
 3  use 5.010;
 4  use Data::Dumper;
 5
 6  my $rbb = do {
 7      use Regexp::Grammars;
 8
 9      qr{
10        <command>
11
12        <rule: command> ls <size_option>
13
14        <rule: size_option>
15            -size <[size]> (?: x <[size]> )?
16
17        <token: size> \d+
18      }x;
19  };
20
21  while (my $input = <>) {
22      while ($input =~ m{$rbb}g) {
23          say("matches: <$&>");
24          say Dumper \%/;
25      }
26  }
```

Veamos su comportamiento con diferentes entradas:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 sizes.pl
ls -size 4
matches: <ls -size 4
>
$VAR1 = {
          '' => 'ls -size 4
',
          'command' => {
                         'size_option' => {
                                            '' => '-size 4
',
```

```
                                            'size’ => [ ’4’ ]
                                          },
                         ’’ => ’ls -size 4
’
                       }
        };

ls -size 2x8
matches: <ls -size 2x8
>
$VAR1 = {
          ’’ => ’ls -size 2x8
’,
          ’command’ => {
                         ’size_option’ => {
                                            ’’ => ’-size 2x8
’,
                                            ’size’ => [ ’2’, ’8’ ]
                                          },
                         ’’ => ’ls -size 2x8
’
                       }
        };
```

Aliasing de listas

> *Listifying subrules can also be given aliases, just like ordinary subrules. The alias is always specified inside the square brackets:*
>
> ```
> <rule: size_option>
>     -size <[size=pos_integer]> (?: x <[size=pos_integer]> )?
> ```
>
> *Here, the sizes are parsed using the* `pos_integer` *rule, but saved in the result-hash in an array under the key* `size`*.*

Sigue un ejemplo:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n aliasedsizes.pl
use strict;
use warnings;
use 5.010;
use Data::Dumper;

my $rbb = do {
    use Regexp::Grammars;

    qr{
      <command>

      <rule: command> ls <size_option>

      <rule: size_option>
          -size <[size=int]> (?: x <[size=int]> )?

```

```
      <token: int> \d+
    }x;
};

while (my $input = <>) {
    while ($input =~ m{$rbb}g) {
        say("matches: <$&>");
        say Dumper \%/;
    }
}
```

Veamos el resultado de una ejecución:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 aliasedsizes.pl
ls -size 2x4
matches: <ls -size 2x4
>
$VAR1 = {
          '' => 'ls -size 2x4
',
          'command' => {
                         'size_option' => {
                                            '' => '-size 2x4
',
                                            'size' => [
                                                        '2',
                                                        '4'
                                                      ]
                                          },
                         '' => 'ls -size 2x4
'
                       }
        };
```

Caveat: Cierres y Warnings

En este ejemplo aparece `<number>+` sin corchetes ni paréntesis:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n numbers5.pl
     1  use strict;
     2  use warnings;
     3  use 5.010;
     4  use Data::Dumper;
     5
     6  my $rbb = do {
     7      use Regexp::Grammars;
     8
     9      qr{
    10        <numbers>
    11
    12        <rule: numbers>
    13          <number>+
    14
    15        <token: number> \d+
    16      }xms;
```

```
};

while (my $input = <>) {
    if ($input =~ m{$rbb}) {
        say("matches: <$&>");
        say Dumper \%/;
    }
}
```

Este programa produce un mensaje de advertencia:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 numbers5.pl
  warn | Repeated subrule <number>+ will only capture its final match
       | (Did you mean <[number]>+ instead?)
       |
```

Si se quiere evitar el mensaje y se está dispuesto a asumir la pérdida de los valores asociados con los elementos de la lista se deberán poner el operando entre paréntesis (con o sin memoria).

Esto es lo que dice la documentación sobre este warning:

> `Repeated subrule <rule> will only capture its final match`
>
> *You specified a subrule call with a repetition qualifier, such as:*
>
> ```
> <ListElem>*
> ```
>
> *or:*
>
> ```
> <ListElem>+
> ```
>
> *Because each subrule call saves its result in a hash entry of the same name, each repeated match will overwrite the previous ones, so only the last match will ultimately be saved. If you want to save all the matches, you need to tell Regexp::Grammars to save the sequence of results as a nested array within the hash entry, like so:*
>
> ```
> <[ListElem]>*
> ```
>
> *or:*
>
> ```
> <[ListElem]>+
> ```
>
> *If you really did intend to throw away every result but the final one, you can silence the warning by placing the subrule call inside any kind of parentheses. For example:*
>
> ```
> (<ListElem>)*
> ```
>
> *or:*
>
> ```
> (?: <ListElem> )+
> ```

1.11.5. Pseudo sub-reglas

Subpatrones

> *Aliases can also be given to standard Perl subpatterns, as well as to code blocks within a regex. The syntax for subpatterns is:*
>
> ```
> <ALIAS= (SUBPATTERN) >
> ```
>
> *In other words, the syntax is exactly like an aliased subrule call, except that the rule name is replaced with a set of parentheses containing the subpattern. Any parentheses–capturing or non-capturing–will do.*
>
> *The effect of aliasing a standard subpattern is to cause whatever that subpattern matches to be saved in the result-hash, using the alias as its key. For example:*
>
> ```
> <rule: file_command>
>
>     <cmd=(mv|cp|ln)>  <from=file>  <to=file>
> ```
>
> *Here, the* `<cmd=(mv|cp|ln)>` *is treated exactly like a regular* `(mv|cp|ln)`*, but whatever substring it matches is saved in the result-hash under the key* `'cmd'`*.*

Sigue un ejemplo:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n subpattern.pl
 1  use strict;
 2  use warnings;
 3  use 5.010;
 4  use Data::Dumper;
 5
 6  my $rbb = do {
 7      use Regexp::Grammars;
 8
 9      qr{
10          <file_command>
11
12          <rule: file_command>
13
14          <cmd=(mv|cp|ln)>  <from=([\w./]+)>  <to=([\w./]+)>
15
16      }x;
17  };
18
19  while (my $input = <>) {
20      while ($input =~ m{$rbb}g) {
21          say("matches: <$&>");
22          say Dumper \%/;
23      }
24  }
```

y una ejecución:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 subpattern.pl
mv a b
matches: <mv a b>
$VAR1 = {
```

```
          '' => 'mv a b',
          'file_command' => {
                              '' => 'mv a b',
                              'to' => 'b',
                              'cmd' => 'mv',
                              'from' => 'a'
                            }
        };

cp c d
matches: <cp c d>
$VAR1 = {
          '' => 'cp c d',
          'file_command' => {
                              '' => 'cp c d',
                              'to' => 'd',
                              'cmd' => 'cp',
                              'from' => 'c'
                            }
        }
```

Bloques de código

The syntax for aliasing code blocks is:

```
<ALIAS= (?{ your($code->here) }) >
```

Note, however, that the code block must be specified in the standard Perl 5.10 regex notation: `(?{...})`*. A common mistake is to write:*

```
<ALIAS= { your($code->here } >
```

instead, which will attempt to interpolate `$code` *before the regex is even compiled, as such variables are only* protected *from interpolation inside a* `(?{...})`*.*

When correctly specified, this construct executes the code in the block and saves the result of that execution in the result-hash, using the alias as its key. Aliased code blocks are useful for adding semantic information based on which branch of a rule is executed. For example, consider the `copy_cmd` *alternatives shown earlier:*

```
<rule: copy_cmd>
    copy <from=file>        <to=file>
  | dup    <to=file>  as  <from=file>
  |      <from=file>  ->    <to=file>
  |        <to=file>  <-  <from=file>
```

Using aliased code blocks, you could add an extra field to the result- hash to describe which form of the command was detected, like so:

```
<rule: copy_cmd>
    copy <from=file>        <to=file>  <type=(?{ 'std' })>
  | dup    <to=file>  as  <from=file>  <type=(?{ 'rev' })>
  |      <from=file>  ->    <to=file>  <type=(?{ 'fwd' })>
  |        <to=file>  <-  <from=file>  <type=(?{ 'bwd' })>
```

Now, if the rule matched, the result-hash would contain something like:

```
         copy_cmd => {
             from => ’oldfile’,
               to => ’newfile’,
             type => ’fwd’,
         }
```

El siguiente ejemplo ilustra lo dicho en la documentación. En la línea 15 hemos introducido una regla para el control de errores[8]:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n aliasedcodeblock2.pl
 1  use strict;
 2  use warnings;
 3  use 5.010;
 4  use Data::Dumper;
 5
 6  my $rbb = do {
 7      use Regexp::Grammars;
 8      qr{
 9        <copy_cmd>
10
11        <rule: copy_cmd>
12              copy (<from=file>) (<to=file>) <type=(?{ ’std’ })>
13          |   <from=file> ->   <to=file> <type=(?{ ’fwd’ })>
14          |   <to=file>   <- <from=file> <type=(?{ ’bwd’ })>
15          |   .+ (?{ die "Syntax error!\n" })
16
17        <token: file> [\w./\\]+
18      }x;
19  };
20
21  while (my $input = <>) {
22      while ($input =~ m{$rbb}g) {
23          say("matches: <$&>");
24          say Dumper \%/;
25      }
26  }
```

La ejecución muestra el comportamiento del programa con tres entradas válidas y una errónea:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 aliasedcodeblock2.pl
copy a b
matches: <copy a b
>
$VAR1 = {
          ’’ => ’copy a b
’,
          ’copy_cmd’ => {
                          ’’ => ’copy a b
’,
                          ’to’ => ’b’,
                          ’from’ => ’a’,
                          ’type’ => ’std’
                        }
```

---

[8] Versión de `Grammar.pm` obtenida por email con las correcciones de Damian

```
        };

b <- a
matches: <b <- a
>
$VAR1 = {
          '' => 'b <- a
',
          'copy_cmd' => {
                          '' => 'b <- a
',
                          'to' => 'b',
                          'from' => 'a',
                          'type' => 'bwd'
                        }
        };

a -> b
matches: <a -> b
>
$VAR1 = {
          '' => 'a -> b
',
          'copy_cmd' => {
                          '' => 'a -> b
',
                          'to' => 'b',
                          'from' => 'a',
                          'type' => 'fwd'
                        }
        };

cp a b
Syntax error!
```

Pseudo subreglas y depuración

> *Note that, in addition to the semantics described above, aliased subpatterns and code blocks also become visible to Regexp::Grammars integrated debugger (see Debugging).*

1.11.6. Llamadas a subreglas desmemoriadas

> *By default, every subrule call saves its result into the result-hash, either under its own name, or under an alias.*
>
> *However, sometimes you may want to refactor some literal part of a rule into one or more subrules, without having those submatches added to the result-hash. The syntax for calling a subrule, but ignoring its return value is:*
>
> ```
> <.SUBRULE>
> ```
>
> *(which is stolen directly from Perl 6).*
>
> *For example, you may prefer to rewrite a rule such as:*
>
> ```
> <rule: paren_pair>
> ```

```
    \(
        (?: <escape> | <paren_pair> | <brace_pair> | [^()] )*
    \)
```

without any literal matching, like so:

```
<rule: paren_pair>

    <.left_paren>
        (?: <escape> | <paren_pair> | <brace_pair> | <.non_paren> )*
    <.right_paren>

<token: left_paren>   \(
<token: right_paren>  \)
<token: non_paren>    [^()]
```

Moreover, as the individual components inside the parentheses probably aren't being captured for any useful purpose either, you could further optimize that to:

```
<rule: paren_pair>

    <.left_paren>
        (?: <.escape> | <.paren_pair> | <.brace_pair> | <.non_paren> )*
    <.right_paren>
```

Note that you can also use the dot modifier on an aliased subpattern:

```
<.Alias= (SUBPATTERN) >
```

This seemingly contradictory behaviour (of giving a subpattern a name, then deliberately ignoring that name) actually does make sense in one situation. Providing the alias makes the subpattern visible to the debugger, while using the dot stops it from affecting the result-hash. See Debugging non-grammars *for an example of this usage.*

Ejemplo: Números entre comas

Por ejemplo, queremos reconocer listas de números separados por comas. Supongamos también que queremos darle un nombre a la expresión regular de separación. Quizá, aunque no es el caso, porque la expresión regular de separación sea suficientemente compleja. Si no usamos la notación *punto* la coma aparecerá en la estructura:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n numberscomma.pl
     1  use strict;
     2  use warnings;
     3  use 5.010;
     4  use Data::Dumper;
     5  $Data::Dumper::Indent = 1;
     6
     7  my $rbb = do {
     8      use Regexp::Grammars;
     9
    10      qr{
    11        <numbers>
    12
    13        <objrule: numbers>
```

```
            <[number]> (<comma> <[number]>)*

          <objtoken: number> \s*\d+
          <token: comma>  \s*,
        }xms;
};

while (my $input = <>) {
    if ($input =~ m{$rbb}) {
        say("matches: <$&>");
        say Dumper \%/;
    }
}
```

En efecto, aparece la clave `comma`:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 numberscomma.pl
2, 3, 4
matches: <2, 3, 4>
$VAR1 = {
  ’’ => ’2, 3, 4’,
  ’numbers’ => bless( {
    ’’ => ’2, 3, 4’,
    ’number’ => [
      bless( { ’’ => ’2’ }, ’number’ ),
      bless( { ’’ => ’3’ }, ’number’ ),
      bless( { ’’ => ’4’ }, ’number’ )
    ],
    ’comma’ => ’,’
  }, ’numbers’ )
};
```

Si cambiamos la llamada a la regla `<comma>` por `<.comma>`

```
pl@nereida:~/Lregexpgrammars/demo$ diff numberscomma.pl numberscomma2.pl
14c14
<          <[number]> (<comma> <[number]>)*
---
>          <[number]> (<.comma> <[number]>)*
```

eliminamos la aparición de la innecesaria clave:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 numberscomma2.pl
2, 3, 4
matches: <2, 3, 4>
$VAR1 = {
  ’’ => ’2, 3, 4’,
  ’numbers’ => bless( {
    ’’ => ’2, 3, 4’,
    ’number’ => [
      bless( { ’’ => ’2’ }, ’number’ ),
      bless( { ’’ => ’3’ }, ’number’ ),
      bless( { ’’ => ’4’ }, ’number’ )
    ]
  }, ’numbers’ )
};
```

1.11.7. Destilación del resultado

Destilación manual

> *Regexp::Grammars also offers full manual control over the distillation process. If you use the reserved word* `MATCH` *as the alias for a subrule call:*
>
> ```
> <MATCH=filename>
> ```
>
> *or a subpattern match:*
>
> ```
> <MATCH=( \w+ )>
> ```
>
> *or a code block:*
>
> ```
> <MATCH=(?{ 42 })>
> ```
>
> *then the current rule will treat the return value of that subrule, pattern, or code block as its complete result, and return that value instead of the usual result-hash it constructs. This is the case even if the result has other entries that would normally also be returned.*
>
> *For example, in a rule like:*
>
> ```
> <rule: term>
>       <MATCH=literal>
>     | <left_paren> <MATCH=expr> <right_paren>
> ```
>
> *The use of* `MATCH` *aliases causes the rule to return either whatever* `<literal>` *returns, or whatever* `<expr>` *returns (provided it's between left and right parentheses).*
>
> *Note that, in this second case, even though* `<left_paren>` *and* `<right_paren>` *are captured to the result-hash, they are not returned, because the* `MATCH` *alias overrides the normal* return the result-hash *semantics and returns only what its associated subrule (i.e.* `<expr>`*) produces.*

El siguiente ejemplo ilustra el uso del alias `MATCH`:

```
$ cat -n demo_calc.pl
#!/usr/local/lib/perl/5.10.1/bin/perl5.10.1
use v5.10;
use warnings;

my $calculator = do{
    use Regexp::Grammars;
    qr{
        <Answer>

        <rule: Answer>
            <X=Mult> <Op=([+-])> <Y=Answer>
          | <MATCH=Mult>

        <rule: Mult>
            <X=Pow> <Op=([*/%])> <Y=Mult>
          | <MATCH=Pow>

        <rule: Pow>
            <X=Term> <Op=(\^)> <Y=Pow>
          | <MATCH=Term>
```

```

        <rule: Term>
               <MATCH=Literal>
          | \( <MATCH=Answer> \)

        <token: Literal>
            <MATCH=( [+-]? \d++ (?: \. \d++ )?+ )>
    }xms
};

while (my $input = <>) {
    if ($input =~ $calculator) {
        use Data::Dumper 'Dumper';
        warn Dumper \%/;
    }
}
```

Veamos una ejecución:

```
$ ./demo_calc.pl
2+3*5
$VAR1 = {
          '' => '2+3*5',
          'Answer' => {
                        '' => '2+3*5',
                        'Op' => '+',
                        'X' => '2',
                        'Y' => {
                                 '' => '3*5',
                                 'Op' => '*',
                                 'X' => '3',
                                 'Y' => '5'
                               }
                      }
        };
4-5-2
$VAR1 = {
          '' => '4-5-2',
          'Answer' => {
                        '' => '4-5-2',
                        'Op' => '-',
                        'X' => '4',
                        'Y' => {
                                 '' => '5-2',
                                 'Op' => '-',
                                 'X' => '5',
                                 'Y' => '2'
                               }
                      }
        };
```

Obsérvese como el árbol construido para la expresión `4-5-2` se hunde a derechas dando lugar a una jerarquía errónea. Para arreglar el problema sería necesario eliminar la recursividad por la izquierda en las reglas correspondientes.

Destilación en el programa

It's also possible to control what a rule returns from within a code block. Regexp::Grammars provides a set of reserved variables that give direct access to the result-hash.

The result-hash itself can be accessed as `%MATCH` *within any code block inside a rule. For example:*

```
<rule: sum>
    <X=product> \+ <Y=product>
        <MATCH=(?{ $MATCH{X} + $MATCH{Y} })>
```

Here, the rule matches a product (aliased `'X'` *in the result-hash), then a literal* `'+'`*, then another product (aliased to* `'Y'` *in the result-hash). The rule then executes the code block, which accesses the two saved values (as* `$MATCH{X}` *and* `$MATCH{Y}`*), adding them together. Because the block is itself aliased to* `MATCH`*, the sum produced by the block becomes the (only) result of the rule.*

It is also possible to set the rule result from within a code block (instead of aliasing it). The special override *return value is represented by the special variable* `$MATCH`*. So the previous example could be rewritten:*

```
<rule: sum>
    <X=product> \+ <Y=product>
        (?{ $MATCH = $MATCH{X} + $MATCH{Y} })
```

Both forms are identical in effect. Any assignment to `$MATCH` *overrides the normal* return all subrule results *behaviour.*

Assigning to `$MATCH` *directly is particularly handy if the result may not always be* distillable*, for example:*

```
<rule: sum>
    <X=product> \+ <Y=product>
        (?{ if (!ref $MATCH{X} && !ref $MATCH{Y}) {
                # Reduce to sum, if both terms are simple scalars...
                $MATCH = $MATCH{X} + $MATCH{Y};
            }
            else {
                # Return full syntax tree for non-simple case...
                $MATCH{op} = '+';
            }
        })
```

Note that you can also partially override the subrule return behaviour. Normally, the subrule returns the complete text it matched under the empty key *of its result-hash. That is, of course,* `$MATCH{""}`*, so you can override just that behaviour by directly assigning to that entry.*

For example, if you have a rule that matches key/value pairs from a configuration file, you might prefer that any trailing comments not be included in the matched text entry of the rule's result-hash. You could hide such comments like so:

```
<rule: config_line>
    <key> : <value>  <comment>?
        (?{
            # Edit trailing comments out of "matched text" entry...
            $MATCH = "$MATCH{key} : $MATCH{value}";
        })
```

Some more examples of the uses of `$MATCH`*:*

```
<rule: FuncDecl>
  # Keyword  Name               Keep return the name (as a string)...
    func     <Identifier> ;     (?{ $MATCH = $MATCH{'Identifier'} })


<rule: NumList>
  # Numbers in square brackets...
    \[
        ( \d+ (?: , \d+)* )
    \]

  # Return only the numbers...
    (?{ $MATCH = $CAPTURE })


<token: Cmd>
  # Match standard variants then standardize the keyword...
    (?: mv | move | rename )      (?{ $MATCH = 'mv'; })
```

`$CAPTURE` *and* `$CONTEXT` *are both aliases for the built-in read-only* `$^N` *variable, which always contains the substring matched by the nearest preceding* `(...)` *capture.* `$^N` *still works perfectly well, but these are provided to improve the readability of code blocks and error messages respectively.*

El siguiente código implementa una calculadora usando destilación en el código:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n demo_calc_inline.pl
use v5.10;
use warnings;

my $calculator = do{
    use Regexp::Grammars;
    qr{
        <Answer>

        <rule: Answer>
            <X=Mult> \+ <Y=Answer>
                (?{ $MATCH = $MATCH{X} + $MATCH{Y}; })
          | <X=Mult> - <Y=Answer>
                (?{ $MATCH = $MATCH{X} - $MATCH{Y}; })
          | <MATCH=Mult>

        <rule: Mult>
            <X=Pow> \* <Y=Mult>
                (?{ $MATCH = $MATCH{X} * $MATCH{Y}; })
          | <X=Pow>  / <Y=Mult>
                (?{ $MATCH = $MATCH{X} / $MATCH{Y}; })
          | <X=Pow>  % <Y=Mult>
                (?{ $MATCH = $MATCH{X} % $MATCH{Y}; })
          | <MATCH=Pow>

        <rule: Pow>
```

```
            <X=Term> \^ <Y=Pow>
                (?{ $MATCH = $MATCH{X} ** $MATCH{Y}; })
          | <MATCH=Term>

        <rule: Term>
               <MATCH=Literal>
          | \( <MATCH=Answer> \)

        <token: Literal>
            <MATCH=( [+-]? \d++ (?: \. \d++ )?+ )>
    }xms
};

while (my $input = <>) {
    if ($input =~ $calculator) {
        say ’--> ’, $/{Answer};
    }
}
```

Ejercicio 1.11.1. *Cual es la salida del programa anterior para las entradas:*

- `4-2-2`
- `8/4/2`
- `2^2^3`

1.11.8. Llamadas privadas a subreglas y subreglas privadas

> *If a rule name (or an alias) begins with an underscore:*
>
> ```
>  <_RULENAME>        <_ALIAS=RULENAME>
> <[_RULENAME]>      <[_ALIAS=RULENAME]>
> ```
>
> *then matching proceeds as normal, and any result that is returned is stored in the current result-hash in the usual way.*
>
> *However, when any rule finishes (and just before it returns) it first filters its result-hash, removing any entries whose keys begin with an underscore. This means that any subrule with an underscored name (or with an underscored alias) remembers its result, but only until the end of the current rule. Its results are effectively private to the current rule.*
>
> *This is especially useful in conjunction with result distillation.*

1.11.9. Mas sobre listas

Reconocimiento manual de listas

Analizando listas manualmente

El siguiente ejemplo muestra como construir un reconocedor de listas (posiblemente vacías) de números:

```
casiano@millo:~/Lregexp-grammar-examples$ cat -n simple_list.pl
     1  #!/soft/perl5lib/bin/perl5.10.1
     2  use v5.10;
     3
     4  use Regexp::Grammars;
     5
```

```
my $list = qr{
    <List>

    <rule: List>
         <digit> <List>
       | # empty

    <rule: digit>
        <MATCH=(\d+)>

}xms;

while (my $input = <>) {
    chomp $input;
    if ($input =~ $list) {
        use Data::Dumper 'Dumper';
        warn Dumper \%/;
    }
    else {
      warn "Does not match\n"
    }
}
```

Sigue una ejecución:

```
casiano@millo:~/Lregexp-grammar-examples$ ./simple_list.pl
2 3 4
$VAR1 = {
          '' => '2 3 4',
          'List' => {
                      '' => '2 3 4',
                      'digit' => '2'
                      'List' => {
                                  '' => '3 4',
                                  'digit' => '3'
                                  'List' => {
                                              '' => '4',
                                              'digit' => '4'
                                              'List' => '',
                                            },
                                },
                    }
        };
```

Influencia del orden en el lenguaje reconocido

Tenga en cuenta que el orden de las reglas influye en el lenguaje reconocido. Véase lo que ocurre si cambiamos en el ejemplo anterior el orden de las reglas:

```
casiano@millo:~/Lregexp-grammar-examples$ cat -n simple_list_empty_first.pl
#!/soft/perl5lib/bin/perl5.10.1
use v5.10;

use Regexp::Grammars;

my $list = qr{
```

```
    <List>

    <rule: List>
         # empty
       | <digit> <List>

    <rule: digit>
        <MATCH=(\d+)>

}xms;

while (my $input = <>) {
    chomp $input;
    if ($input =~ $list) {
        use Data::Dumper ’Dumper’;
        warn Dumper \%/;
    }
    else {
      warn "Does not match\n"
    }
}
```

Al ejecutar se obtiene:

```
casiano@millo:~/Lregexp-grammar-examples$ ./simple_list_empty_first.pl
2 3 4
$VAR1 = {
          ’’ => ’’,
          ’List’ => ’’
        };
```

Por supuesto basta poner anclas en el patrón a buscar para forzar a que se reconozca la lista completa:

```
pl@nereida:~/Lregexpgrammars/demo$ diff simple_list_empty_first.pl simple_list_empty_first_wit
7c7
<     <List>
---
>     ^<List>$
```

En efecto, la nueva versión reconoce la lista:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 simple_list_empty_first_with_anchors.pl
2 3 4
$VAR1 = {
          ’’ => ’2 3 4’,
          ’List’ => {
                      ’List’ => {
                                  ’List’ => {
                                              ’List’ => ’’,
                                              ’’ => ’4’,
                                              ’digit’ => ’4’
                                            },
                                  ’’ => ’3 4’,
                                  ’digit’ => ’3’
                                },
```

```
                     ’’ => ’2 3 4’,
                     ’digit’ => ’2’
                   }
        };
```

Si se quiere mantener la producción vacía en primer lugar pero forzar el reconocimiento de la lista completa, se puede hacer uso de un lookahead negativo:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n simple_list_empty_first_with_lookahead.pl
     1  #!/soft/perl5lib/bin/perl5.10.1
     2  use v5.10;
     3
     4  use strict;
     5  use Regexp::Grammars;
     6
     7  my $list = qr{
     8      <List>
     9
    10      <rule: List>
    11           (?! <digit> ) # still empty production
    12         | <digit> <List>
    13
    14      <rule: digit>
    15          <MATCH=(\d+)>
    16
    17  }xms;
    18
    19  while (my $input = <>) {
    20      chomp $input;
    21      if ($input =~ $list) {
    22          use Data::Dumper 'Dumper';
    23          warn Dumper \%/;
    24      }
    25      else {
    26        warn "Does not match\n"
    27      }
    28  }
```

Así, sólo se reducirá por la regla vacía si el siguiente token no es un número. Sigue un ejemplo de ejecución:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 simple_list_empty_first_with_lookahead.pl
2 3 4
$VAR1 = {
          ’’ => ’2 3 4’,
          ’List’ => {
                      ’List’ => {
                                  ’List’ => {
                                              ’List’ => ’’,
                                              ’’ => ’4’,
                                              ’digit’ => ’4’
                                            },
                                  ’’ => ’3 4’,
                                  ’digit’ => ’3’
                                },
```

```
                      ’’ => ’2 3 4’,
                      ’digit’ => ’2’
                    }
        };
```

Aplanamiento manual de listas

¿Cómo podemos hacer que la estructura retornada por el reconocedor sea una lista?. Podemos añadir acciones como sigue:

```
casiano@millo:~/Lregexp-grammar-examples$ cat -n simple_list_action.pl
#!/soft/perl5lib/bin/perl5.10.1
use v5.10;

use Regexp::Grammars;

my $list = qr{
    <List>

    <rule: List>
         <digit> <X=List> <MATCH= (?{ unshift @{$MATCH{X}}, $MATCH{digit}; $MATCH{X} }
       | # empty
         <MATCH= (?{ [] })>

    <rule: digit>
        <MATCH=(\d+)>

}xms;

while (my $input = <>) {
    chomp $input;
    if ($input =~ $list) {
        use Data::Dumper ’Dumper’;
        warn Dumper \%/;
    }
    else {
      warn "Does not match\n"
    }
}
```

Al ejecutarse este programa produce una salida como:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 simple_list_action.pl
2 3 4
$VAR1 = {
          ’’ => ’2 3 4’,
          ’List’ => [ ’2’, ’3’, ’4’ ]
        };
```

Los operadores de repetición

Los operadores de repetición como `*`, `+`, etc. permiten simplificar el análisis de lenguajes de listas:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n simple_list_star.pl
#!/soft/perl5lib/bin/perl5.10.1
```

```
use v5.10;

use Regexp::Grammars;

my $list = qr{
    <List>

    <rule: List>
        (?: <[digit]>)*

    <rule: digit>
        <MATCH=(\d+)>

}xms;

while (my $input = <>) {
    chomp $input;
    if ($input =~ $list) {
        use Data::Dumper 'Dumper';
        warn Dumper \%/;
    }
    else {
      warn "Does not match\n"
    }
}
```

Los corchetes alrededor de `digit` hacen que el valor asociado con el patrón sea la lista de números. Si no los ponemos el valor asociado sería el último valor de la lista.

Listas separadas por Algo

> *One of the commonest tasks in text parsing is to match a list of unspecified length, in which items are separated by a fixed token. Things like:*
>
> ```
> 1, 2, 3 , 4 ,13, 91        # Numbers separated by commas and spaces
>
> g-c-a-g-t-t-a-c-a          # Bases separated by dashes
>
> /usr/local/bin             # Names separated by directory markers
>
> /usr:/usr/local:bin        # Directories separated by colons
> ```
>
> *The usual construct required to parse these kinds of structures is either:*
>
> ```
> <rule: list>
>
>     <item> <separator> <list               # recursive definition
>   | <item>                                 # base case
> ```
>
> *Or, more efficiently, but less prettily:*
>
> ```
> <rule: list>
>
>     <[item]> (?: <separator> <[item]> )*   # iterative definition
> ```

Because this is such a common requirement, Regexp::Grammars provides a cleaner way to specify the iterative version. The syntax is taken from Perl 6:

```
<rule: list>

    <[item]> ** <separator>              # iterative definition
```

This is a repetition specifier on the first subrule (hence the use of `**` *as the marker, to reflect the repetitive behaviour of* `*`*). However, the number of repetitions is controlled by the second subrule: the first subrule will be repeatedly matched for as long as the second subrule matches immediately after it.*

So, for example, you can match a sequence of numbers separated by commas with:

```
<[number]> ** <comma>

<token: number>  \d+
<token: comma>   \s* , \s*
```

Note that it's important to use the `<[...]>` *form for the items being matched, so that all of them are saved in the result hash. You can also save all the separators (if that's important):*

```
<[number]> ** <[comma]>
```

The repeated item must be specified as a subrule call fo some kind, but the separators may be specified either as a subrule or a bracketed pattern. For example:

```
<[number]> ** ( , )
```

The separator must always be specified in matched delimiters of some kind: either matching `<...>` *or matching* `(...)`*. A common error is to write:*

```
<[number]> ** ,
```

You can also use a pattern as the item matcher, but it must be aliased into a subrule:

```
<[item=(\d+)]> ** ( , )
```

Ejemplo: Listas de números separados por comas

Veamos un ejemplo sencillo:

```
casiano@millo:~/src/perl/regexp-grammar-examples$ cat -n demo_list.pl
 1  #!/soft/perl5lib/bin/perl5.10.1
 2  use v5.10;
 3
 4  use Regexp::Grammars;
 5
 6  my $list_nonempty = qr{
 7      <List>
 8
 9      <rule: List>
10          \(  <[Value]> ** (,)  \)
11
12      <token: Value>
13          \d+
```

```
}xms;

my $list_empty = qr{
    <List>

    <rule: List>
        \(  (?: <[Value]> ** <_Sep=(,)> )?  \)

    <token: Value>
        \d+
}xms;

use Smart::Comments;


while (my $input = <>) {
    my $input2 = $input;
    if ($input =~ $list_nonempty) {
        ### nonempty: $/{List}
    }
    if ($input2 =~ $list_empty) {
        ### empty: $/{List}
    }
}
```

Sigue un ejemplo de ejecución:

```
casiano@millo:~/src/perl/regexp-grammar-examples$ ./demo_list.pl
(3,4,5)

### nonempty: {
###             '' => '(3,4,5)',
###             Value => [
###                        '3',
###                        '4',
###                        '5'
###                      ]
###           }

### empty: {
###          '' => '(3,4,5)',
###          Value => [
###                     '3',
###                     '4',
###                     '5'
###                   ]
###        }
()

### empty: '()'
```

Ejemplo: AST para las expresiones aritméticas

Las expresiones aritméticas puede definirse como una jerarquía de listas como sigue:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n calcaslist.pl
 1  use strict;
 2  use warnings;
 3  use 5.010;
 4  use Data::Dumper;
 5  $Data::Dumper::Indent = 1;
 6
 7  my $rbb = do {
 8      use Regexp::Grammars;
 9
10      qr{
11        \A<expr>\z
12
13        <objrule: expr>      <[operands=term]> ** <[operators=addop]>
14
15        <objrule: term>      <[operands=uneg]> ** <[operators=mulop]>
16
17        <objrule: uneg>      <[operators=minus]>* <[operands=power]>
18
19        <objrule: power>     <[operands=factorial]> ** <[operators=powerop]>
20
21        <objrule: factorial> <[operands=factor]>  <[operators=(!)]>*
22
23        <objrule: factor>    <val=([+-]?\d+(?:\.\d*)?)>
24                           | \( <MATCH=expr> \)
25
26        <token: addop>        [+-]
27
28        <token: mulop>        [*/]
29
30        <token: powerop>      \*\*|\^
31
32        <token: minus>        - <MATCH=(?{ ’NEG’ })>
33
34      }x;
35  };
36
37  while (my $input = <>) {
38      chomp($input);
39      if ($input =~ m{$rbb}) {
40          my $tree = $/{expr};
41          say Dumper $tree;
42
43      }
44      else {
45          say("does not match");
46      }
47  }
```

Obsérvese el árbol generado para la expresión `4-2-2`:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 calcaslist.pl
4-2-2
$VAR1 = bless( {
```

```
'operands' => [
  bless( {
    'operands' => [
      bless( {
        'operands' => [
          bless( {
            'operands' => [
              bless( {
                'operands' => [
                  bless( { '' => '4', 'val' => '4' }, 'factor' )
                ],
                '' => '4'
              }, 'factorial' )
            ],
            '' => '4'
          }, 'power' )
        ],
        '' => '4'
      }, 'uneg' )
    ],
    '' => '4'
  }, 'term' ),
  bless( {
    'operands' => [
      bless( {
        'operands' => [
          bless( {
            'operands' => [
              bless( {
                'operands' => [
                  bless( { '' => '2', 'val' => '2' }, 'factor' )
                ],
                '' => '2'
              }, 'factorial' )
            ],
            '' => '2'
          }, 'power' )
        ],
        '' => '2'
      }, 'uneg' )
    ],
    '' => '2'
  }, 'term' ),
  bless( {
    'operands' => [
      bless( {
        'operands' => [
          bless( {
            'operands' => [
              bless( {
                'operands' => [
                  bless( { '' => '2', 'val' => '2' }, 'factor' )
                ],
```

```
                  '' => '2'
                }, 'factorial' )
              ],
              '' => '2'
            }, 'power' )
          ],
          '' => '2'
        }, 'uneg' )
      ],
      '' => '2'
    }, 'term' )
  ],
  '' => '4-2-2',
  'operators' => [
    '_',
    '_'
  ]
}, 'expr' );
```

1.11.10. La directiva `require`

La directiva `require` es similar en su funcionamiento al paréntesis 5.10 `(??{ Código Perl })` el cuál hace que el `Código Perl` sea evaluado durante el tiempo de matching. El resultado de la evaluación se trata como una expresión regular con la que deberá casarse. (véase la sección 1.2.9 para mas detalles).

La sintáxis de la directiva `<require:>` es

```
<require: (?{ CODE }) >
```

> *The code block is executed and if its final value is true, matching continues from the same position. If the block's final value is false, the match fails at that point and starts backtracking.*
>
> *The* `<require:...>` *directive is useful for testing conditions that it's not easy (or even possible) to check within the syntax of the the regex itself. For example:*
>
> ```
> <rule: IPV4_Octet_Decimal>
>     # Up three digits...
>     <MATCH= ( \d{1,3}+ )>
>
>     # ...but less that 256...
>     <require: (?{ $MATCH <= 255 })>
> ```
>
> *A require expects a regex codeblock as its argument and succeeds if the final value of that codeblock is true. If the final value is false, the directive fails and the rule starts backtracking.*
>
> *Note, in this example that the digits are matched with* `\d{1,3}+` . The trailing `+` *prevents the* `{1,3}` *repetition from backtracking to a smaller number of digits if the* `<require:...>` *fails.*

El programa `demo_IP4.pl` ilustra el uso de la directiva:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n ./demo_IP4.pl
 1  #!/usr//bin/env perl5.10.1
 2  use v5.10;
 3  use warnings;
```

```

use Regexp::Grammars;

my $grammar = qr{
    \A <IP4_addr> \Z

    <token: quad>
        <MATCH=(\d{1,3})>
        <require: (?{ $MATCH < 256 })>

    <token: IP4_addr>
        <[MATCH=quad]>**(\.)
        <require: (?{ @$MATCH == 4 })>
}xms;

while (my $line = <>) {
    if ($line =~ $grammar) {
        use Data::Dumper ’Dumper’;
        say Dumper \%/;
    }
    else {
        say ’Does not match’
    }
}
```

Las condiciones usadas en el `require` obligan a que cada quad[9] sea menor que 256 y a que existan sólo cuatro quads.

Sigue un ejemplo de ejecución:

```
pl@nereida:~/Lregexpgrammars/demo$ ./demo_IP4.pl
123 . 145 . 105 . 252
Does not match
pl@nereida:~/Lregexpgrammars/demo$ ./demo_IP4.pl
123.145.105.252
$VAR1 = {
          ’’ => ’123.145.105.252’,
          ’IP4_addr’ => [
                          123,
                          145,
                          105,
                          252
                        ]
        };
pl@nereida:~/Lregexpgrammars/demo$ ./demo_IP4.pl
148.257.128.128
Does not match
0.0.0.299
Does not match
pl@nereida:~/Lregexpgrammars/demo$  ./demo_IP4.pl
123.145.105.242.193
```

---

[9] A quad (pronounced KWAHD) is a unit in a set of something that comes in four units. The term is sometimes used to describe each of the four numbers that constitute an Internet Protocol (IP) address. Thus, an Internet address in its numeric form (which is also sometimes called a dot address) consists of four quads separated by ”dots”(periods).

A quad also means *a quarter* in some usages. (A quarter as a U.S. coin or monetary unit means *a quarter of a dollar,* and in slang is sometimes called *two bits.* However, this usage does not mean two binary bits as used in computers.)

```
Does not match
```

Obsérvese como no se aceptan blancos entre los puntos en esta versión. ¿Sabría explicar la causa?

1.11.11. Casando con las claves de un hash

In some situations a grammar may need a rule that matches dozens, hundreds, or even thousands of one-word alternatives. For example, when matching command names, or valid userids, or English words. In such cases it is often impractical (and always inefficient) to list all the alternatives between `|` *alterators:*

```
<rule: shell_cmd>
    a2p | ac | apply | ar | automake | awk | ...
    # ...and 400 lines later
    ... | zdiff | zgrep | zip | zmore | zsh

<rule: valid_word>
    a | aa | aal | aalii | aam | aardvark | aardwolf | aba | ...
    # ...and 40,000 lines later...
    ... | zymotize | zymotoxic | zymurgy | zythem | zythum
```

To simplify such cases, Regexp::Grammars provides a special construct that allows you to specify all the alternatives as the keys of a normal hash. The syntax for that construct is simply to put the hash name inside angle brackets (with no space between the angles and the hash name).

Which means that the rules in the previous example could also be written:

```
<rule: shell_cmd>
    <%cmds>

<rule: valid_word>
    <%dict>
```

provided that the two hashes (`%cmds` *and* `%dict`*) are visible in the scope where the grammar is created.*

Internally, the construct is converted to something equivalent to:

```
<rule: shell_cmd>
    (<.hk>)  <require: exists $cmds{$CAPTURE}>

<rule: valid_word>
    (<.hk>)  <require: exists $dict{$CAPTURE}>
```

The special `<hk>` *rule is created automatically, and defaults to* `\S+`*, but you can also define it explicitly to handle other kinds of keys. For example:*

```
<rule: hk>
    .+            # Key may be any number of chars on a single line

<rule: hk>
    [ACGT]{10,}   # Key is a base sequence of at least 10 pairs
```

Matching a hash key in this way is typically significantly faster than matching a full set of alternations. Specifically, it is O(length of longest potential key), instead of O(number of keys).

Ejemplo de uso de la directiva hash

Sigue un ejemplo:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n hash.pl
 1  #!/usr/bin/env perl5.10.1
 2  use strict;
 3  use warnings;
 4  use 5.010;
 5  use Data::Dumper;
 6  $Data::Dumper::Deparse = 1;
 7
 8  my %cmd = map { ($_ => undef ) } qw( uname pwd date );
 9
10  my $rbb = do {
11      use Regexp::Grammars;
12
13      qr{
14        ^<command>$
15
16        <rule: command>
17          <cmd=%cmd> (?: <[arg]> )*
18
19        <token: arg> [^\s<>`&]+
20      }xms;
21  };
22
23  while (my $input = <>) {
24      chomp($input);
25      if ($input =~ m{$rbb}) {
26          say("matches: <$&>");
27          say Dumper \%/;
28          system $/{''}
29      }
30      else {
31          say("does not match");
32      }
33  }
```

Sigue un ejemplo de ejecución:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 hash.pl
a2p f1 f2
matches: <a2p f1 f2>
$VAR1 = {
          '' => 'a2p f1 f2',
          'command' => {
                         '' => 'a2p f1 f2',
                         'cmd' => 'a2p',
                         'arg' => [
                                    'f1',
                                    'f2'
                                  ]
                       }
        };
```

```
pocho 2 5
does not match
```

1.11.12. Depuración

Regexp::Grammars provides a number of features specifically designed to help debug both grammars and the data they parse.

All debugging messages are written to a log file (which, by default, is just `STDERR`*). However, you can specify a disk file explicitly by placing a* `"<logfile:...>"` *directive* at the start of your grammar[10]*:*

```
$grammar = qr{

    <logfile: LaTeX_parser_log >

    \A <LaTeX_file> \Z    # Pattern to match

    <rule: LaTeX_file>
        # etc.
}x;
```

You can also explicitly specify that messages go to the terminal:

```
<logfile: - >
```

Debugging grammar creation

Whenever a log file has been directly specified, Regexp::Grammars automatically does verbose static analysis of your grammar. That is, whenever it compiles a grammar containing an explicit `"<logfile:...>"` *directive it logs a series of messages explaining how it has interpreted the various components of that grammar. For example, the following grammar:*

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n log.pl
#!/usr/bin/env perl5.10.1
use strict;
use warnings;
use 5.010;
use Data::Dumper;

my $rbb = do {
    use Regexp::Grammars;

    qr{
      <logfile: ->

      <numbers>

      <rule: numbers>
        <number> ** <.comma>

      <token: number> \d+

```

---

[10] no funcionará si no se pone al principio de la gramática

```
      <token: comma>   ,
    }xms;
};

while (my $input = <>) {
    if ($input =~ m{$rbb}) {
        say("matches: <$&>");
        say Dumper \%/;
    }
}
```

would produce the following analysis in the terminal:

```
pl@nereida:~/Lregexpgrammars/demo$ ./log.pl
  warn | Repeated subrule <number>* will only capture its final match
       | (Did you mean <[number]>* instead?)
       |
  info | Processing the main regex before any rule definitions
       |    |
       |    |...Treating <numbers> as:
       |    |      |  match the subrule <numbers>
       |    |       \ saving the match in $MATCH{'numbers'}
       |    |
       |     \___End of main regex
       |
       | Defining a rule: <numbers>
       |    |...Returns: a hash
       |    |
       |    |...Treating <number> as:
       |    |      |  match the subrule <number>
       |    |       \ saving the match in $MATCH{'number'}
       |    |
       |    |...Treating <.comma> as:
       |    |      |  match the subrule <comma>
       |    |       \ but don't save anything
       |    |
       |    |...Treating <number> ** <.comma> as:
       |    |      |  repeatedly match the subrule <number>
       |    |       \ as long as the matches are separated by matches of <.comma>
       |    |
       |     \___End of rule definition
       |
       | Defining a rule: <number>
       |    |...Returns: a hash
       |    |
       |    |...Treating '\d' as:
       |    |       \ normal Perl regex syntax
       |    |
       |    |...Treating '+ ' as:
       |    |       \ normal Perl regex syntax
       |    |
       |     \___End of rule definition
       |
       | Defining a rule: <comma>
```

```
        |    |...Returns: a hash
        |    |
        |    |...Treating ', ' as:
        |    |       \ normal Perl regex syntax
        |    |
        |     \___End of rule definition
        |
2, 3, 4
matches: <2, 3, 4>
$VAR1 = {
          '' => '2, 3, 4',
          'numbers' => {
                         '' => '2, 3, 4',
                         'number' => '4'
                       }
        };
```

This kind of static analysis is a useful starting point in debugging a miscreant grammar[11]*, because it enables you to see what you actually specified (as opposed to what you* thought *you'd specified).*

Debugging grammar execution

Regexp::Grammars also provides a simple interactive debugger, with which you can observe the process of parsing and the data being collected in any result-hash.

To initiate debugging, place a `<debug:...>` *directive anywhere in your grammar. When parsing reaches that directive the debugger will be activated, and the command specified in the directive immediately executed. The available commands are:*

```
<debug: on>    - Enable debugging, stop when entire grammar matches
<debug: match> - Enable debugging, stope when a rule matches
<debug: try>   - Enable debugging, stope when a rule is tried
<debug: off>   - Disable debugging and continue parsing silently

<debug: continue> - Synonym for <debug: on>
<debug: run>      - Synonym for <debug: on>
<debug: step>     - Synonym for <debug: try>
```

These directives can be placed anywhere within a grammar and take effect when that point is reached in the parsing. Hence, adding a `<debug:step>` *directive is very much like setting a breakpoint at that point in the grammar. Indeed, a common debugging strategy is to turn debugging on and off only around a suspect part of the grammar:*

```
<rule: tricky>   # This is where we think the problem is...
    <debug:step>
    <preamble> <text> <postscript>
    <debug:off>
```

Once the debugger is active, it steps through the parse, reporting rules that are tried, matches and failures, backtracking and restarts, and the parser's location within both the grammar and the text being matched. That report looks like this:

---

[11] miscreant - *One who has behaved badly, or illegally; One not restrained by moral principles; an unscrupulous villain; One who holds an incorrect religious belief; an unbeliever; Lacking in conscience or moral principles; unscrupulous; Holding an incorrect religious belief.*

```
===============> Trying <grammar> from position 0
> cp file1 file2 |...Trying <cmd>
                 |   |...Trying <cmd=(cp)>
                 |   |    \FAIL <cmd=(cp)>
                 |    \FAIL <cmd>
                  \FAIL <grammar>
===============> Trying <grammar> from position 1
 cp file1 file2  |...Trying <cmd>
                 |   |...Trying <cmd=(cp)>
 file1 file2     |   |    \_____<cmd=(cp)> matched ’cp’
file1 file2      |   |...Trying <[file]>+
 file2           |   |    \_____<[file]>+ matched ’file1’
                 |   |...Trying <[file]>+
[eos]            |   |    \_____<[file]>+ matched ’ file2’
                 |   |...Trying <[file]>+
                 |   |    \FAIL <[file]>+
                 |   |...Trying <target>
                 |   |   |...Trying <file>
                 |   |   |    \FAIL <file>
                 |   |    \FAIL <target>
 <~~~~~~~~~~~~~~ |   |...Backtracking 5 chars and trying new match
file2            |   |...Trying <target>
                 |   |   |...Trying <file>
                 |   |   |    \____ <file> matched ’file2’
[eos]            |   |    \_____<target> matched ’file2’
                 |    \_____<cmd> matched ’ cp file1 file2’
                  \_____<grammar> matched ’ cp file1 file2’
```

The first column indicates the point in the input at which the parser is trying to match, as well as any backtracking or forward searching it may need to do. The remainder of the columns track the parser's hierarchical traversal of the grammar, indicating which rules are tried, which succeed, and what they match.

Provided the logfile is a terminal (as it is by default), the debugger also pauses at various points in the parsing process–before trying a rule, after a rule succeeds, or at the end of the parse–according to the most recent command issued. When it pauses, you can issue a new command by entering a single letter:

```
m       - to continue until the next subrule matches
t or s  - to continue until the next subrule is tried
r or c  - to continue to the end of the grammar
o       - to switch off debugging
```

Note that these are the first letters of the corresponding `<debug:...>` *commands, listed earlier. Just hitting ENTER while the debugger is paused repeats the previous command.*

While the debugger is paused you can also type a `d`*, which will display the result-hash for the current rule. This can be useful for detecting which rule isn't returning the data you expected.*

Veamos un ejemplo. El siguiente programa activa el depurador:

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n demo_debug.pl
     1  #!/usr/bin/env perl5.10.1
     2  use 5.010;
     3  use warnings;
     4
```

```
use Regexp::Grammars;

my $balanced_brackets = qr{
    <debug:on>

    <left_delim=(  \( )>
    (?:
        <[escape=(  \\ )]>
    |   <recurse=( (?R) )>
    |   <[simple=(  .  )]>
    )*
    <right_delim=( \) )>
}xms;

while (<>) {
    if (/$balanced_brackets/) {
        say 'matched:';
        use Data::Dumper 'Dumper';
        warn Dumper \%/;
    }
}
```

Al ejecutar obtenemos

```
pl@nereida:~/Lregexpgrammars/demo$ ./demo_debug.pl
(a)
=====> Trying <grammar> from position 0
(a)\n  |...Trying <left_delim=(  \( )>
a)\n   |    \_____<left_delim=(  \( )> matched '('     c
       |...Trying <[escape=(  \ )]>
       |    \FAIL <[escape=(  \ )]>
       |...Trying <recurse=( (?R) )>
=====> Trying <grammar> from position 1
a)\n   |   |...Trying <left_delim=(  \( )>
       |   |    \FAIL <left_delim=(  \( )>
        \FAIL <grammar>
       |...Trying <[simple=(  .  )]>
)\n    |    \_____<[simple=(  .  )]> matched 'a'
       |...Trying <[escape=(  \ )]>
       |    \FAIL <[escape=(  \ )]>
       |...Trying <recurse=( (?R) )>
=====> Trying <grammar> from position 2
)\n    |   |...Trying <left_delim=(  \( )>
       |   |    \FAIL <left_delim=(  \( )>
        \FAIL <grammar>
       |...Trying <[simple=(  .  )]>
\n     |    \_____<[simple=(  .  )]> matched ')'
       |...Trying <[escape=(  \ )]>
       |    \FAIL <[escape=(  \ )]>
       |...Trying <recurse=( (?R) )>
=====> Trying <grammar> from position 3
\n     |   |...Trying <left_delim=(  \( )>
       |   |    \FAIL <left_delim=(  \( )>
        \FAIL <grammar>
```

```
       |...Trying <[simple=(  .  )]>
[eos]  |    \_____<[simple=(  .  )]> matched ''
       |...Trying <[escape=(  \ )]>
       |    \FAIL <[escape=(  \ )]>
       |...Trying <recurse=( (?R) )>
=====> Trying <grammar> from position 4
[eos]  |   |...Trying <left_delim=(  \( )>
       |   |    \FAIL <left_delim=(  \( )>
        \FAIL <grammar>
       |...Trying <[simple=(  .  )]>
       |    \FAIL <[simple=(  .  )]>
       |...Trying <right_delim=( \) )>
       |    \FAIL <right_delim=( \) )>
 <~~~~ |...Backtracking 1 char and trying new match
\n     |...Trying <right_delim=( \) )>
       |    \FAIL <right_delim=( \) )>
 <~~~~ |...Backtracking 1 char and trying new match
)\n    |...Trying <right_delim=( \) )>
\n     |    \_____<right_delim=( \) )> matched ')'
        \_____<grammar> matched '(a)'   d
             :         {
             :           '' => '(a)',
             :           'left_delim' => '(',
             :           'simple' => [
             :                         'a'
             :                       ],
             :           'right_delim' => ')'
             :         };      o
matched:
$VAR1 = {
          '' => '(a)',
          'left_delim' => '(',
          'simple' => [
                        'a'
                      ],
          'right_delim' => ')'
        };
```

1.11.13. Mensajes de `log` del usuario

Both static and interactive debugging send a series of predefined log messages to whatever log file you have specified. It is also possible to send additional, user-defined messages to the log, using the `"<log:...>"` *directive.*

This directive expects either a simple text or a codeblock as its single argument. If the argument is a code block, that code is expected to return the text of the message; if the argument is anything else, that something else is *the literal message. For example:*

```
<rule: ListElem>

    <Elem=   ( [a-z]\d+) >
        <log: Checking for a suffix, too...>

    <Suffix= ( : \d+   ) >?
        <log: (?{ "ListElem: $MATCH{Elem} and $MATCH{Suffix}" })>
```

User-defined log messages implemented using a codeblock can also specify a severity level. If the codeblock of a `<log:...>` *directive returns two or more values, the first is treated as a log message severity indicator, and the remaining values as separate lines of text to be logged. For example:*

```
<rule: ListElem>
    <Elem=   ( [a-z]\d+) >
    <Suffix= ( : \d+   ) >?

        <log: (?{
            warn => "Elem was: $MATCH{Elem}",
                    "Suffix was $MATCH{Suffix}",
        })>
```

When they are encountered, user-defined log messages are interspersed between any automatic log messages (i.e. from the debugger), at the correct level of nesting for the current rule.

1.11.14. Depuración de Regexps

It is possible to use Regexp::Grammars without creating any *subrule definitions, simply to debug a recalcitrant regex. For example, if the following regex wasn't working as expected:*

```
my $balanced_brackets = qr{
    \(             # left delim
    (?:
        \\         # escape or
    |   (?R)       # recurse or
    |   .          # whatever
    )*
    \)             # right delim
}xms;
```

you could instrument it with aliased subpatterns and then debug it step-by-step, using Regexp::Grammars:

```
use Regexp::Grammars;

my $balanced_brackets = qr{
    <debug:step>

    <.left_delim=  (  \(  )>
    (?:
        <.escape=  (  \\  )>
    |   <.recurse= ( (?R) )>
    |   <.whatever=(  .   )>
    )*
    <.right_delim= (  \)  )>
}xms;

while (<>) {
    say 'matched' if /$balanced_brackets/;
}
```

Note the use of amnesiac aliased subpatterns to avoid needlessly building a result-hash. Alternatively, you could use listifying aliases to preserve the matching structure as an additional debugging aid:

```
use Regexp::Grammars;

my $balanced_brackets = qr{
    <debug:step>

    <[left_delim=  (  \(  )]>
    (?:
        <[escape=  (  \\  )]>
    |   <[recurse= ( (?R) )]>
    |   <[whatever=(  .   )]>
    )*
    <[right_delim= (  \)  )]>
}xms;

if ( ’(a(bc)d)’ =˜ /$balanced_brackets/) {
    use Data::Dumper ’Dumper’;
    warn Dumper \%/;
}
```

1.11.15. Manejo y recuperación de errores

En este punto debo decir que no he podido reproducir el comportamiento de las directivas `<error:>` y `<warning:>` tal y como las describe Conway en el manual de Regexp::Grammars.

El siguiente ejemplo ilustra un conjunto de técnicas de gestión de errores que son independientes del soprote dado por Regexp::Grammars.

Se trata de la misma calculadora explicada en la sección 1.11.18.

```
pl@nereida:˜/Lregexpgrammars/demo/calculator$ cat -n calculatorwitherrmanagement.pl
#!/usr/bin/env perl5.10.1
use strict;
use warnings;
use 5.010;
use Lingua::EN::Inflect qw(PL);
use Scalar::Util qw{blessed};

my $rbb = do {
    my ($warnings, $errors);    # closure
    sub warnings { $warnings }  # accessor
    sub errors { $errors }      # accessor

    use Regexp::Grammars;
    qr{
      (?{
          $warnings = 0;
          $errors = 0;
      })
      \A<expr>
      (?:    \z
           |
             (.*) (?{
```

```
                      # Accept the string but emit a warning
                      $warnings++;
                      local our $expr = \$MATCH{expr}{''};
                      local our $endlegal = length($$expr) > 4? "... ".substr($$expr, -4)
                      warn "Warning: Unexpected '". substr($^N, 0, 10)."' after '$endlegal
                   })
      )

      <objrule: expr>      <[operands=term]> ** <[operators=addop]>

      <objrule: term>      <[operands=uneg]> ** <[operators=mulop]>

      <objrule: uneg>      <[operators=minus]>* <[operands=power]>

      <objrule: power>     <[operands=factorial]> ** <[operators=powerop]>

      <objrule: factorial> <[operands=factor]>  <[operators=(!)]>*

      <objrule: factor>    (<val=([+-]?\d+(?:\.\d*)?)>)
                         | \( <MATCH=expr> \)
                         | ([^-+(0-9]+) (?{
                                          # is + and not * to avoid infinite recursion
                                          warn "Error: expecting a number or a open parent
                                          $warnings++;
                                          $errors++;
                                      }) <MATCH=factor>

      <token: addop>        [+-]

      <token: mulop>        [*/]

      <token: powerop>      \*\*|\^

      <token: minus>        - <MATCH=(?{ 'NEG' })>

    }x;
};

sub test_calc {
  my $prompt = shift;

  print $prompt;
  while (my $input = <>) {
      chomp($input);

      local %/;
      $input =~ m{$rbb};

      say warnings." ".PL('warning',warnings) if warnings;
      say errors." ".PL('error',errors)       if errors;

      my $tree = $/{expr};
      if (blessed($tree)) {
```

```
          do "PostfixCalc.pm";
          say "postfix: ".$tree->ceval;

          do "EvalCalc.pm";
          say "result: ".$tree->ceval;
      }
      print $prompt;
  }
  say "Bye!"
}

########## main
test_calc(
  'Parsing infix arithmetic expressions (CTRL-D to end in unix) ',
);
```

Veamos algunas ejecuciones que incluyen entradas erróneas:

```
pl@nereida:~/Lregexpgrammars/demo/calculator$ ./calculatorwitherrmanagement.pl
Parsing infix arithmetic expressions (CTRL-D to end in unix) 2+3
postfix: 2 3 +
result: 5
Parsing infix arithmetic expressions (CTRL-D to end in unix) 2*(3+#)
Error: expecting a number or a open parenthesis, found: '#)'
Error: expecting a number or a open parenthesis, found: '#'
Error: expecting a number or a open parenthesis, found: ')'
Warning: Unexpected '*(3+#)' after '2'
4 warnings
3 errors
postfix: 2
result: 2
Parsing infix arithmetic expressions (CTRL-D to end in unix) 2+#*4
Error: expecting a number or a open parenthesis, found: '#*'
1 warning
1 error
postfix: 2 4 +
result: 6
Parsing infix arithmetic expressions (CTRL-D to end in unix) Bye!
```

Obsérvese los mensajes de error repetidos para la entrada `2*(3+#)`. Ellos son debidos a los reiterados intentos de casar `<factor>` en la regla de recuperación de errores:

```
      <objrule: factor>    (<val=([+-]?\d+(?:\.\d*)?)>)
                         | \( <MATCH=expr> \)
                         | ([^-+(0-9]+) (?{
                                          # is + and not * to avoid infinite recursion
                                          warn "Error: expecting a number or a open parent
                                          $warnings++;
                                          $errors++;
                                      }) <MATCH=factor>
```

en este caso resulta imposible encontrar un factor. Se puede cambiar la conducta indicando un `(* COMMIT)` antes de la llamada a `<MATCH=factor>`:

```
      <objrule: factor>    (<val=([+-]?\d+(?:\.\d*)?)>)
                         | \( <MATCH=expr> \)
```

```
                        | ([^-+(0-9]+) (?{
                                         # is + and not * to avoid infinite recursion
                                         warn "Error: expecting a number or a open parent
                                         $warnings++;
                                         $errors++;
                                     }) (*COMMIT) <MATCH=factor>
```

en este caso la conducta es abandonar en el caso de que no se pueda encontrar un `<factor>`:

```
pl@nereida:~/Lregexpgrammars/demo/calculator$ ./calculatorwitherrmanagement.pl
Parsing infix arithmetic expressions (CTRL-D to end in unix) 2*(3+#)
Error: expecting a number or a open parenthesis, found: '#)'
1 warning
1 error
Parsing infix arithmetic expressions (CTRL-D to end in unix) 2*3
postfix: 2 3 *
result: 6
Parsing infix arithmetic expressions (CTRL-D to end in unix) @
Error: expecting a number or a open parenthesis, found: '@'
1 warning
1 error
Parsing infix arithmetic expressions (CTRL-D to end in unix) Bye!
```

1.11.16. Mensajes de Warning

> *Sometimes, you want to detect problems, but not invalidate the entire parse as a result. For those occasions, the module provides a* less stringent *form of error reporting: the* `<warning:...>` *directive.*
>
> *This directive is exactly the same as an* `<error:...>` *in every respect except that it does not induce a failure to match at the point it appears.*
>
> *The directive is, therefore, useful for reporting* non-fatal *problems in a parse. For example:*
>
> ```
>     qr{ \A            # ...Match only at start of input
>         <ArithExpr>   # ...Match a valid arithmetic expression
>
>         (?:
>             # Should be at end of input...
>             \s* \Z
>           |
>             # If not, report the fact but don't fail...
>             <warning: Expected end-of-input>
>             <warning: (?{ "Extra junk at index $INDEX: $CONTEXT" })>
>         )
>
>         # Rule definitions here...
>     }xms;
> ```
>
> *Note that, because they do not induce failure, two or more* `<warning:...>` *directives can be "stacked"in sequence, as in the previous example.*

1.11.17. Simplificando el AST

```
pl@nereida:~/Lregexpgrammars/demo$ cat -n exprdamian.pl
     1  use strict;
```

```
use warnings;
use 5.010;
use Data::Dumper;
$Data::Dumper::Indent = 1;

my $rbb = do {
    use Regexp::Grammars;

    qr{
      \A<expr>\z

      <objrule: expr>    <MATCH=term> (?! <addop> )                  # bypass
                       | <[operands=term]> ** <[operators=addop]>

      <objrule: term>    <MATCH=factor> (?! <mulop> )                # bypass
                       | <[operands=factor]> ** <[operators=mulop]>

      <objrule: factor>    <val=([+-]?\d+(?:\.\d*)?)>
                       | \( <MATCH=expr> \)

      <token: addop> [+-]

      <token: mulop> [*/]

    }x;
};

while (my $input = <>) {
    chomp($input);
    if ($input =~ m{$rbb}) {
        my $tree = $/{expr};
        say Dumper $tree;
        say $tree->ceval;

    }
    else {
        say("does not match");
    }
}

BEGIN {

  package LeftBinaryOp;
  use strict;
  use base qw(Class::Accessor);

  LeftBinaryOp->mk_accessors(qw{operators operands});

  my %f = (
    '+' => sub { shift() + shift() },
    '-' => sub { shift() - shift() },
    '*' => sub { shift() * shift() },
    '/' => sub { shift() / shift() },
```

```
  );

  sub ceval {
    my $self = shift;

    # recursively evaluate the children first
    my @operands = map { $_->ceval } @{$self->operands};

    # then combine them
    my $s = shift @operands;
    for (@{$self->operators}) {
      $s = $f{$_}->($s, shift @operands);
    }
    return $s;
  }

  package term;
  use base qw{LeftBinaryOp};

  package expr;
  use base qw{LeftBinaryOp};

  package factor;

  sub ceval {
    my $self = shift;

    return $self->{val};
  }

  1;
}
```

Ejecuciones:

```
pl@nereida:~/Lregexpgrammars/demo$ perl5.10.1 exprdamian.pl
4-2-2
$VAR1 = bless( {
  'operands' => [
    bless( {
      '' => '4',
      'val' => '4'
    }, 'factor' ),
    bless( {
      '' => '2',
      'val' => '2'
    }, 'factor' ),
    bless( {
      '' => '2',
      'val' => '2'
    }, 'factor' )
  ],
  '' => '4-2-2',
  'operators' => [
```

```
    '-',
    '-'
  ]
}, 'expr' );

0
8/4/2
$VAR1 = bless( {
  'operands' => [
    bless( {
      '' => '8',
      'val' => '8'
    }, 'factor' ),
    bless( {
      '' => '4',
      'val' => '4'
    }, 'factor' ),
    bless( {
      '' => '2',
      'val' => '2'
    }, 'factor' )
  ],
  '' => '8/4/2',
  'operators' => [
    '/',
    '/'
  ]
}, 'term' );

1
3
$VAR1 = bless( {
  '' => '3',
  'val' => '3'
}, 'factor' );

3
2*(3+4)
$VAR1 = bless( {
  'operands' => [
    bless( {
      '' => '2',
      'val' => '2'
    }, 'factor' ),
    bless( {
      'operands' => [
        bless( {
          '' => '3',
          'val' => '3'
        }, 'factor' ),
        bless( {
          '' => '4',
          'val' => '4'
```

```
        }, 'factor' )
      ],
      '' => '3+4',
      'operators' => [
        '+'
      ]
    }, 'expr' )
  ],
  '' => '2*(3+4)',
  'operators' => [
    '*'
  ]
}, 'term' );

14
```

1.11.18. Reciclando una `Regexp::Grammar`

Ejecución

El siguiente programa `calculator.pl` recibe como entrada una expresión en infijo.

La ejecución consta de dos bucles. En la primera parte se inyecta a la jerarquía de clases de los AST generados para las expresiones en infijo una semántica que permite evaluar la expresión:

```
require EvalCalc;

test_calc(
  'Evaluating infix arithmetic expressions (CTRL-D to end in unix) ',
  sub { print &Data::Dumper::Dumper(shift()) },
);
```

En esta primera parte mostraremos además el AST construido para la expresión infija de entrada.

```
pl@nereida:~/Lregexpgrammars/demo$ ./calculator.pl
Evaluating infix arithmetic expressions (CTRL-D to end in unix)
8-4-2
$VAR1 = bless( {
  'operands' => [
    bless( {
      'operands' => [
        bless( {
          'operands' => [
            bless( {
              'operands' => [
                bless( {
                  'operands' => [
                    bless( { '' => '8', 'val' => '8' }, 'factor' )
                  ],
                  '' => '8'
                }, 'factorial' )
              ],
              '' => '8'
            }, 'power' )
          ],
          '' => '8'
        }, 'uneg' )
```

```
      ],
      ’’ => ’8’
    }, ’term’ ),
    bless( {
      ’operands’ => [
        bless( {
          ’operands’ => [
            bless( {
              ’operands’ => [
                bless( {
                  ’operands’ => [
                    bless( { ’’ => ’4’, ’val’ => ’4’ }, ’factor’ )
                  ],
                  ’’ => ’4’
                }, ’factorial’ )
              ],
              ’’ => ’4’
            }, ’power’ )
          ],
          ’’ => ’4’
        }, ’uneg’ )
      ],
      ’’ => ’4’
    }, ’term’ ),
    bless( {
      ’operands’ => [
        bless( {
          ’operands’ => [
            bless( {
              ’operands’ => [
                bless( {
                  ’operands’ => [
                    bless( { ’’ => ’2’, ’val’ => ’2’ }, ’factor’ )
                  ],
                  ’’ => ’2’
                }, ’factorial’ )
              ],
              ’’ => ’2’
            }, ’power’ )
          ],
          ’’ => ’2’
        }, ’uneg’ )
      ],
      ’’ => ’2’
    }, ’term’ )
  ],
  ’’ => ’8-4-2’,
  ’operators’ => [
    ’-’,
    ’-’
  ]
}, ’expr’ );
2
```

Observamos que la asociatividad es la correcta. El 2 final es el resultado de la evaluación de `8-4-2`.

La estructura del árbol se corresponde con la de la gramática:

```
my $rbb = do {
    use Regexp::Grammars;

    qr{
      \A<expr>\z

      <objrule: expr>      <[operands=term]> ** <[operators=addop]>

      <objrule: term>      <[operands=uneg]> ** <[operators=mulop]>

      <objrule: uneg>      <[operators=minus]>* <[operands=power]>

      <objrule: power>     <[operands=factorial]> ** <[operators=powerop]>

      <objrule: factorial> <[operands=factor]>  <[operators=(!)]>*

      <objrule: factor>    <val=([+-]?\d+(?:\.\d*)?)>
                         | \( <MATCH=expr> \)

      <token: addop>        [+-]

      <token: mulop>        [*/]

      <token: powerop>      \*\*|\^

      <token: minus>        - <MATCH=(?{ ’NEG’ })>

    }x;
};
```

Ahora, en una segunda parte sobreescribimos los métodos `sem` que describen la semántica para producir una traducción de infijo a postfijo:

```
require PostfixCalc;
test_calc(’Translating expressions to postfix (CTRL-D to end in unix) ’);
```

Ahora al proporcionar la entrada `6--3!` obtenemos:

```
Translating expressions to postfix (CTRL-D to end in unix)
6--3!
6 3 ! ~ -
```

Aquí `~` es el operador de negación unaria y `!` es el operador factorial.

Estructura de la aplicación

Estos son los ficheros que integran la aplicación:

```
pl@nereida:~/Lregexpgrammars/demo/calculator$ tree
.
|-- EvalCalc.pm            # Soporte para la evaluación de la expresión: sem
|-- Operator.pm            # Soporte a las clases nodo: recorridos
|-- PostfixCalc.pm         # Soporte para la traducción a postfijo: sem
`-- calculator.pl          # programa principal
```

Programa principal

En el programa principal definimos la gramática y escribimos una subrutina `test_calc` que realiza el parsing.

```
pl@nereida:~/Lregexpgrammars/demo/calculator$ cat -n calculator.pl
#!/usr/bin/env perl5.10.1
use strict;
use warnings;
use 5.010;
use Data::Dumper;
$Data::Dumper::Indent = 1;

my $rbb = do {
    use Regexp::Grammars;

    qr{
      \A<expr>\z

      <objrule: expr>      <[operands=term]> ** <[operators=addop]>

      <objrule: term>      <[operands=uneg]> ** <[operators=mulop]>

      <objrule: uneg>      <[operators=minus]>* <[operands=power]>

      <objrule: power>     <[operands=factorial]> ** <[operators=powerop]>

      <objrule: factorial> <[operands=factor]>  <[operators=(!)]>*

      <objrule: factor>    <val=([+-]?\d+(?:\.\d*)?)>
                         | \( <MATCH=expr> \)

      <token: addop>        [+-]

      <token: mulop>        [*/]

      <token: powerop>      \*\*|\^

      <token: minus>        - <MATCH=(?{ 'NEG' })>

    }x;
};

sub test_calc {
  my $prompt = shift;
  my $handler = shift;

  say $prompt;
  while (my $input = <>) {
      chomp($input);
      if ($input =~ m{$rbb}) {
          my $tree = $/{expr};
          $handler->($tree) if $handler;

          say $tree->ceval;
```

```

      }
      else {
          say("does not match");
      }
  }
}

require EvalCalc;

test_calc(
  'Evaluating infix arithmetic expressions (CTRL-D to end in unix) ',
  sub { print &Data::Dumper::Dumper(shift()) },
);


require PostfixCalc;
test_calc('Translating expressions to postfix (CTRL-D to end in unix) ');
```

Los nodos del AST poseen un método `ceval` que se encarga de realizar la traducción del nodo.

Las Clases de nodos del AST

```
pl@nereida:~/Lregexpgrammars/demo/calculator$ cat -n Operator.pm
#   Class hierarchy diagram:
#  $ vgg -t 'Operator(LeftBinaryOp(expr,term),RightBinaryOp(power),PreUnaryOp(uneg),Post
#                           +--------+
#                           |Operator|
#                           +--------+
#          .---------------.----^--------.-------------.
#    +------------+ +-------------+ +----------+ +-----------+
#    |LeftBinaryOp| |RightBinaryOp| |PreUnaryOp| |PostUnaryOp|
#    +------------+ +-------------+ +----------+ +-----------+
#      .---^--.        |              |               |
#    +-----+ +-----+  +------+        +-----+      +----------+
#    |expr| |term|  |power|         |uneg|       |factorial|
#    +-----+ +-----+  +------+        +-----+      +----------+
#
#
# NOTE: package "factor" actually implements numbers and is
#       outside this hierarchy
#
package Operator;
use strict;
use Carp;

sub Operands {
  my $self = shift;

  return () unless exists $self->{operands};
  return @{$self->{operands}};
}

sub Operators {
```

```
  my $self = shift;

  return () unless exists $self->{operators};
  return @{$self->{operators}};
}

sub sem {
  confess "not defined sem";
}

sub make_sem {
  my $class = shift;
  my %semdesc = @_;

  for my $class (keys %semdesc) {
    my %sem = %{$semdesc{$class}};

    # Install ’sem’ method in $class
    no strict ’refs’;
    no warnings ’redefine’;
    *{$class."::sem"} = sub {
      my ($self, $op) = @_;
      $sem{$op}
    };
  }
}

package LeftBinaryOp;
use base qw{Operator};

sub ceval {
  my $self = shift;

  # recursively evaluate the children first
  my @operands = map { $_->ceval } $self->Operands;

  # then combine them
  my $s = shift @operands;
  for ($self->Operators) {
    $s = $self->sem($_)->($s, shift @operands);
  }
  return $s;
}

package RightBinaryOp;
use base qw{Operator};

sub ceval {
  my $self = shift;

  # recursively evaluate the children first
  my @operands = map { $_->ceval } $self->Operands;

```

```
  # then combine them
  my $s = pop @operands;
  for (reverse $self->Operators) {
    $s = $self->sem($_)->(pop @operands, $s);
  }
  return $s;
}

package PreUnaryOp;
use base qw{Operator};

sub ceval {
  my $self = shift;

  # recursively evaluate the children first
  my @operands = map { $_->ceval } $self->Operands;

  # then combine them
  my $s = shift @operands;
  for (reverse $self->Operators) {
    $s = $self->sem($_)->($s);
  }
  return $s;
}

package PostUnaryOp;
use base qw{Operator};

sub ceval {
  my $self = shift;

  # recursively evaluate the children first
  my @operands = map { $_->ceval } $self->Operands;

  # then combine them
  my $s = shift @operands;
  for ($self->Operators) {
    $s = $self->sem($_)->($s);
  }
  return $s;
}

package term;
use base qw{LeftBinaryOp};

package expr;
use base qw{LeftBinaryOp};

package power;
use base qw{RightBinaryOp};

package uneg;
use base qw{PreUnaryOp};
```

```
137
138  package factorial;
139  use base qw{PostUnaryOp};
140
141  package factor;
142
143  sub ceval {
144    my $self = shift;
145
146    return $self->{val};
147  }
148
149  1;
```

Definiendo `sem` para la evaluación de la expresión

```
pl@nereida:~/Lregexpgrammars/demo/calculator$ cat -n EvalCalc.pm
 1  package EvalCalc;
 2  use strict;
 3  use Carp;
 4
 5  use Operator;
 6
 7  ####
 8  sub f {
 9    $_[0]>1?$_[0]*f($_[0]-1):1;
10  }
11
12  sub fac {
13    my $n = shift;
14
15    confess "Not valid number" unless $n =~ /^\d+$/;
16    f($n);
17  };
18
19  my $s = sub { shift() **  shift() };
20
21  Operator->make_sem(
22     expr => {
23        ’+’ => sub { shift()  +  shift() },
24        ’-’ => sub { shift()  -  shift() },
25     },
26     term => {
27       ’*’ => sub { shift()  *  shift() },
28       ’/’ => sub { shift()  /  shift() },
29     },
30     power => {
31        ’^’  => $s,
32        ’**’ => $s,
33     },
34     uneg => {
35        ’NEG’ => sub { -shift() },
36     },
37     factorial => {
```

```
      '!' => \&fac,
   },
);

1;
```

Definiendo `sem` para la traducción a postfijo

```
pl@nereida:~/Lregexpgrammars/demo/calculator$ cat -n PostfixCalc.pm
package PostfixCalc;
use strict;

use Operator;

# Modify semantics: now translate to postfix
my $powers = sub { shift().' '.shift().' **' };

Operator->make_sem(
   expr => {
      '+' => sub { shift().' '.shift().' +'  },
      '-' => sub { shift().' '.shift().' -' },
    },
    term => {
      '*' => sub { shift().' '.shift().' *'  },
      '/' => sub { shift().' '.shift().' /' },
    },
    power => {
      '^'  => $powers,
      '**' => $powers,
    },
    uneg => {
       # use ~ for unary minus
      'NEG' => sub { shift().' ~' },
    },
    factorial => {
      '!' => sub { shift().' !'},
    },
);

1;
```

Ejercicio 1.11.2.

- *Explique el significado de la primera línea del programa principal*

  ```
  pl@nereida:~/Lregexpgrammars/demo$ cat -n calculator.pl
  #!/usr/bin/env perl5.10.1
  ```

- *Explique el significado de* `$handler` *en* `test_calc`*:*

  ```
  sub test_calc {
    my $prompt = shift;
    my $handler = shift;

    say $prompt;
    while (my $input = <>) {
  ```

```
      chomp($input);
      if ($input =~ m{$rbb}) {
          my $tree = $/{expr};
          $handler->($tree) if $handler;

          say $tree->ceval;

      }
      else {
          say("does not match");
      }
  }
}
```

- *Aisle las funciones relacionadas con la creación de semántica como* `make_sem`*,* `fac` *y las llamadas a* `make_sem` *en un módulo* `Calculator::Semantics` *aparte.*
- *Añada un traductor de infijo a prefijo al código presentado en esta sección. Una expresión como* `2*3+4` *se traducirá como* `+ * 2 3 4`

1.11.19. Práctica: Calculadora con `Regexp::Grammars`

- Reforme la estructura del ejemplo para que tenga una jerarquía de desarrollo de acuerdo a los estándares de Perl. Use `h2xs` o bien Module::Starter. Use el espacio de nombres `Calculator`. Mueva el módulo `Operator` a `Calculator::Operator`. Lea el capítulo Modulos de los apuntes de LHP.
- Defina el conjunto de pruebas que deberá pasar su traductor. Añádalas como pruebas `TODO`. Cuando la funcionalidad a comprobar esté operativa cambie su estatus.
- Añada variables y la expresión de asignación:

  ```
  b = a = 4*2
  ```

  que será traducida a postfijo como:

  ```
  4 2 * a = b =
  ```

  El operador de asignación es asociativo a derechas. El valor devuelto por una expresión de asignación es el valor asignado.

  Use un hash para implantar la relación nombre-valor en el caso de la evaluación
- Introduzca la expresión bloque:

  ```
  c = { a = 4; b = 2*a }
  ```

  Los bloques son listas entre llaves de expresiones separadas por punto y coma. El valor retornado por una expresión bloque es el último evaluado en el bloque.

  El símbolo de arranque de la gramática (esto es, el patrón regular contra el que hay que casar) será la expresión bloque.
- Introduzca las expresiones de comparación `<`, `>`, `<=`, `>=`, `==` y `!=` con la prioridad adecuada. Tenga en cuenta que una expresión como:

```
a = b+2 > c*4
```

deberá entenderse como

```
a = ((b+2) > (c*4))
```

Esto es, se traducirá como:

```
b 2 + c 4 * > a =
```

- Introduzca la expresión `if ... then ... else`. La parte del `else` será opcional:

  ```
  c = if a > 0 then { a = a -1; 2*a } else { b + 2 };
  d = if a > 0 then { a = b -1; 2*b };
  ```

  un `else` casa con el `if` mas cercano. La sentencia:

  ```
  if (a > 0) then if (b > 0) then {5} else {6}
  ```

  se interpreta como:

  ```
  if (a > 0) then (if (b > 0) then {5} else {6})
  ```

  y no como:

  ```
  if (a > 0) then (if (b > 0) then {5}) else {6}
  ```

  Se traducirá como:

  ```
          a
          0
          >
          jz endif124
          b
          0
          >
          jz else125
          5
          j endif126
  :else125
          6
  :endif124
  :endif125
          ...
  ```

- Escriba un intérprete de la máquina orientada a pila definida en los apartados anteriores. El código generado debería poder ejecutarse correctamente en el intérprete.

Capítulo 2

La Estructura de los Compiladores: Una Introducción

Este capítulo tiene por objeto darte una visión global de la estructura de un compilador e introducirte en las técnicas básicas de la construcción de compiladores usando Perl.

Puesto que no todos los alumnos que se incorporan en este capítulo han leído los anteriores y no necesariamente conocen Perl, en la sección 2.1 comenzamos haciendo un breve repaso a como construir un módulo en Perl. Si quieres tener un conocimiento mas profundo lee el capítulo sobre módulos en [2].

La sección 2.2 describe las fases en las que -al menos conceptualmente- se divide un compilador. A continuación la sección 2.3 presenta la primera de dichas fases, el análisis léxico. En la sección 2.5 repasamos conceptos de análisis sintáctico que deberían ser familiares a cualquiera que haya seguido un curso en teoría de autómatas y lenguajes formales. Antes de comenzar a traducir es conveniente tener un *esquema* o estrategia de traducción para cada constructo sintáctico. La sección 2.7 introduce el concepto de esquema de traducción. La fase de análisis sintáctico consiste en la construcción del *árbol de análisis* a partir de la *secuencia de unidades léxicas*. Existen diversas estrategias para resolver esta fase. En la sección 2.6 introducimos la que posiblemente sea la mas sencilla de todas: el análisis descendente predictivo recursivo. En la sección 2.8 abordamos una estrategia para transformar ciertas gramáticas para las que dicho método no funciona.

Un analizador sintáctico implícitamente construye el árbol de análisis concreto. En muchas ocasiones resulta mas rentable trabajar con una forma simplificada (*abstracta*) del árbol que contiene la misma información que aquél. La sección 2.9 trata de la construcción de los árboles de análisis abstractos.

2.1. Las Bases

Puesto que no todos los alumnos que están interesados en esta sección tienen conocimientos previos de Perl, en esta sección comenzamos haciendo un breve repaso a como construir un módulo en Perl y al mismo tiempo repasamos las características usadas del lenguaje. Si quieres tener un conocimiento mas profundo de como construir un módulo, lee el capítulo sobre módulos en [2].

Version

El comportamiento de Perl puede variar ligeramente si la versión que tenemos instalada es antigua. Para ver la versión de Perl podemos hacer.

```
lhp@nereida:~/Lperl/src/topdown/PL0506$ perl -v

This is perl, v5.8.4 built for i386-linux-thread-multi

Copyright 1987-2004, Larry Wall
```

```
Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using ‘man perl’ or ‘perldoc perl’.  If you have access to the
Internet, point your browser at http://www.perl.com/, the Perl Home Page.
```

h2xs

En primer lugar, construimos la estructura para nuestro proyecto de mini-lenguaje. La mejor forma de comenzar a escribir un módulo es usando la herramienta Perl `h2xs`. Supongamos que queremos construir un módulo `PL::Tutu`. Los nombres de los módulos siguen un esquema de identificadores separados por una pareja de `:`. Para saber más sobre el esquema de nombres de los módulos y la forma en la que estos se asigna a ficheros del sistema operativo, lea la sección sobre introducción a los módulos [2].

```
lhp@nereida:~/Lperl/src/topdown/PL0506$ h2xs -XA -n PL::Tutu
Defaulting to backwards compatibility with perl 5.8.4
If you intend this module to be compatible with earlier perl versions, please
specify a minimum perl version with the -b option.

Writing PL-Tutu/lib/PL/Tutu.pm
Writing PL-Tutu/Makefile.PL
Writing PL-Tutu/README
Writing PL-Tutu/t/PL-Tutu.t
Writing PL-Tutu/Changes
Writing PL-Tutu/MANIFEST
```

La herramienta `h2xs` fué concebida para ayudar en la transformación de ficheros de cabecera de C en código Perl. La opción `-X` hace que se omita la creación de subrutinas externas (`XS`) La opción `-A` implica que el módulo no hará uso del `AutoLoader`. La opción `-n` proporciona el nombre del módulo. La llamada a `h2xs` crea la siguiente estructura de directorios y ficheros:

```
lhp@nereida:~/Lperl/src/topdown/PL0506$ tree
.
‘-- PL-Tutu
    |-- Changes
    |-- MANIFEST
    |-- Makefile.PL
    |-- README
    |-- lib
    |   ‘-- PL
    |       ‘-- Tutu.pm
    ‘-- t
        ‘-- PL-Tutu.t

4 directories, 6 files
```

Generación del Makefile

Después de esto tenemos un módulo ”funcional” que no hace nada. Lo podríamos instalar como si lo hubieramos descargado desde CPAN (Véase [2]).

Después cambiamos al directorio `PL-Tutu/` y hacemos `perl Makefile.PL`.

```
lhp@nereida:~/Lperl/src/topdown/PL0506$ cd PL-Tutu/
lhp@nereida:~/Lperl/src/topdown/PL0506/PL-Tutu$ perl Makefile.PL
```

```
Checking if your kit is complete...
Looks good
Writing Makefile for PL::Tutu
```

Esto crea el fichero `Makefile` necesario para actualizar nuestra aplicación. Para saber más sobre `perl Makefile.PL` lea [2].

Documentación

Pasamos ahora a trabajar en el módulo. Primero escribimos la parte relativa a la documentación. Para ello editamos `Tutu.pm`:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/PL-Tutu/lib/PL$ pwd
/home/lhp/Lperl/src/topdown/PL0506/PL-Tutu/lib/PL
lhp@nereida:~/Lperl/src/topdown/PL0506/PL-Tutu/lib/PL$ ls -l
total 4
-rw-r--r--  1 lhp lhp 2343 2005-09-28 11:16 Tutu.pm
```

y al final del mismo insertamos la documentación. Para saber más sobre el lenguajes de marcas de Perl (*pod* por *plain old documentation*) lea [2]. En este caso escribimos:

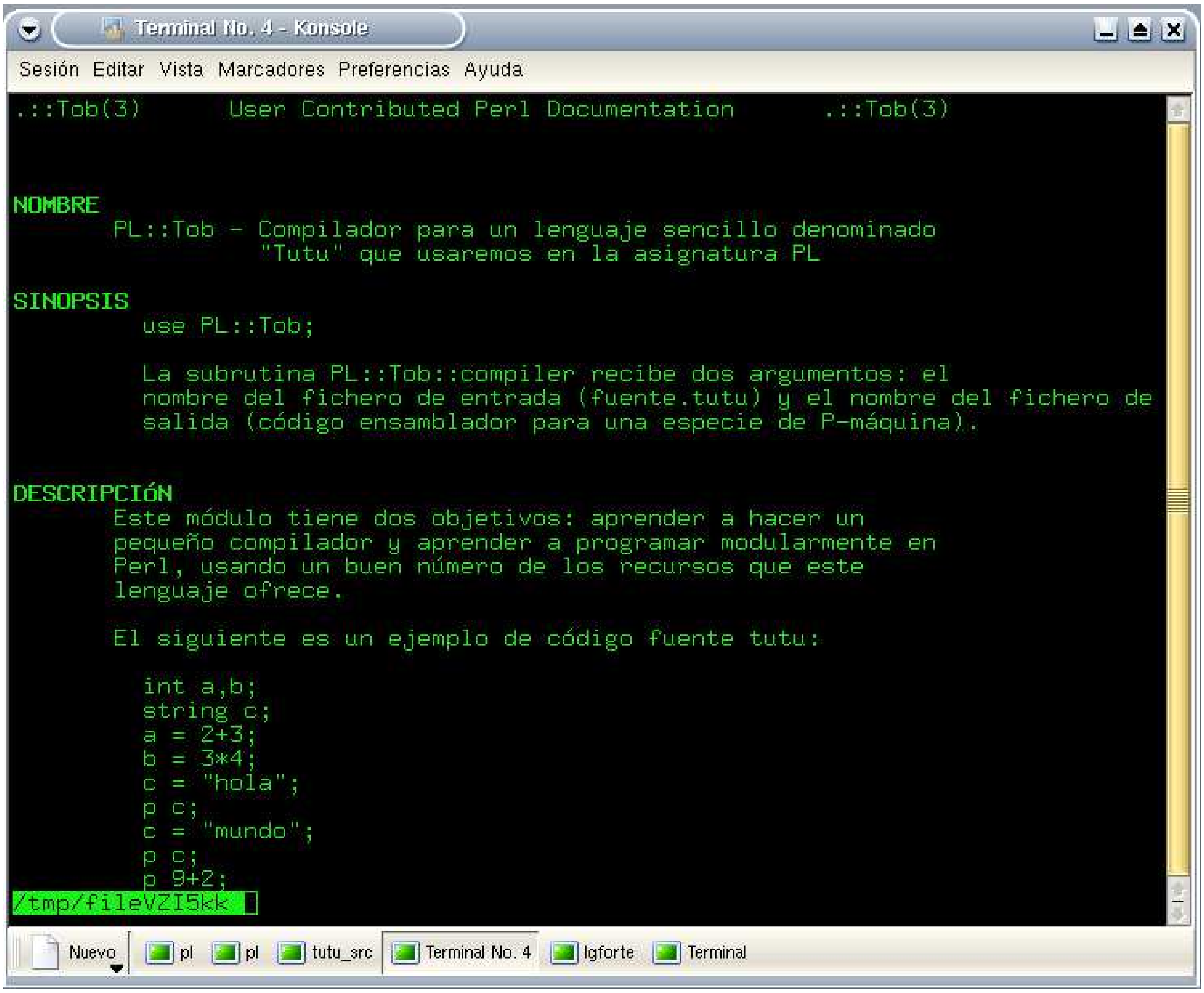
```
1;
__END__

=head1 NOMBRE

PL::Tutu - Compilador para un lenguaje sencillo denominado
          "Tutu" que usaremos en la asignatura PL

=head1 SINOPSIS

  use PL::Tutu;

  La subrutina PL::Tutu::compiler recibe dos argumentos: el
  nombre del fichero de entrada (fuente.tutu) y el nombre del fichero de
  salida (código ensamblador para una especie de P-máquina).

=head1 DESCRIPCIÓN

Este módulo tiene dos objetivos: aprender a hacer un pequeño compilador
y aprender a programar modularmente en Perl, usando un buen número
de los recursos que este lenguaje ofrece.

El siguiente es un ejemplo de código fuente tutu:

  int a,b;
  string c;
  a = 2+3;
  b = 3*4;
  c = "hola";
  p c;
  c = "mundo";
  p c;
  p 9+2;
  p a+1;
  p b+1
```

```
supuesto que está guardado en el fichero "test2.tutu", podemos
escribir un programa Perl "main.pl" para compilarlo:

    $ cat main.pl
    #!/usr/bin/perl -w -I..
    #use PL::Tutu;
    use Tutu;

    PL::Tutu::compiler(@ARGV);

al ejecutar "main.pl":

    $ ./main.pl test2.tutu test2.ok

obtenemos el fichero "test2.ok" con el ensamblador:
    $ cat test2.ok
    DATA holamundo
    PUSH 5
    PUSHADDR 0
    STORE_INT
    PUSH 12
    PUSHADDR 1
    STORE_INT
    PUSHSTR 0 4
    PUSHADDR 2
    STORE_STRING
    LOAD_STRING 2
    PRINT_STR
    PUSHSTR 4 5
    PUSHADDR 2
    STORE_STRING
    LOAD_STRING 2
    PRINT_STR
    PUSH 11
    PRINT_INT
    LOAD 0
    INC
    PRINT_INT
    LOAD 1
    INC
    PRINT_INT

Para mas información consulta la página de la asignatura.
¡Buena suerte!

=head2 EXPORT

No se exporta nada al espacio de nombres del cliente.


=head1 AUTOR
```

```
Casiano Rodríguez León, E<lt>casiano@ull.esE<gt>

=head1 VÉASE TAMBIÉN

L<perl>.

=cut
```

La documentación puede ser mostrada utilizando el comando `perldoc`.

Veáse la figura 2.1.



```
.::Tob(3)      User Contributed Perl Documentation      .::Tob(3)



NOMBRE
       PL::Tob - Compilador para un lenguaje sencillo denominado
                 "Tutu" que usaremos en la asignatura PL

SINOPSIS
         use PL::Tob;

         La subrutina PL::Tob::compiler recibe dos argumentos: el
         nombre del fichero de entrada (fuente.tutu) y el nombre del fichero de
         salida (código ensamblador para una especie de P-máquina).


DESCRIPCIÓN
       Este módulo tiene dos objetivos: aprender a hacer un
       pequeño compilador y aprender a programar modularmente en
       Perl, usando un buen número de los recursos que este
       lenguaje ofrece.

       El siguiente es un ejemplo de código fuente tutu:

         int a,b;
         string c;
         a = 2+3;
         b = 3*4;
         c = "hola";
         p c;
         c = "mundo";
         p c;
         p 9+2;
/tmp/fileVZI5kk
```

Figura 2.1: El resultado de usar `perldoc Tutu`

La forma en la que se controla la calidad de un módulo es mediante el desarrollo de pruebas. Las pruebas son programas perl que se sitúan en el directorio `t/` y que tienen la extensión `.t`. Para ejecutar las pruebas se escribe:

```
make test
```

Vease la sección [2] de los apuntes de LHP para ams detalles.

2.1.1. Repaso: Las Bases

Responda a las siguientes preguntas:

1. ¿Cómo puedo saber con que versión de Perl estoy trabajando?
2. Cuando el intérprete Perl encuentra una sentencia

   ```
   use Este::Modulo;
   ```

   ¿Donde busca el fichero `Modulo.pm`?
3. ¿Con que opción debo usar Perl para ejecutar un programa en la línea de comandos?
4. ¿Cómo se llama el programa que me permite crear el esqueleto para una distribución de un módulo? ¿Con que opciones debo llamarlo?
5. Cuando se crea con `h2xs` el esqueleto para `PL::Tutu`: ¿En que subdirectorio queda el fichero conteniendo el esqueleto del módulo creado `Tutu.pm`?
6. ¿Cuál es la función de `MANIFEST`?
7. ¿Qué es `Makefile.PL`? ¿Cuál es su función? ¿Que significa la frase *looks good*?
8. ¿Con que comando se crea el `Makefile` para trabajar en la plataforma actual?
9. ¿Cómo se puede ver la documentación de un módulo?
10. ¿Que hacen los siguientes comandos `pod`? Repase [2] si tiene dudas.

    ```
    =head1 cabecera
    =head2 cabecera
    =item texto
    =over N
    =back
    =cut
    =pod
    =for X
    =begin X
    =end X
    ```

11. ¿Que secuencia de comandos conocida como *mantra de instalación* es necesario ejecutar para instalar un módulo?
12. ¿Cual es la función del directorio `t`?
13. ¿Que tipo deben tener los programas de prueba para que `make test` los reconozca como pruebas?

2.1.2. Práctica: Crear y documentar el Módulo `PL::Tutu`

Reproduzca los pasos explicados en la sección 2.1 creando el módulo `PL::Tutu` y documentándolo.

- Compruebe que la documentación se muestra correctamente
- Compruebe que puede crear una distribución haciendo `make dist`
- Compruebe que la distribución creada puede instalarse correctamente siguiendo las instrucciones en [2].

2.2. Las Fases de un Compilador

La estructura del compilador, descompuesto en fases, queda explicitada en el código de la subrutina `compile`:

Esquema del Compilador

```
package PL::Tutu;
use 5.008004;   # Versión mínima de Perl 5.84
use strict;     # Variables deben ser declaradas, etc.
use warnings;   # Enviar warnings
use IO::File;
use Carp;       # Provee alternativas a "die" and "warn"

require Exporter;

our @ISA = qw(Exporter);   # Heredamos los métodos de la clase Exporter
our @EXPORT = qw( compile compile_from_file); # Estas funciones serán exportadas
our $VERSION = ’0.01’;     # Variable que define la versión del módulo

our %symbol_table;         # La tabla de símbolos $symbol_table{x} contiene
our $data;                 # la información asociada con el objeto ’x’
our $target;               # tipo, dirección, etc.
our @tokens;               # La lista de terminales
our $errorflag;
our ($lookahead, $value);  # Token actual y su atributo
our $tree;                 # referencia al objeto que contiene
our $global_address;       # el árbol sintáctico

# Lexical analyzer
package Lexical::Analysis;
sub scanner {
}

package Syntax::Analysis;
sub parser {
}

package Machine::Independent::Optimization;
sub Optimize {
}

package Code::Generation;
sub code_generator {
}

package Peephole::Optimization;
sub transform {
}

package PL::Tutu;
sub compile {
  my ($input) = @_; # Observe el contexto!
  local %symbol_table = ();
  local $data = ""; # Contiene todas las cadenas en el programa fuente
  local $target = ""; # target code
  local @tokens =();    # "local" salva el valor que serÃ¡ recuperado al finalizar
  local $errorflag = 0; # el Ã¡mbito
  local ($lookahead, $value) = ();
```

```
  local $tree = undef; # Referencia al Ã¡rbol sintÃctico abstracto
  local $global_address = 0; # Usado para guardar la Ãltima direcciÃn ocupada

  ########lexical analysis
  &Lexical::Analysis::scanner($input);

  ########syntax (and semantic) analysis
  $tree = &Syntax::Analysis::parser;

  ########machine independent optimizations
  &Machine::Independent::Optimization::Optimize;

  ########code generation
  &Code::Generation::code_generator;

  ########peephole optimization
  &Peephole::Optimization::transform($target);

  return \$target; #retornamos una referencia a $target
}

sub compile_from_file {
  my ($input_name, $output_name) = @_; # Nombres de ficheros
  my $fhi;                             # de entrada y de salida
  my $targetref;

  if (defined($input_name) and (-r $input_name)) {
    $fhi = IO::File->new("< $input_name");
  }
  else { $fhi = 'STDIN'; }
  my $input;
  { # leer todo el fichero
    local $/ = undef; # localizamos para evitar efectos laterales
    $input = <$fhi>;
  }
  $targetref = compile($input);

  ########code output
  my $fh = defined($output_name)? IO::File->new("> $output_name") : 'STDOUT';
  $fh->print($$targetref);
  $fh->close;
  1; # El Ã°ltimo valor evaluado es el valor retornado
}

1;  # El 1 indica que la fase de carga termina con éxito
# Sigue la documentación ...
```

Añadiendo Ejecutables

Vamos a añadir un *script* que use el módulo `PL::Tutu` para asi poder ejecutar nuestro compilador:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu/$ mkdir scripts
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu$ cd scripts/
```

A continuación creamos dos versiones del compilador `tutu.pl` y `tutu` y un programa de prueba `test01.tutu`:

```
... # despues de crear los ficheros
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu/scripts$ ls
test01.tutu  tutu  tutu.pl
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu/scripts$ cat tutu.pl
#!/usr/bin/perl -w -I../lib/
use PL::Tutu;

PL::Tutu::compile_from_file(@ARGV);
```

Búsqueda de Librerías en Tiempo de Desarrollo

El programa `tutu` ilustra otra forma de conseguir que el intérprete Perl busque por la librería que está siendo desarrollada, mediante el uso de `use lib`:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu/scripts$ cat tutu
#!/usr/bin/perl -w
use lib ('../lib');
use PL::Tutu;

&PL::Tutu::compile_from_file(@ARGV);
```

Una tercera forma (la que recomiendo):

```
$ export PERL5LIB=~/Lperl/src/topdown/PL0506/02fases/PL-Tutu/lib
$ perl -MPL::Tutu -e 'PL::Tutu::compile_from_file("test01.tutu")'
```

Añadiendo Los Ejecutables al `MANIFEST`

Ahora tenemos que añadir estos ficheros en `MANIFEST` para que formen parte del proyecto. En vez de eso lo que podemos hacer es crear un fichero `MANIFEST.SKIP`:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu$ cat MANIFEST.SKIP
\.o$
^\.cvsignore$
/\.cvsignore$
\.cvsignore$
CVS/[^/]+$
\.svn\b
^Makefile$
/Makefile$
^blib/
\.swp$
\.bak$
\.pdf$
\.ps$
\.sal$
pm_to_blib
\.pdf$
\.tar.gz$
\.tgz$
^META.yml$
```

Ahora al hacer

```
make manifest
```

se crea un fichero `MANIFEST` que contiene los caminos relativos de todos los ficheros en la jerarquía cuyos nombres no casan con una de las expresiones regulares en `MANIFEST.SKIP`.

Para saber mas sobre `MANIFEST` léa [2].

No recomiendo el uso de `MANIFEST.SKIP`. Prefiero un control manual de los ficheros que integran la aplicacion.

Indicando al Sistema de Distribución que los Ficheros son Ejecutables

Es necesario indicarle a Perl que los ficheros añadidos son ejecutables. Esto se hace mediante el parámetro `EXE_FILES` de `WriteMakefile`:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu$ cat Makefile.PL
use 5.008004;
use ExtUtils::MakeMaker;
# See lib/ExtUtils/MakeMaker.pm for details of how to influence
# the contents of the Makefile that is written.
WriteMakefile(
    NAME              => 'PL::Tutu',
    VERSION_FROM      => 'lib/PL/Tutu.pm', # finds $VERSION
    PREREQ_PM         => {}, # e.g., Module::Name => 1.1
    EXE_FILES         => [ 'scripts/tutu.pl', 'scripts/tutu' ],
    ($] >= 5.005 ?     ## Add these new keywords supported since 5.005
      (ABSTRACT_FROM  => 'lib/PL/Tutu.pm', # retrieve abstract from module
       AUTHOR         => 'Lenguajes y Herramientas de Programacion <lhp@>') : ()),
);
```

Perl utilizará esa información durante la fase de instalación para instalar los ejecutables en el *path* de búsqueda.

Reconstrucción de la aplicación

A continuación hay que rehacer el `Makefile`:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu$ perl Makefile.PL
Writing Makefile for PL::Tutu
```

Para crear una versión funcional hacemos `make`:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu$ make
cp scripts/tutu blib/script/tutu
/usr/bin/perl "-MExtUtils::MY" -e "MY->fixin(shift)" blib/script/tutu
cp scripts/tutu.pl blib/script/tutu.pl
/usr/bin/perl "-MExtUtils::MY" -e "MY->fixin(shift)" blib/script/tutu.pl
Manifying blib/man3/PL::Tutu.3pm
```

Para crear el `MANIFEST` hacemos `make manifest`:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu$ make manifest
/usr/bin/perl "-MExtUtils::Manifest=mkmanifest" -e mkmanifest
Added to MANIFEST: scripts/tutu
```

Comprobemos que el test de prueba generado automáticamente por `h2xs` se pasa correctamente:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib', 'b
t/PL-Tutu....ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.08 cusr +  0.00 csys =  0.08 CPU)
```

Ejecución

Podemos ahora ejecutar los guiones:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu$ cd scripts/
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu/scripts$ ls -l
total 12
-rw-r--r--  1 lhp lhp 15 2005-09-29 12:56 test01.tutu
-rwxr-xr-x  1 lhp lhp 92 2005-09-29 13:29 tutu
-rwxr-xr-x  1 lhp lhp 80 2005-09-29 12:58 tutu.pl
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu/scripts$ tutu test01.tutu test01.sal
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu/scripts$ ls -l
total 12
-rw-r--r--  1 lhp lhp  0 2005-09-29 13:53 test01.sal
-rw-r--r--  1 lhp lhp 15 2005-09-29 12:56 test01.tutu
-rwxr-xr-x  1 lhp lhp 92 2005-09-29 13:29 tutu
-rwxr-xr-x  1 lhp lhp 80 2005-09-29 12:58 tutu.pl
```

Veamos los contenidos del programa fuente `test01.tutu` que usaremos para hacer una prueba:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu/scripts$ cat test01.tutu
int a,b;
a = 4
```

Construcción de una Distribución

Para hacer una distribución instalable hacemos `make dist`:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu$ make dist
rm -rf PL-Tutu-0.01
/usr/bin/perl "-MExtUtils::Manifest=manicopy,maniread" \
        -e "manicopy(maniread(),'PL-Tutu-0.01', 'best');"
mkdir PL-Tutu-0.01
mkdir PL-Tutu-0.01/scripts
mkdir PL-Tutu-0.01/lib
mkdir PL-Tutu-0.01/lib/PL
mkdir PL-Tutu-0.01/t
tar cvf PL-Tutu-0.01.tar PL-Tutu-0.01
PL-Tutu-0.01/
PL-Tutu-0.01/scripts/
PL-Tutu-0.01/scripts/test01.tutu
PL-Tutu-0.01/scripts/tutu
PL-Tutu-0.01/scripts/tutu.pl
PL-Tutu-0.01/META.yml
PL-Tutu-0.01/Changes
PL-Tutu-0.01/MANIFEST
PL-Tutu-0.01/lib/
PL-Tutu-0.01/lib/PL/
PL-Tutu-0.01/lib/PL/Tutu.pm
PL-Tutu-0.01/MANIFEST.SKIP
PL-Tutu-0.01/t/
PL-Tutu-0.01/t/PL-Tutu.t
PL-Tutu-0.01/Makefile.PL
PL-Tutu-0.01/README
rm -rf PL-Tutu-0.01
gzip --best PL-Tutu-0.01.tar
```

Después de esto tenemos en el directorio de trabajo el fichero `PL-Tutu-0.01.tar.gz` con la distribución:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu$ ls -ltr
total 72
drwxr-xr-x  2 lhp lhp  4096 2005-09-29 12:01 t
-rw-r--r--  1 lhp lhp  1196 2005-09-29 12:01 README
drwxr-xr-x  3 lhp lhp  4096 2005-09-29 12:01 lib
-rw-r--r--  1 lhp lhp   152 2005-09-29 12:01 Changes
-rw-r--r--  1 lhp lhp   167 2005-09-29 13:23 MANIFEST.SKIP
-rw-r--r--  1 lhp lhp     0 2005-09-29 13:23 pm_to_blib
drwxr-xr-x  6 lhp lhp  4096 2005-09-29 13:23 blib
-rw-r--r--  1 lhp lhp   113 2005-09-29 13:23 MANIFEST.bak
drwxr-xr-x  2 lhp lhp  4096 2005-09-29 13:29 scripts
-rw-r--r--  1 lhp lhp   616 2005-09-29 13:49 Makefile.PL
-rw-r--r--  1 lhp lhp 20509 2005-09-29 13:51 Makefile
-rw-r--r--  1 lhp lhp  3654 2005-09-29 16:34 PL-Tutu-0.01.tar.gz
-rw-r--r--  1 lhp lhp   298 2005-09-29 16:34 META.yml
-rw-r--r--  1 lhp lhp   205 2005-09-29 16:34 MANIFEST
```

2.2.1. Repaso: Fases de un Compilador

1. ¿Que hace la declaración `package nombredepaquete`?
2. ¿Cual es la función de la declaración `use 5.008004`?
3. ¿Cuál es la función de la declaración `use strict`?
4. ¿Cuál es la función de la declaración `use warnings`?
5. ¿Que diferencia hay entre `use warnings` y `perl -w`?
6. ¿Cuál es la función de la declaración `use Carp`? ¿Que diferencia hay entre `croak` y `die`?
7. ¿Qué hace la declaración `our`?
8. ¿Qué es una variable de paquete?
9. ¿Cuál es el nombre completo de una variable de paquete?
10. ¿En que variable especial se sitúan los argumentos pasados a una subrutina?
11. ¿Que hace la declaración `local`?
12. ¿Cómo se declara una variable léxica?
13. ¿Cuál es el prefijo para los hashes?
14. ¿Cómo se hace referencia a un elemento de un hash `%h` de clave `k`?
15. ¿Cómo se hace referencia a un elemento de un array `@a` de índice 7? ¿Que lugar ocupa ese elemento en el array?
16. ¿Cuál es el significado de `undef`?
17. ¿Cuál es el prefijo para las subrutinas?
18. Señale la diferencia entre

    ```
    my ($input) = @_;
    ```

   y

   ```
   my $input = @_;
   ```

   Repase [2].

19. Toda referencia es un escalar: ¿Cierto o falso?
20. Toda referencia es verdadera ¿Cierto o falso?
21. ¿Que diferencia hay entre `use` y `require`? ¿La línea `require Exporter` se ejecuta en tiempo de compilación o en tiempo de ejecución?
22. ¿Que hace la línea `our @ISA = qw(Exporter)`?. Repáse [2].
23. ¿Que hace la línea `our @EXPORT = qw( compile compile_from_file)`?
24. ¿Que diferencia hay entre `EXPORT` y `EXPORT_OK`?. Repase [2].
25. ¿Que hace la línea `our $VERSION = ’0.01`?
26. ¿Que valor tiene una variable no incializada? ¿y si es un array?
27. ¿Que es un array anónimo? (Repase [2])
28. ¿Que es un hash anónimo? (Repase [2])
29. ¿Que hace el operador `=>`?. Repase [2].
30. ¿En que lugar se dejan los ejecutables asociados con una distribución? ¿Cómo se informa a Perl que se trata de ejecutables?
31. ¿Cuál es la función de `MANIFEST.SKIP`? ¿Que hace `make manifest`?
32. ¿Que hace la opción `-I`? ¿Porqué la primera línea de `tutu.pl` comienza:
    `#!/usr/bin/perl -w -I../lib/`?
33. ¿Cómo puedo saber lo que hace el módulo `lib`? ¿Qué hace la línea `use lib (’../lib’)` en el programa `tutu`?
34. ¿Que contiene la variable `PERL5LIB`?
35. ¿Cómo se crea una distribución?
36. ¿Que devuelve `-r $input_name` en la línea 79? Repase [2].
37. ¿Cuál es la función de la variable mágica `$/`? ¿Que se leerá en la línea 86

    ```
    85   local $/ = undef;
    86   my $input = <$fhi>;
    ```

38. ¿Que hace el operador `\`? ¿Que relación hay entre `\$target` y `$target`?.
39. Si `$targetref` es una referencia a la cadena que va a contener el código objeto, ¿Cómo se denota a la cadena referenciada por `$targetref`? Explique la línea

    ```
    92   $fh->print($$targetref);
    ```

2.2.2. Práctica: Fases de un Compilador

Reproduzca los pasos explicados en la sección 2.2 extendiendo el módulo `PL::Tutu` con las funciones de compilación y los correspondientes guiones de compilación.

Mejore el script `tutu` para que acepte opciones desde la línea de comandos. Debera soportar al menos las siguientes opciones:

- `--usage`

  Muestra de forma concisa el comando de uso

- `--help`

  Un resumen de cada opción disponible

- `--version`

  Muestra la versión del programa

- `--man`

  Muestra la documentación

Use para ello el módulo `Getopt::Long`. Este módulo provee la función `GetOptions` la cual se atiene a los estándares de especificación de opciones en la línea de comandos POSIX y GNU. Esta función soporta el uso del guión doble `--` y el simple así como admitir el prefijo mas corto que deshace la ambiguedad entre las diferentes opciones.

La llamada a `GetOptions` analiza la línea de comandos en `ARGV` inicializa la variable asociada de manera adecuada. Retorna un valor verdadero si la línea de comandos pudo ser procesada con En caso contrario emitirá un mensaje de error y devolverá falso. Recuerde hacer `perldoc Getopt::Long` para obtener información mas detallada

El siguiente ejemplo ilustra el uso de `Getopt::Long`. Se hace uso también del módulo (función `pod2usage` en la línea 63) `Pod::Usage` el cual permite la documentación empotrada.

```
nereida:˜/LEyapp/examples> cat -n treereg
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use Parse::Eyapp::YATW;
 4  use Parse::Eyapp::Node;
 5  use Parse::Eyapp::Treeregexp;
 6  use Carp;
 7  use Getopt::Long;
 8  use Pod::Usage;
 9
10  my $infile;
11  my $outfile;
12  my $packagename;
13  my $prefix = ’’;
14  my $syntax = 1;
15  my $numbers = 1;
16  my $severity = 0; # 0 = Don’t  check arity. 1 = Check arity.
17                    # 2 = Check arity and give a warning 3 = ... and croak
18  GetOptions(
19    ’in=s’       => \$infile,
20    ’out=s’      => \$outfile,
21    ’mod=s’      => \$packagename,
22    ’prefix=s’   => \$prefix,
23    ’severity=i’ => \$severity,
24    ’syntax!’    => \$syntax,
```

```
  ’numbers!’   => \$numbers,
  ’version’    => \&version,
  ’usage’      => \&usage,
  ’help’       => \&man,
) or croak usage();

# If an argument remains is the inputfile
($infile) = @ARGV unless defined($infile);
die usage() unless defined($infile);

my $treeparser = Parse::Eyapp::Treeregexp->new(
                    INFILE   => $infile,
                    OUTFILE  => $outfile,
                    PACKAGE  => $packagename,
                    PREFIX   => $prefix,
                    SYNTAX   => $syntax,
                    NUMBERS  => $numbers,
                    SEVERITY => $severity
                  );

$treeparser->generate();

sub version {
  print "Version $Parse::Eyapp::Treeregexp::VERSION\n";
  exit;
}

sub usage {
  print <<"END_ERR";
Supply the name of a file containing a tree grammar (.trg)
Usage is:
treereg [-m packagename] [[no]syntax] [[no]numbers] [-severity 0|1|2|3] \
        [-p treeprefix] [-o outputfile] -i filename[.trg]
END_ERR
  exit;
}

sub man {
  pod2usage(
    -exitval => 1,
    -verbose => 2
  );
}
__END__

=head1 SYNOPSIS

  treereg [-m packagename] [[no]syntax] [[no]numbers] [-severity 0|1|2|3] \
          [-p treeprefix] [-o outputfile] -i filename[.trg]
  treereg [-m packagename] [[no]syntax] [[no]numbers] [-severity 0|1|2|3] \
          [-p treeprefix] [-o outputfile] filename[.trg]
... # Follows the documentation bla, bla, bla
```

Ahora podemos ejecutar el guión de múltiples formas:

```
nereida:~/LEyapp/examples> treereg -nos -nonu -se 3 -m Tutu Foldonly1.trg
nereida:~/LEyapp/examples> treereg -nos -nonu -s 3 -m Tutu Foldonly1.trg
Option s is ambiguous (severity, syntax)
nereida:~/LEyapp/examples> treereg -nos -bla -nonu -m Tutu Foldonly1.trg
Unknown option: bla
nereida:~/LEyapp/examples>
```

La librería estandar de Perl incluye el módulo `Getopt::Long`. No es el caso de `Pod::Usage`. Descarge el módulo e instalelo en un directorio local en el que tenga permisos. Si es preciso repase las secciones [2] y [2] de los apuntes de introducción a Perl.

2.3. Análisis Léxico

Comenzaremos con la parte mas sencilla del compilador: el analizador léxico. Habitualmente el término "análisis léxico" se refiere al tratamiento de la entrada que produce como salida la lista de *tokens*. Un *token* hace alusión a las unidades mas simples que tiene significado. Habitualmente un *token* o lexema queda descrito por una expresión regular. Léxico viene del griego *lexis*, que significa "palabra". Perl es, sobra decirlo, una herramienta eficaz para encontrar en que lugar de la cadena se produce un emparejamiento. Sin embargo, en el análisis léxico, el problema es encontrar la subcadena a partir de la última posición en la que se produjo un emparejamiento y que es aceptada por una de las expresiones regulares que definen los lexemas del lenguaje dado.

La estructura general del analizador léxico consiste en un bucle en el que se va recorriendo la entrada, buscando por un emparejamiento con uno de los patrones/lexemas especificados y, cuando se encuentra, se retorna esa información al analziador sintáctico. Como no tenemos escrito el analaizador sintáctico simplemente iremos añadiéndo los terminales al final de una lista.

Una iteración del bucle tiene la forma de una secuencia de condicionales en las que se va comprobando si la entrada casa con cada una de las expresiones regulares que definen los terminales del lenguaje. Las condiciones tienen un aspecto similar a este:

```
...
if (m{\G\s*(\d+)}gc) {
  push @tokens, 'NUM', $1;
}
elsif (m{\G\s*([a-z_]\w*)\b}igc) {
  push @tokens, 'ID', $1;
}
...
```

Una expresión como `m{\G\s*(\d+)}gc` es una expresión regular. Es conveniente que en este punto repase la introducción a las expresiones regulares en [2].

El Operador de Binding

Nótese que, puesto que no se menciona sobre que variable se hace el *binding* (no se usa ninguno de los operadores de *binding* `=~` y `!~`): se entiende que es sobre la variable por defecto `$_` sobre la que se hace el matching.

Casando a partir del Ültimo Emparejamiento

El ancla `\G` casa con el punto en la cadena en el que terminó el último emparejamiento.

La expresión regular describiendo el patrón de interés se pone entre paréntesis para usar la estrategia de los paréntesis con memoria (véanse 1.1.4 y 1.1.1).

Las opciones `c` y `g` son especialmente útiles para la construcción del analizador.

La opción `g`

Como lo usamos en un contexto escalar, la opción `g` *itera sobre la cadena, devolviendo cierto cada vez que casa, y falso cuando deja de casar.* Se puede averiguar la posicion del emparejamiento utilizando la función `pos`. (véase la sección 1.1.6 para mas información sobre la opción `g`).

La opción `c`

La opción `/c` afecta a las operaciones de emparejamiento con `/g` en un contexto escalar. Normalmente, *cuando una búsqueda global escalar tiene lugar y no ocurre casamiento, la posición de comienzo de búsqueda es reestablecida* al comienzo de la cadena. La opción `/c` hace que la posición inicial de emparejamiento permanezca donde la dejó el último emparejamiento con éxito y no se vaya al comienzo. Al combinar esto con el ancla `\G`, la cuál casa con el final del último emparejamiento, obtenemos que la combinación

```
m{\G\s*(...)}gc
```

logra el efecto deseado: Si la primera expresión regular en la cadena `elsif` fracasa, la posición de búsqueda no es inicializada de nuevo gracias a la opción `c` y el ancla `\G` sigue recordando donde terminó el ultimo casamiento.

La opción `i`

Por último, la opción `i` permite ignorar el tipo de letra (mayúsculas o minúsculas).

Repase la sección 1.1.6 para ver algunas de las opciones mas usadas.

Código del Analizador Léxico

Este es el código completo de la subrutina `scanner` que se encarga del análisis léxico:

```
package Lexical::Analysis;
sub scanner {
  local $_ = shift;
  { # Con el redo del final hacemos un bucle "infinito"
    if (m{\G\s*(\d+)}gc) {
      push @tokens, 'NUM', $1;
    }
    elsif (m{\G\s*int\b}igc) {
      push @tokens, 'INT', 'INT';
    }
    elsif (m{\G\s*string\b}igc) {
      push @tokens, 'STRING', 'STRING';
    }
    elsif (m{\G\s*p\b}igc) {
      push @tokens, 'P', 'P'; # P para imprimir
    }
    elsif (m{\G\s*([a-z_]\w*)\b}igc) {
      push @tokens, 'ID', $1;
    }
    elsif (m{\G\s*\"([^"]*)\"}igc) {
      push @tokens, 'STR', $1;
    }
    elsif (m{\G\s*([+*()=;,])}gc) {
      push @tokens, 'PUN', $1;
    }
    elsif (m{\G\s*(\S)}gc) { # Hay un caracter "no blanco"
      Error::fatal "Caracter invalido: $1\n";
    }
```

```
    else {
      last;
    }
    redo;
  }
}
```

Relación de corutina con el Analizador Sintáctico

Si decidieramos establecer una relación de corutina con el analizador léxico los condicionales se pueden programar siguiendo secuencias con esta estructura:

```
return (’INT’, ’INT’) if (m{\G\s*int\b}igc);
```

Manejo de Errores

Para completar el analizador solo quedan declarar las variables usadas y las subrutinas de manejo de errores:

```
######## global scope variables
our @tokens = ();
our $errorflag = 0;

package Error;

sub error($) {
  my $msg = shift;
  if (!$errorflag) {
    warn "Error: $msg\n";
    $errorflag = 1;
  }
}

sub fatal($) {
  my $msg = shift;
  die "Error: $msg\n";
}
```

El uso de `our` es necesario porque hemos declarado al comienzo del módulo `use strict`. El pragma `use strict` le indica al compilador Perl que debe considerar como obligatorias un conjunto de reglas de buen estilo de programación. Entre otras restricciones, el uso del pragma implica que todas las variables (no-mágicas) deben ser declaradas explícitamente (uso de `my`, `our`, etc.) La declaración `our` se describe en [2].

2.3.1. Ejercicio: La opción g

Explique cada una de las líneas que siguen:

```
$ perl -wde 0
main::(-e:1):   0
  DB<1> $x = "ababab"
  DB<2> $x =~ m{b}g; print "match= ".$&." pos = ".pos($x)
match= b pos = 2
  DB<3> $x =~ m{b}g; print "match= ".$&." pos = ".pos($x)
match= b pos = 4
  DB<4> $x =~ m{b}g; print "match= ".$&." pos = ".pos($x)
```

```
match= b pos = 6
  DB<5> print "falso" unless $x =~ m{b}g
falso
```

2.3.2. Ejercicio: Opciones `g` y `c` en Expresiones Regulares

Explique cada una de las conductas que siguen

- ¿Porqué en la línea 18 se casa con la primera `b`?

```
  DB<5> $x = "bbabab"
  DB<6> $x =~ m{a}g; print "match= ".$&." pos = ".pos($x)
match= a pos = 3
  DB<7> $x =~ m{b}g; print "match= ".$&." pos = ".pos($x)
match= b pos = 4
  DB<8> $x =~ m{c}g; print "match= ".$&." pos = ".pos($x)
Use of uninitialized value in concatenation (.)
  DB<18> $x =~ m{b}g; print "match= ".$&." pos = ".pos($x)
match= b pos = 1
```

- ¿Porqué en la línea 27 se casa con la última `b`?

```
  DB<23> $x = "bbabab"
  DB<24> $x =~ m{a}g; print "match= ".$&." pos = ".pos($x)
match= a pos = 3
  DB<25> $x =~ m{b}g; print "match= ".$&." pos = ".pos($x)
match= b pos = 4
  DB<26> $x =~ m{c}gc
  DB<27> $x =~ m{b}g; print "match= ".$&." pos = ".pos($x)
match= b pos = 6
```

- ¿Porqué en la línea 5 se produce casamiento y en la línea 8 no?

```
  DB<3> $x = "bcbabab"
  DB<4> $x =~ m{b}gc; print "match= ".$&." pos = ".pos($x)
match= b pos = 1
  DB<5> $x =~ m{a}gc; print "match= ".$&." pos = ".pos($x)
match= a pos = 4

  DB<6> $x = "bcbabab"
  DB<7> $x =~ m{b}gc; print "match= ".$&." pos = ".pos($x)
match= b pos = 1
  DB<8> $x =~ m{\Ga}gc; print "match= ".$&." pos = ".pos($x)
Use of uninitialized value in concatenation
match=  pos = 1
```

2.3.3. Ejercicio: El orden de las expresiones regulares

¿Que ocurriría en la subrutina `scanner` si el código en las líneas 17-19 que reconoce los identificadores se adelanta a la línea 8? ¿Que ocurriría con el reconocimiento de las palabras reservadas como `INT`? ¿Seguiría funcionando correctamente el analizador?

2.3.4. Ejercicio: Regexp para cadenas

En la rutina `scanner`. ¿Es legal que una cadena correspondiente al terminal `STR` contenga retornos de carro entre las comillas dobles?

2.3.5. Ejercicio: El `or` es vago

Explique el resultado de la siguiente sesión con el depurador:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu/lib/PL/Lexical$ perl -de 0
  DB<1> 'bb' =~ m{b|bb}; print $&
  b
  DB<2> 'bb' =~ m{bb|b}; print $&
  bb
```

Perl convierte la expresión regular en un NFA. A diferencia de lo que ocurre en otras herramientas, el NFA no es convertido en un DFA. El NFA es entonces simulado. ¿Que está ocurriendo en la simulación?

2.3.6. Práctica: Números de Línea, Errores, Cadenas y Comentarios

Extienda el analizador léxico para que:

- Reescriba la expresión regular para las cadenas de manera que acepte comillas dobles escapadas `\"` en el interior de una cadena. Por ejemplo en `"esta \"palabra\" va entre comillas"`.

  Analice esta solución ¿Es correcta?:

  ```
  DB<1> $stringre = qr{"(\\.|[^\\"])*"}
  DB<2> print $& if '"esta \"palabra\" va entre comillas"' =~ $stringre
  "esta \"palabra\" va entre comillas"
  ```

- Consuma comentarios a la Perl: cualesquiera caracteres después de una almohadilla hasta el final de la línea (`# ...`).

- Consuma comentarios no anidados a la *C* (`/* ... */`.). Repase las secciones sobre expresiones regulares no "greedy" (p. ej. sección 1.1.1) y la sección 1.1.1. Recuerde que, en una expresión regular, la opción `/s` hace que el punto `'.'` empareje con un retorno de carro `\n`. Esto es, el punto "casa" con cualquier carácter.

  Observe el siguiente ejemplo:

  ```
  pl@nereida:~/src/perl/testing$ cat -n ccomments.pl
   1  #!/usr/bin/perl -w
   2  use strict;
   3
   4  sub showmatches {
   5   my ($x, $re) = @_;
   6
   7    for (my $i = 0; $x =~ /$re/gsx; $i++) {
   8      print "matching $i: $1\n";
   9      $i++;
  10    }
  11  }
  12
  13  my $x = <<'EOC';
  14  if (x) {
  15    /* a comment */ return x + 1; /* another comment */
  16  }
  17  else {
  18    return x + 2; /* a last comment */
  19  }
  20  EOC
  ```

```

print "\n************************\n";

my $greedy = q{ (
         /\*  # Abrir comentario
         .*   # Consumir caracteres (greedy)
         \*/  # Cerrar comentario
         )
};
print "Greedy:\n";
showmatches($x, $greedy);

print "\n************************\n";

my $lazy = q{ (
         /\*  # Abrir comentario
         .*?  # Consumir caracteres (lazy)
         \*/  # Cerrar comentario
         )
};
print "Lazy:\n";
showmatches($x, $lazy);
```

Cuando se ejecuta produce:

```
pl@nereida:~/src/perl/testing$ ccomments.pl

************************
Greedy:
matching 0: /* a comment */ return x + 1; /* another comment */
}
else {
  return x + 2; /* a last comment */

************************
Lazy:
matching 0: /* a comment */
matching 2: /* another comment */
matching 4: /* a last comment */
```

Explique la conducta.

- Números en punto flotante (como `-1.32e-04` o `.91`). El siguiente ejemplo intenta ayudarle en la búsqueda de la solución:

```
lhp@nereida:~$ perl -de 0
  DB<1> print "Si" if  (’.5’ =~ m{\d+})
Si
  DB<2> print "Si" if  (’.5’ =~ m{^\d+})

  DB<3> print "Si" if  ’0.7’ =~ m{^\d+(\.\d+)?(e[+-]?\d+)?$}
Si
  DB<4> print "Si" if  ’.7’ =~ m{^\d+(\.\d+)?(e[+-]?\d+)?$}
```

```
  DB<5> print "Si" if  ’1e2’ =~ m{^\d+(\.\d+)?(e[+-]?\d+)?$}
Si
  DB<6> print "Si " while  ’ababa’ =~ m{a}g
Si Si Si
  DB<7> print "@a" if @a = ’ababa’ =~ m{(a)}g
a a a
```

¿Sabría decir porque la respuesta al primer comando del depurador es `si`?.

- Tenga presente el posible conflicto entre los terminales `INT` y `FLOAT`. Si la entrada contiene `3.5` el terminal debería ser `(FLOAT, ’3.5’)` y no `(INT, 3), (’.’, ’.’), (INT, 5)`.
- En esta práctica si lo desea puede instalar y usar el módulo `Regexp::Common` mantenido por *Abigail* el cual provee expresiones regulares para las situaciones mas comunes: números, teléfonos, IP, códigos postales, listas, etc. Puede incluso usarlo para encontrar soluciones a las cuestiones planteadas en esta práctica:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK> perl -MRegexp::Common -e ’print "$RE{num}{int}\n"’
(?:(?:[+-]?)(?:[0123456789]+))
```

Podemos hacer uso directo del hash `%RE` directamente en las expresiones regulares aprovechando que estas interpolan las variables en su interior:

```
nereida:/tmp> cat -n prueba.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3  use Regexp::Common;
     4
     5  my $input = <>;
     6
     7  print "$&\n" if $input =~ /^$RE{num}{real}$/;
nereida:/tmp> ./prueba.pl
23.45
23.45
nereida:/tmp> ./prueba.pl
jshdf
nereida:/tmp>
```

- Para mejorar la calidad de los mensajes de error extienda el par `(terminal, valor)` devuelto por el `scanner` a un par `(terminal, [valor, número de línea ])` cuya segunda componente es un array anónimo conteniendo el valor y el número de línea en el que aparece el terminal.

El siguiente extracto de un analizador léxico muestra como hacerlo:

```
sub _Lexer {

  return(’’, undef) unless defined($input);

  #Skip blanks
      $input=~m{\G((?:
              \s+       # any white space char
          |   \#[^\n]*  # Perl like comments
          )+)}xsgc
    and do {
        my($blanks)=$1;
```

```
    #Maybe At EOF
      pos($input) >= length($input)
    and return(’’, undef);
    $tokenend += $blanks =~ tr/\n//;
  };

  $tokenbegin = $tokenend;

   $input=~/\G(and)/gc
  and return($1, [$1, $tokenbegin]);

   $input=~/\G(?:[A-Za-z_][A-Za-z0-9_]*::)*([A-Za-z_][A-Za-z0-9_]*)/gc
  and do {
   return(’IDENT’, [$1, $tokenbegin]);
  };

  ...................

    $input=~/\G{/gc
  and do {
    my($level,$from,$code);

    $from=pos($input);

    $level=1;
    while($input=~/([{}])/gc) {
        substr($input,pos($input)-1,1) eq ’\\’ #Quoted
      and next;
        $level += ($1 eq ’{’ ? 1 : -1)
      or last;
    }
      $level
    and  _SyntaxError("Not closed open curly bracket { at $tokenbegin");
    $code = substr($input,$from,pos($input)-$from-1);
    $tokenend+= $code=~tr/\n//;
    return(’CODE’, [$code, $tokenbegin]);
  };


  #Always return something
   $input=~/\G(.)/sg
  and do {
   $1 eq "\n" and ++$tokenend;
   return ($1, [$1, $tokenbegin]);
  };
  #At EOF
  return(’’, undef);
}
```

El operador `tr` ha sido utilizado para contar los retornos de carro (descrito en la sección 1.6). El operador, ademas de reemplazar devuelve el número de carácteres reeemplazados o suprimidos:

```
$cuenta = $cielo =~ tr/*/*/; # cuenta el numero de estrellas en cielo
```

Para aprender soluciones alternativas consulte `perldoc -q 'substring'`.

- Mejore sus mensajes de error ahora que lleva la cuenta de los números de línea. En vez de usar las rutinas `error` y `fatal` introducidas en la sección anterior escriba una sola rutina que recibe el nivel de severidad del error (parámetro `$level` en el siguiente ejemplo) y ejecuta la acción apropiada. El código de `_SyntaxError` ilustra como hacerlo:

```
sub _SyntaxError {
  my($level,$message,$lineno)=@_;

  $message= "*".
      [ 'Warning', 'Error', 'Fatal' ]->[$level].
      "* $message, at ".
      ($lineno < 0 ? "eof" : "line $lineno")." at file $filename\n";

      $level > 1
  and die $message;

  warn $message;

  $level > 0 and ++$nberr;

      $nberr == $max_errors
  and die "*Fatal* Too many errors detected.\n"
}
```

2.4. Pruebas para el Analizador Léxico

Queremos separar/aislar las diferentes fases del compilador en diferentes módulos.

Módulo `PL::Error`

Para ello comenzamos creando un módulo conteniendo las rutinas de tratamiento de errores:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu/lib/PL$ pwd
/home/lhp/Lperl/src/topdown/PL0506/03lexico/PL-Tutu/lib/PL
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu/lib/PL$ cat -n Error.pm
package Error;
use strict;
use warnings;
use Carp;

require Exporter;

our @ISA = qw(Exporter);
our @EXPORT = qw( error fatal);
our $VERSION = '0.01';

sub error {
  my $msg = join " ", @_;
  if (!$PL::Tutu::errorflag) {
    carp("Error: $msg\n");
    $PL::Tutu::errorflag = 1;
  }
}
```

```

sub fatal {
  my $msg = join " ", @_;
  croak("Error: $msg\n");
}
```

Observa como accedemos a la variable `errorflag` del paquete `PL::Tutu`. Para usar este módulo desde `PL::Tutu`, tenemos que declarar su uso:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu/lib/PL$ cat -n Tutu.pm | head -8
     1  package PL::Tutu;
     2
     3  use 5.008004;
     4  use strict;
     5  use warnings;
     6  use IO::File;
     7  use Carp;
     8  use PL::Error;
```

En la línea 8 hacemos `use PL::Error` y no `use Error` ya que el módulo lo hemos puesto en el directorio `PL`. No olvides hacer `make manifest` para actualizar el fichero `MANIFEST`.

Módulo `PL::Lexical::Analysis`

Supongamos que además de modularizar el grupo de rutinas de tratamiento de errores queremos hacer lo mismo con la parte del análisis léxico. Parece lógico que el fichero lo pongamos en un subdirectorio de `PL/` por lo que cambiamos el nombre del módulo a `PL::Lexical::Analysis` quedando la jerarquía de ficheros asi:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu/lib/PL$ tree
.
|-- Error.pm
|-- Lexical
|   `-- Analysis.pm
`-- Tutu.pm
```

Por supuesto debemos modificar las correspondientes líneas en `Tutu.pm`:

```
package PL::Tutu;

use 5.008004;
use strict;
use warnings;
use IO::File;
use Carp;
use PL::Error;
use PL::Lexical::Analysis;
...

sub compile {
  my ($input) = @_;
  local %symbol_table = ();
  local $data = ""; # Contiene todas las cadenas en el programa fuente
  local $target = ""; # target code
  my @tokens = ();
  local $errorflag = 0;
```

```
19   local ($lookahead, $value) = ();
20   local $tree = undef; # abstract syntax tree
21   local $global_address = 0;
22
23
24   ########lexical analysis
25   @tokens = &PL::Lexical::Analysis::scanner($input);
26   print "@tokens\n";
27
28   ...
29
30   return \$target;
31 }
```

Observe que ahora `PL::Lexical::Analysis::scanner` devuelve ahora la lista con los terminales y que `@tokens` se ha ocultado en `compile` como una variable léxica (línea 17). En la línea 26 mostramos el contenido de la lista de terminales.

Sigue el listado del módulo conteniendo el analizador léxico. Obsérve las líneas 6, 16 y 44.

```
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu/lib/PL/Lexical$ cat -n Analysis.pm
     1  # Lexical analyzer
     2  package PL::Lexical::Analysis;
     3  use strict;
     4  use warnings;
     5  use Carp;
     6  use PL::Error;
     7
     8  require Exporter;
     9
    10  our @ISA = qw(Exporter);
    11  our @EXPORT = qw( scanner );
    12  our $VERSION = ’0.01’;
    13
    14  sub scanner {
    15    local $_ = shift;
    16    my @tokens;
    17
    18    { # Con el redo del final hacemos un bucle "infinito"
    19      if (m{\G\s*(\d+)}gc) {
    20        push @tokens, ’NUM’, $1;
    21      }
    ..      ...
    37      elsif (m{\G\s*([+*()=;,])}gc) {
    38        push @tokens, ’PUN’, $1;
    39      }
    40      elsif (m{\G\s*(\S)}gc) {
    41        Error::fatal "Caracter invalido: $1\n";
    42      }
    43      else {
    44        return @tokens;
    45      }
    46      redo;
    47    }
    48  }
```

El Programa Cliente

Puesto que en el paquete `PL::Lexical::Analysis` exportamos `scanner` no es necesario llamar la rutina por el nombre completo desde `compile`. Podemos simplificar la línea en la que se llama a `scanner` que queda así:

```
########lexical analysis
@tokens = &scanner($input);
print "@tokens\n";
```

De la misma forma, dado que `PL::Tutu` exporta la función `compile_from_file`, no es necesario llamarla por su nombre completo desde el guión `tutu`. Reescribimos la línea de llamada:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu/scripts$ cat tutu
#!/usr/bin/perl -w
use lib ('../lib');
use PL::Tutu;

&compile_from_file(@ARGV);
```

Actualización del MANIFEST

Como siempre que se añaden o suprimen archivos es necesario actualizar `MANIFEST`:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu$ make manifest
/usr/bin/perl "-MExtUtils::Manifest=mkmanifest" -e mkmanifest
Added to MANIFEST: lib/PL/Lexical/Analysis.pm
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu$ cat -n MANIFEST
     1  Changes
     2  lib/PL/Error.pm
     3  lib/PL/Lexical/Analysis.pm
     4  lib/PL/Tutu.pm
     5  Makefile.PL
     6  MANIFEST
     7  MANIFEST.SKIP
     8  README
     9  scripts/test01.tutu
    10  scripts/tutu
    11  scripts/tutu.pl
    12  t/PL-Tutu.t
```

2.4.1. Comprobando el Analizador Léxico

Queremos comprobar si nuestro código funciona. ¿Cómo hacerlo?. Lo adecuado es llevar una aproximación sistemática que permita validar el código.

Principios Básicos del Desarrollo de Pruebas

En general, la filosofía aconsejable para realizar un banco de pruebas de nuestro módulo es la que se articula en la metodología denominada *Extreme Programming*, descrita en múltiples textos, en concreto en el libro de Scott [3]:

- Todas las pruebas deben automatizarse
- Todos los fallos que se detecten deberían quedar traducidos en pruebas
- La aplicación debería pasar todas las pruebas después de cualquier modificación importante y también al final del día
- El desarrollo de las pruebas debe preceder el desarrollo del código
- Todos los requerimientos deben ser expresados en forma de pruebas

La Jerarquía de Una Aplicación

Pueden haber algunas diferencias entre el esquema que se describe aqui y su versión de Perl. Lea detenidamente el capítulo Test Now, test Forever del libro de Scott [3] y el libro [4] de Ian Langworth y chromatic.

Si usas una versión de Perl posterior la 5.8.0, entonces tu versión del programa `h2xs` creará un subdirectorio `/t` en el que guardar los ficheros de prueba Estos ficheros deberán ser programas Perl de prueba con el tipo `.t`. La utilidad `h2xs` incluso deja un programa de prueba `PL-Tutu.t` en ese directorio. La jerarquía de ficheros con la que trabajamos actualmente es:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu$ make veryclean
rm -f blib/script/tutu blib/script/tutu.pl
rm -rf ./blib Makefile.aperl ...
mv Makefile Makefile.old > /dev/null 2>&1
rm -rf blib/lib/auto/PL/Tutu blib/arch/auto/PL/Tutu
rm -rf PL-Tutu-0.01
rm -f  blib/lib/PL/.Tutu.pm.swp ...
rm -f *~ *.orig */*~ */*.orig
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu$ tree
.
|-- .svn                # use siempre un sistema de control de versiones
|-- Changes             # la historia de cambios
|-- MANIFEST            # lista de ficheros que componen la distribución
|-- MANIFEST.SKIP       # regexps para determinar que ficheros no pertenecen
|-- META.yml            # YML no es XML
|-- Makefile.PL         # generador del Makefle independiente de la plataforma
|-- PL-Tutu-0.01.tar.gz
|-- README              # instrucciones de instalacion
|-- lib
|   ‘-- PL
|       |-- Error.pm            # rutinas de manejo de errores
|       |-- Lexical
|       |   ‘-- Analysis.pm     # modulo con el analizador lexico
|       ‘-- Tutu.pm             # modulo principal
|-- scripts
|   |-- test01.sal   # salida del programa de prueba
|   |-- test01.tutu  # programa de prueba
|   |-- tutu         # compilador
|   ‘-- tutu.pl      # compilador
‘-- t
    ‘-- 01Lexical.t  # prueba consolidada
```

Un Ejemplo de Programa de Prueba

Estos son los contenidos de nuestro primer test:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu$ cd t
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu/t$ ls -l
total 4
-rw-r--r--  1 lhp lhp 767 2005-10-10 11:27 01Lexical.t
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu/t$ cat -n 01Lexical.t
 1  # Before ‘make install’ is performed this script should be runnable with
 2  # ‘make test’. After ‘make install’ it should work as ‘perl PL-Tutu.t’
 3
 4  #########################
 5
```

```
# change ’tests => 1’ to ’tests => last_test_to_print’;

use Test::More tests => 5;
use Test::Exception;

BEGIN { use_ok(’PL::Lexical::Analysis’) };
BEGIN { use_ok(’PL::Tutu’) };

#########################

# Insert your test code below, the Test::More module is use()ed here so read
# its man page ( perldoc Test::More ) for help writing this test script.

can_ok(’PL::Lexical::Analysis’, ’scanner’);

# Test result of call
my $a = ’int a,b; string c; c = "hello"; a = 4; b = a +1; p b’;
my @tokens = scanner($a);
my @expected_tokens = qw{
INT INT
ID a
PUN ,
ID b
PUN ;
STRING STRING
ID c
PUN ;
ID c
PUN =
STR "hello"
PUN ;
ID a
PUN =
NUM 4
PUN ;
ID b
PUN =
ID a
PUN +
NUM 1
PUN ;
P P
ID b
};
is(@tokens, @expected_tokens, "lexical analysis");

# test a lexically  erroneous program
$a = ’int a,b; string c[2]; c = "hello"; a = 4; b = a +1; p b’;
throws_ok { scanner($a) } qr{Error: Caracter invalido:}, ’erroneous program’;
```

El nombre del fichero de prueba debe cumplir que:

- Sea significativo del tipo de prueba
- Que los prefijos de los nombres `01`, `02`, … nos garanticen el orden de ejecución

Ejecución de Las Pruebas

Ahora ejecutamos las pruebas:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib', 'b
t/01Lexical....ok 1/5Possible attempt to separate words with commas at t/01Lexical.t line 49.
t/01Lexical....ok
All tests successful.
Files=1, Tests=5,  0 wallclock secs ( 0.08 cusr +  0.00 csys =  0.08 CPU)
```

O bien usamos `prove`:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu/t$ prove -I../lib 01Lexical.t
01Lexical....ok
All tests successful.
Files=1, Tests=2,  0 wallclock secs ( 0.03 cusr +  0.01 csys =  0.04 CPU)
```

También podemos añadir la opción `verbose` a `prove`:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/03lexico/PL-Tutu/t$ prove -v 01Lexical.t
01Lexical....1..5
ok 1 - use PL::Lexical::Analysis;
ok 2 - use PL::Tutu;
ok 3 - PL::Lexical::Analysis->can('scanner')
ok 4 - lexical analysis
ok 5 - erroneous program
ok
All tests successful.
Files=1, Tests=5,  0 wallclock secs ( 0.07 cusr +  0.01 csys =  0.08 CPU)
```

Repáse [2] para un mejor conocimiento de la metodología de pruebas en Perl.

Versiones anteriores a la 5.8

En esta sección he usado la versión 5.6.1 de Perl. Creeemos un subdirectorio `tutu_src/` y en él un programa de prueba `pruebalex.pl`:

```
$ pwd
/home/lhp/projects/perl/src/tmp/PL/Tutu/tutu_src
$ cat pruebalex.pl
#!/usr/bin/perl -w -I..
#use PL::Tutu;
use Tutu;

my $a = 'int a,b; string c; c = "hello"; a = 4; b = a +1; p b';
Lexical::Analysis::scanner($a);
print "prog = $a\ntokens = @PL::Tutu::tokens\n";
```

Observa como la opción `-I..` hace que se busque por las librerías en el directorio padre del actual. Cuando ejecutamos `pruebalex.pl` obtenemos la lista de terminales:

```
$ ./pruebalex.pl
prog = int a,b; string c; c = "hello"; a = 4; b = a +1; p b
tokens = INT INT ID a PUN , ID b PUN ; STRING STRING ID c PUN ;
ID c PUN = STR hello PUN ; ID a PUN = NUM 4 PUN ; ID b PUN =
ID a PUN + NUM 1 PUN ; P P ID b
```

La última línea ha sido partida por razones de legibilidad, pero consituye una sóla línea. Editemos el fichero `test.pl` en el directorio del módulo. Sus contenidos son como sigue:

```
$ cat -n test.pl
 1  # Before ‘make install’ is performed this script should be runnable with
 2  # ‘make test’. After ‘make install’ it should work as ‘perl test.pl’
 3
 4  #########################
 5
 6  # change ’tests => 1’ to ’tests => last_test_to_print’;
 7
 8  use Test;
 9  BEGIN { plan tests => 1 };
10  use PL::Tutu;
11  ok(1); # If we made it this far, we’re ok.
12
13  #########################
14
15  # Insert your test code below, the Test module is use()ed here so read
16  # its man page ( perldoc Test ) for help writing this test script.
17
```

En la línea 9 se establece el número de pruebas a realizar. La primera prueba aparece en la línea 11. Puede parecer que no es una prueba, ¡pero lo es!. Si se ha alcanzado la línea 11 es que se pudo cargar el módulo `PL::Tutu` y eso ¡tiene algún mérito!.

Seguiremos el consejo de la línea 15 y escribiremos nuestra segunda prueba al final del fichero `test.pl`:

```
$ cat -n test.pl | tail -7
16  # its man page ( perldoc Test ) for help writing this test script.
17
18  my $a = ’int a,b; string c; c = "hello"; a = 4; b = a +1; p b’;
19  local @PL::Tutu::tokens = ();
20  Lexical::Analysis::scanner($a);
21  ok("@PL::Tutu::tokens" eq
22  ’INT INT ID a PUN , ID b PUN ; STRING STRING ID c PUN ; ID c
     PUN = STR hello PUN ; ID a PUN = NUM 4 PUN ; ID b PUN =
     ID a PUN + NUM 1 PUN ; P P ID b’);
```

La línea 22 ha sido partida por razones de legibilidad, pero constituye una sóla línea. Ahora podemos ejecutar `make test` y comprobar que las dos pruebas funcionan:

```
$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl -Iblib/arch -Iblib/lib -I/usr/lib/perl/5.6.1 \
                               -I/usr/share/perl/5.6.1 test.pl
1..2
ok 1
ok 2
```

¿Recordaste cambiar la línea 9 de `test.pl`? ¿Añadiste los nuevos ficheros a la lista en `MANIFEST`?

2.4.2. Práctica: Pruebas en el Análisis Léxico

Extienda su compilador para modularizar el analizador léxico tal y como se explicó en la sección 2.4.

1. Lea los siguientes documentos
   - `Test::Tutorial` (Test::Tutorial - A tutorial about writing really basic tests) por Michael Schwern.
   - Test Now, test Forever " del libro de Scott [3].
   - Perl Testing Reference Card por Ian Langworth.
   - Chapter 4: Distributing Your Tests (and Code) del libro Perl Testing: A Developer's Notebook
2. Incluya la estrategia de pruebas de no regresión explicada en las secciones previas. Dado que ahora la estructura del terminal es una estructura de datos mas compleja `(token, [value, line_number])` no podrá usar `is`, ya que este último sólo comprueba la igualdad entre escalares. Use `is_deeply` para comprobar que la estructura de datos devuelta por el analizador léxico es igual a la esperada. Sigue un ejemplo:

```
nereida:~/src/perl/YappWithDefaultAction/t> cat -n  15treeregswith2arrays.t
 1  #!/usr/bin/perl -w
 2  use strict;
 3  #use Test::More qw(no_plan);
 4  use Test::More tests => 3;
 5  use_ok qw(Parse::Eyapp) or exit;

..  ..... etc., etc.

84  my $expected_tree = bless( {
85    ’children’ => [
86      bless( { ’children’ => [
87          bless( { ’children’ => [], ’attr’ => ’a’, ’token’ => ’a’ }, ’TERMINAL’ )
88        ]
89      }, ’A’ ),
90      bless( { ’children’ => [
91          bless( { ’children’ => [], ’attr’ => ’c’, ’token’ => ’c’ }, ’TERMINAL’ )
92        ]
93      }, ’C’ )
94    ]
95  }, ’ABC’ );
96  is_deeply($t, $expected_tree, "deleting node between arrays");
```

3. Extienda los tests con una prueba en la que la entrada contenga un carácter ilegal. Obsérve que, tal y como esta escrito la rutina `scanner`, si la entrada tiene un carácter ilegal se ejecutarán las líneas

```
26      elsif (/\G\s*(.)/gc) {
27        Error::fatal "Caracter invalido: $1\n";
28      }
```

lo que causa la parada del programa de prueba, al ejecutarse `fatal` el cuál llama a `croak`.

```
  sub fatal {
    my $msg = join " ", @_;
    croak("Error: $msg\n");
  }
```

El objetivo es lograr que el programa de pruebas continúe ejecutando las subsiguientes pruebas.

Para ello puede usar `Test::Exception` o bien `eval` y la variable especial `$@` para controlar que el programa `.t` no termine prematuramente. Repase las secciones sobre pruebas en [2].

4. Pruebe a dar como entrada un fichero vacío
5. Pruebe a dar como entrada un fichero que no existe
6. Pruebe a dar como entrada un fichero binario
7. Si tiene sentido en su caso, llame a las subrutinas con mas argumentos (y también con menos) de los que esperan.
8. Si tiene sentido en su caso, llame a las subrutinas con argumentos cuyo tipo no es el que se espera.

   No use prototipos para lograrlo. No es una buena idea. Los prototipos en Perl a menudo producen un preprocesado del parámetro. Escriba código que controle que la naturaleza del parámetro es la que se espera. Por ejemplo:

   ```
   sub tutu {
     my $refhash = shift;
     croak "Error" unless UNIVERSAL::isa($refhash, 'HASH');
     ...
   }
   ```

9. Cambie los argumentos de orden (si es que se aplica a su código)
10. Comentarios: pruebe con `/* * */ a = 4; /* * / */`. Tambien con comentarios anidados (debería producirse un error)
11. Flotantes: compruebe su expresión regular con 0.0 0e0 .0 0 1.e-5 1.0e2 -2.0 . (un punto sólo)
12. Cadenas. Pruebe con las cadenas

    ```
    ""
    "h\"a\"h"
    "\""
    ```

    Pruebe también con una cadena con varias líneas y otra que contenga un carácter de control en su interior.

13. Convierta los fallos (bugs) que encontró durante el desarrollo en pruebas
14. Compruebe la documentación usando el módulo Test::Pod de Andy Lester. Instálelo si es necesario.
15. Utilice el módulo Test::Warn para comprobar que los mensajes de warning (uso de `warn` and `carp`) se muestran correctamente.
16. Una prueba `SKIP` declara un bloque de pruebas que - bajo ciertas circustancias - puede saltarse. Puede ser que sepamos que ciertas pruebas sólo funcionan en ciertos sistemas operativos o que la prueba requiera que ciertos paquetes están instalados o que la máquina disponga de ciertos recursos (por ejemplo, acceso a internet). En tal caso queremos que los tests se consideren si se dan las circustancias favorables pero que en otro caso se descarten sin protestas. Consulte la documentación de los módulos `Test::More` y `Test::Harness` sobre pruebas tipo `SKIP`. El ejemplo que sigue declara un bloque de pruebas que pueden saltarse. La llamada a `skip` indica cuantos tests hay, bajo que condición saltarselos.

```
SKIP: {
    eval { require HTML::Lint };

    skip "HTML::Lint not installed", 2 if $@;

    my $lint = new HTML::Lint;
    isa_ok( $lint, "HTML::Lint" );

    $lint->parse( $html );
    is( $lint->errors, 0, "No errors found in HTML" );
}
```

Si el usuario no dispone del módulo `HTML::Lint` el bloque no será ejecutado. El módulo `Test::More` producirá `ok`s que serán interpretados por `Test::Harness` como tests *skipped* pero `ok`.

Otra razón para usar una prueba `SKIP` es disponer de la posibilidad de saltarse ciertos grupos de pruebas. Por ejemplo, aquellas que llevan demasiado tiempo de ejecución y no son tan significativas que no se pueda prescindir de ellas cuando se introducen pequeños cambios en el código. El siguiente código muestra como usando una variable de entorno `TEST_FAST` podemos controlar que pruebas se ejecutan.

```
nereida:~/src/perl/YappWithDefaultAction/t> cat 02Cparser.t | head -n 56 -
#!/usr/bin/perl -w
use strict;
#use Test::More qw(no_plan);
use Test::More tests => 6;

use_ok qw(Parse::Eyapp) or exit;

SKIP: {
  skip "You decided to skip C grammar test (env var TEST_FAST)", 5 if $ENV{TEST_FAST} ;
  my ($grammar, $parser);
  $grammar=join(’’,<DATA>);
  $parser=new Parse::Eyapp(input => $grammar, inputfile => ’DATA’, firstline => 52);

  #is($@, undef, "Grammar module created");

  # Does not work. May I have done s.t. wrong?
  #is(keys(%{$parser->{GRAMMAR}{NULLABLE}}), 43, "43 nullable productions");

  is(keys(%{$parser->{GRAMMAR}{NTERM}}), 233, "233 syntactic variables");

  is(scalar(@{$parser->{GRAMMAR}{UUTERM}}), 3, "3 UUTERM");

  is(scalar(keys(%{$parser->{GRAMMAR}{TERM}})), 108, "108 terminals");

  is(scalar(@{$parser->{GRAMMAR}{RULES}}), 825, "825 rules");

  is(scalar(@{$parser->{STATES}}), 1611, "1611 states");
}

__DATA__
/*
   This grammar is a stripped form of the original C++ grammar
```

```
   from the GNU CC compiler :

   YACC parser for C++ syntax.
   Copyright (C) 1988, 89, 93-98, 1999 Free Software Foundation, Inc.
   Hacked by Michael Tiemann (tiemann@cygnus.com)

   The full gcc compiler an the original grammar file are freely
   available under the GPL license at :

   ftp://ftp.gnu.org/gnu/gcc/
   ...................... etc. etc.
*/
nereida:~/src/perl/YappWithDefaultAction> echo $TEST_FAST
1
nereida:~/src/perl/YappWithDefaultAction> make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib
t/01calc...................................ok
t/02Cparser................................ok
        5/6 skipped: various reasons
t/03newgrammar.............................ok
t/04foldandzero............................ok
t/05treewithvars...........................ok
t/06meta...................................ok
t/07translationschemetype..................ok
t/08tschemetypestar........................ok
t/09ts_with_defaultaction..................ok
t/10ts_with_treereg........................ok

etc., etc..................................ok

t/28unshifttwoitems........................ok
t/29foldinglistsofexpressions..............ok
t/30complextreereg.........................ok
t/32deletenodewithwarn.....................ok
t/33moveinvariantoutofloop.................ok
t/34moveinvariantoutofloopcomplexformula....ok
All tests successful, 5 subtests skipped.
Files=33, Tests=113,  5 wallclock secs ( 4.52 cusr +  0.30 csys =  4.82 CPU)
```

Introduzca una prueba `SKIP` similar a la anterior y otra que si el módulo `Test::Pod` esta instalado comprueba que la documentación esta bien escrita. Estudie la documentación del módulo `Test::Pod`.

17. Introduzca pruebas `TODO` (que, por tanto, deben fallar) para las funciones que están por escribir (`parser`, `Optimize`, `code_generator`, `transform`). Repáse [2]. Sigue un ejemplo:

```
TODO: {
  local $TODO = "Randomly generated problem";
  can_ok('Algorithm::Knap01DP', 'GenKnap'); # sub GenKnap no ha sido escrita aún
}
```

18. Cuando compruebe el funcionamiento de su módulo *nunca descarte que el error pueda estar en el código de la prueba.* En palabras de Schwern

> Code has bugs. Tests are code. Ergo, tests have bugs.
>
> Michael Schwern

19. Instale el módulo Devel::Cover. El módulo `Devel::Cover` ha sido escrito por Paul Johnson y proporciona estadísticas del cubrimiento alcanzado por una ejecución. Para usarlo siga estos pasos:

```
pl@nereida:~/src/perl/YappWithDefaultAction$ cover -delete
Deleting database /home/pl/src/perl/YappWithDefaultAction/cover_db
pl@nereida:~/src/perl/YappWithDefaultAction$ HARNESS_PERL_SWITCHES=-MDevel::Cover make te
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib
t/01calc...................................ok
t/01calc...................................ok
t/02Cparser................................ok
        5/6 skipped: various reasons
t/03newgrammar.............................ok
t/03newgrammar.............................ok
t/04foldandzero............................ok
etc., etc. ................................ok
t/34moveinvariantoutofloopcomplexformula....ok
All tests successful, 5 subtests skipped.
Files=33, Tests=113, 181 wallclock secs (177.95 cusr +  2.94 csys = 180.89 CPU)
```

La ejecución toma ahora mucho mas tiempo: ¡181 segundos frente a los 5 que toma la ejecución sin `cover`!. Al ejecutar `cover` de nuevo obtenemos una tabla con las estadísticas de cubrimiento:

```
pl@nereida:~/src/perl/YappWithDefaultAction$ cover
Reading database from /home/pl/src/perl/YappWithDefaultAction/cover_db

---------------------------- ------ ------ ------ ------ ------ ------ ------
File                           stmt   bran   cond    sub    pod   time  total
---------------------------- ------ ------ ------ ------ ------ ------ ------
blib/lib/Parse/Eyapp.pm       100.0    n/a    n/a  100.0    n/a    0.2  100.0
...lib/Parse/Eyapp/Driver.pm   72.4   63.2   50.0   64.3    0.0   21.3   64.4
...ib/Parse/Eyapp/Grammar.pm   90.9   77.8   66.7  100.0    0.0   16.6   84.3
blib/lib/Parse/Eyapp/Lalr.pm   91.4   72.6   78.6  100.0    0.0   48.3   85.6
blib/lib/Parse/Eyapp/Node.pm   74.4   58.3   29.2   88.2    0.0    1.6   64.7
...ib/Parse/Eyapp/Options.pm   86.4   50.0    n/a  100.0    0.0    2.7   72.8
...lib/Parse/Eyapp/Output.pm   82.3   47.4   60.0   70.6    0.0    3.7   70.0
.../lib/Parse/Eyapp/Parse.pm  100.0    n/a    n/a  100.0    n/a    0.2  100.0
...Parse/Eyapp/Treeregexp.pm  100.0    n/a    n/a  100.0    n/a    0.1  100.0
blib/lib/Parse/Eyapp/YATW.pm   89.4   63.9   66.7   85.7    0.0    4.8   77.6
...app/_TreeregexpSupport.pm   73.1   33.3   50.0  100.0    0.0    0.4   60.8
main.pm                        52.2    0.0    n/a   80.0    0.0    0.0   45.7
Total                          83.8   64.7   60.0   84.5    0.0  100.0   75.5
---------------------------- ------ ------ ------ ------ ------ ------ ------

Writing HTML output to /home/pl/src/perl/YappWithDefaultAction/cover_db/coverage.html ...
pl@nereida:~/src/perl/YappWithDefaultAction$
```

El HTML generado nos permite tener una visión mas detallada de los niveles de cubrimiento.

Para mejorar el cubrimiento de tu código comienza por el informe de cubrimiento de subrutinas. Cualquier subrutina marcada como no probada es un candidato a contener errores o incluso a ser *código muerto*.

Para poder hacer el cubrimiento del código usando Devel::Cover, si se usa una **`csh`** o **`tcsh`** se debe escribir:

```
nereida:~/src/perl/YappWithDefaultAction> setenv HARNESS_PERL_SWITCHES -MDevel::Cover
nereida:~/src/perl/YappWithDefaultAction> make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib
t/01calc....................................ok
t/01calc....................................ok
t/02Cparser.................................ok
        5/6 skipped: various reasons
t/03newgrammar..............................ok
t/03newgrammar..............................ok
t/04foldandzero.............................ok
t/05treewithvars............................ok
t/06meta....................................ok
t/06meta....................................ok
t/07translationschemetype...................ok
............................................ok
t/38tspostfix_resultisarray.................ok
t/39tspostfix...............................ok
All tests successful, 5 subtests skipped.
Files=38, Tests=135, 210 wallclock secs (206.28 cusr +  3.27 csys = 209.55 CPU)
nereida:~/src/perl/YappWithDefaultAction>
```

Aún mas robusto - más independiente de la shell que usemos - es pasar las opciones en `HARNESS_PERL_SWITCHE` como parámetro a `make`:

```
make HARNESS_PERL_SWITCHES=-MDevel::Cover test
```

Añade el informe de cubrimiento al `MANIFEST` para que se incluya en la distribución que subas. Si lo consideras conveniente añade un directorio informes en los que vayan los informes asociados a esta práctica. Incluye en el `README` o en la documentación una breve descripción de donde están los informes.

20. Se conoce con el nombre de *perfilado* o *profiling* de un programa al estudio de su rendimiento mediante un programa (conocido como *profiler*) que monitoriza la ejecución del mismo mediante una técnica que interrumpe cada cierto tiempo el programa para comprobar en que punto de la ejecución se encuentra. Las estadísticas acumuladas se vuelcan al final de la ejecución en un fichero que puede ser visualizado mediante la aplicación apropiada.

    En Perl hay dos módulos que permiten realizar profiling. El mas antiguo es `Devel::DProf`. La aplicación para visualizar los resultados se llama `dprofpp`. Sigue un ejemplo de uso:

```
nereida:~/src/perl/YappWithDefaultAction/t> perl -d:DProf 02Cparser.t
1..6
ok 1 - use Parse::Eyapp;
ok 2 - 233 syntactic variables
ok 3 - 3 UUTERM
ok 4 - 108 terminals
ok 5 - 825 rules
ok 6 - 1611 states
nereida:~/src/perl/YappWithDefaultAction/t> dprofpp tmon.out
Total Elapsed Time = 3.028396 Seconds
  User+System Time = 3.008396 Seconds
```

```
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 31.4   0.945  1.473   1611   0.0006 0.0009  Parse::Eyapp::Lalr::_Transitions
 17.5   0.528  0.528   1611   0.0003 0.0003  Parse::Eyapp::Lalr::_Closures
 16.1   0.486  0.892      1   0.4861 0.8918  Parse::Eyapp::Lalr::_ComputeFollows
 8.04   0.242  0.391      1   0.2419 0.3906  Parse::Yapp::Driver::_Parse
 8.04   0.242  0.242  11111   0.0000 0.0000  Parse::Eyapp::Lalr::__ANON__
 4.59   0.138  0.138   8104   0.0000 0.0000  Parse::Eyapp::Lalr::_Preds
 2.66   0.080  0.080      1   0.0800 0.0800  Parse::Eyapp::Lalr::_SetDefaults
 2.66   0.080  0.972      1   0.0800 0.9718  Parse::Eyapp::Lalr::_ComputeLA
 2.46   0.074  0.074   3741   0.0000 0.0000  Parse::Eyapp::Parse::_Lexer
 1.89   0.057  0.074   8310   0.0000 0.0000  Parse::Eyapp::Parse::__ANON__
 0.96   0.029  0.028      1   0.0288 0.0276  Parse::Eyapp::Lalr::_SolveConflict
                                             s
 0.66   0.020  0.050      6   0.0033 0.0083  Parse::Eyapp::Output::BEGIN
 0.60   0.018  1.500      1   0.0176 1.4997  Parse::Eyapp::Lalr::_LR0
 0.53   0.016  0.259      3   0.0054 0.0863  Parse::Eyapp::Lalr::_Digraph
 0.33   0.010  0.010      1   0.0100 0.0100  Parse::Eyapp::Grammar::_SetNullable
```

Tambien es posible usar el módulo `-MDevel::Profiler`:

```
nereida:~/src/perl/YappWithDefaultAction/examples> perl -MDevel::Profiler eyapp 02Cparser
Unused terminals:

        END_OF_LINE, declared line 128
        ALL, declared line 119
        PRE_PARSED_CLASS_DECL, declared line 120

27 shift/reduce conflicts and 22 reduce/reduce conflicts
nereida:~/src/perl/YappWithDefaultAction/examples> dprofpp tmon.out
Total Elapsed Time = 3.914144 Seconds
  User+System Time = 3.917144 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 22.3   0.877  1.577   1611   0.0005 0.0010  Parse::Eyapp::Lalr::_Transitions
 17.8   0.700  0.700   1611   0.0004 0.0004  Parse::Eyapp::Lalr::_Closures
 15.6   0.614  1.185      1   0.6142 1.1854  Parse::Eyapp::Lalr::_ComputeFollow
                                             s
 9.60   0.376  0.545      1   0.3758 0.5453  Parse::Yapp::Driver::_Parse
 7.99   0.313  0.313   8104   0.0000 0.0000  Parse::Eyapp::Lalr::_Preds
 5.85   0.229  0.229      3   0.0763 0.0763  Parse::Eyapp::Lalr::_Digraph
 4.06   0.159  0.159   3741   0.0000 0.0000  Parse::Eyapp::Parse::_Lexer
 3.32   0.130  0.130      1   0.1300 0.1300  Parse::Eyapp::Lalr::DfaTable
 2.27   0.089  0.089      1   0.0890 0.0890  Parse::Eyapp::Lalr::_SetDefaults
 2.04   0.080  1.265      1   0.0800 1.2654  Parse::Eyapp::Lalr::_ComputeLA
 1.17   0.046  0.057      1   0.0464 0.0567  Parse::Eyapp::Grammar::Rules
 1.02   0.040  1.617      1   0.0397 1.6169  Parse::Eyapp::Lalr::_LR0
 0.77   0.030  0.030   1185   0.0000 0.0000  Parse::Eyapp::Lalr::_FirstSfx
 0.71   0.028  0.039      1   0.0284 0.0387  Parse::Eyapp::Grammar::RulesTable
 0.54   0.021  0.021   1650   0.0000 0.0000  Parse::Eyapp::Grammar::classname
```

Presente un informe del perfil de su compilador. Añade el informe del perfil al `MANIFEST` para que se incluya en la distribución que subas.

21. El módulo `Devel::Size` proporciona la posibilidad de conocer cuanto ocupa una estructura de datos. Considere el siguiente ejemplo:

    ```
    .................................... codigo omitido

    use Devel::Size qw(size total_size);
    use Perl6::Form;

    sub sizes {
      my $d = shift;
      my ($ps, $ts) = (size($d), total_size($d));
      my $ds = $ts-$ps;
      return ($ps, $ds, $ts);
    }

    print form(
    ' ==============================================================',
    '| VARIABLE | SOLO ESTRUCTURA |     SOLO DATOS |          TOTAL |',
    '|----------+-----------------+----------------+----------------|',
    '| $parser  | {>>>>>>} bytes  | {>>>>>>} bytes | {>>>>>>} bytes |', sizes($parser),
    '| $t       | {>>>>>>} bytes  | {>>>>>>} bytes | {>>>>>>} bytes |', sizes($t),
    ' ==============================================================',
    );
    ```

    Al ejecutarlo se obtiene esta salida:

    ```
     ....... salida previa omitida

     ==============================================================
    | VARIABLE | SOLO ESTRUCTURA |     SOLO DATOS |          TOTAL |
    |----------+-----------------+----------------+----------------|
    | $parser  |      748 bytes  |      991 bytes |     1739 bytes |
    | $t       |       60 bytes  |     1237 bytes |     1297 bytes |
     ==============================================================
    ```

    Elabore un informe con el consumo de memoria de las variables mas importantes de su programa. Añadelo el informe al `MANIFEST` para que se incluya en la distribución que subas. Explica en el `README` o en la documentación el significado de los ficheros de informe.

2.4.3. Repaso: Pruebas en el Análisis Léxico

1. ¿Cuál es la diferencia entre los operadores `==` y `eq`?
2. ¿Cuáles son los parámetros de la función `ok`?
3. ¿Cuáles son los parámetros de la función `is`?
4. ¿Porqué es conveniente nombrar las pruebas con un nombre que empiece por un número?
5. ¿Como puedo ejecutar los tests en modo *verbose*?
6. ¿Como puedo probar un código que produce la detención del programa?
7. ¿Que contiene la variable `$@`?
8. ¿Que hace la función `like`?

9. ¿Que contiene la variable `$#-`? ¿Y `$+`? (Consulte 1.1.4)
10. ¿Porqué la función `use_ok` es llamada dentro de un `BEGIN`?
11. ¿Que es una prueba `SKIP`?
12. ¿Que es una prueba `TODO`?
13. ¿Que hace la función `pod_file_ok`? ¿A que módulo pertenece?
14. ¿Que hace el operador `tr`?
15. ¿Qué devuelve el operador `tr`?
16. ¿Que hace la opción `d` de `tr`? (consulte 1.6)
17. Explique la conducta de la siguiente sesión con el depurador:

```
  DB<1> $a = ’"Focho \"mucha\" chufa"’
  DB<2> print $a
"Focho \"mucha\" chufa"
  DB<3> print $& if $a =˜ m{ˆ"([ˆ"]|\\")*"}
"Focho \"
  DB<4> print $& if $a =˜ m{ˆ"(\\"||[ˆ"])*"}
"Focho \"mucha\" chufa"
```

18. Supuesto que tuvieramos el operador menos en nuestro lenguaje y dada la entrada `a = 2-3`, ¿Que devolverá su analizador léxico? ¿Devuelve `ID PUN NUM NUM` o bien `ID PUN NUM PUN NUM`? (hemos simplificado el flujo eliminando los atributos).
19. ¿Que hace la llamada `use Test::More qw(no_plan);`?
20. ¿Que hace la función `can_ok`? ¿Qué argumentos tiene?
21. Explique las causas de la siguiente conducta del depurador:

```
  DB<1> $a=’4+5’
  DB<2> print "($&) " while ($a =˜ m/(\G\d+)/gc) or ($a =˜ m/(\G\+)/gc);
(4) (+) (5)
  DB<3> $a=’4+5’ # inicializamos la posición de búsqueda
  DB<4> print "($&) " while ($a =˜ m/(\G\d+)/g) or ($a =˜ m/(\G\+)/g);
(4)
  DB<5> $a=’4+5’
  DB<6> print "($&) " while ($a =˜ m/\G\d+|\G\+/g)
(4) (+) (5)
```

22. ¿Que diferencia hay entre `is_deeply` e `is`?
23. ¿Que argumentos recibe la función `throws_ok`? ¿En que módulo se encuentra?
24. ¿Que hace el comando `HARNESS_PERL_SWITCHES=-MDevel::Cover make test`?
25. ¿Cómo se interpreta el cubrimiento de las sentencias? ¿y de las subrutinas? ¿y de las ramas? ¿y las condiciones lógicas? ¿En cual de estos factores es realista y deseable lograr un cubrimiento del %100 con nuestras pruebas?
26. ¿Que pasa si después de haber desarrollado un número de pruebas cambio la interfaz de mi API?
27. ¿Que hace el comando `perl -d:DProf programa`? ¿Para que sirve?

2.5. Conceptos Básicos para el Análisis Sintáctico

Suponemos que el lector de esta sección ha realizado con éxito un curso en teoría de autómatas y lenguajes formales. Las siguientes definiciones repasan los conceptos mas importantes.

Definición 2.5.1. *Dado un conjunto A, se define A^* el cierre de Kleene de A como: $A^* = \cup_{n=0}^{\infty} A^n$*

Se admite que $A^0 = \{\epsilon\}$, donde ϵ denota la palabra vacía, esto es la palabra que tiene longitud cero, formada por cero símbolos del conjunto base A.

Definición 2.5.2. *Una gramática G es una cuaterna $G = (\Sigma, V, P, S)$. Σ es el conjunto de terminales. V es un conjunto (disjunto de Σ) que se denomina conjunto de* variables sintácticas *o* categorías gramáticales*, P es un conjunto de pares de $V \times (V \cup \Sigma)^*$. En vez de escribir un par usando la notación $(A, \alpha) \in P$ se escribe $A \to \alpha$. Un elemento de P se denomina producción. Por último, S es un símbolo del conjunto V que se denomina símbolo de arranque.*

Definición 2.5.3. *Dada una gramática $G = (\Sigma, V, P, S)$ y $\mu = \alpha A \beta \in (V \cup \Sigma)^*$ una frase formada por variables y terminales y $A \to \gamma$ una producción de P, decimos que μ deriva en un paso en $\alpha\gamma\beta$. Esto es, derivar una cadena $\alpha A \beta$ es sustituir una variable sintáctica A de V por la parte derecha γ de una de sus reglas de producción. Se dice que μ deriva en n pasos en δ si deriva en $n-1$ pasos en una cadena $\alpha A \beta$ la cual deriva en un paso en δ. Se escribe entonces que $\mu \overset{*}{\Longrightarrow} \delta$. Una cadena deriva en 0 pasos en si misma.*

Definición 2.5.4. *Dada una gramática $G = (\Sigma, V, P, S)$ se denota por $L(G)$ o lenguaje generado por G al lenguaje:*

$$L(G) = \{x \in \Sigma^* : S \overset{*}{\Longrightarrow} x\}$$

Esto es, el lenguaje generado por la gramática G esta formado por las cadenas de terminales que pueden ser derivados desde el símbolo de arranque.

Definición 2.5.5. *Una derivación que comienza en el símbolo de arranque y termina en una secuencia formada por sólo terminales de Σ se dice* completa.

Una derivación $\mu \overset{*}{\Longrightarrow} \delta$ en la cual en cada paso $\alpha A x$ la regla de producción aplicada $A \to \gamma$ se aplica en la variable sintáctica mas a la derecha se dice una derivación a derechas

Una derivación $\mu \overset{*}{\Longrightarrow} \delta$ en la cual en cada paso $x A \alpha$ la regla de producción aplicada $A \to \gamma$ se aplica en la variable sintáctica mas a la izquierda se dice una derivación a izquierdas

Definición 2.5.6. *Observe que una derivación puede ser representada como un árbol cuyos nodos están etiquetados en $V \cup \Sigma$. La aplicación de la regla de producción $A \to \gamma$ se traduce en asignar como hijos del nodo etiquetado con A a los nodos etiquetados con los símbolos $X_1 \ldots X_n$ que constituyen la frase $\gamma = X_1 \ldots X_n$. Este árbol se llama* árbol sintáctico concreto *asociado con la derivación.*

Definición 2.5.7. *Observe que, dada una frase $x \in L(G)$ una derivación desde el símbolo de arranque da lugar a un árbol. Ese árbol tiene como raíz el símbolo de arranque y como hojas los terminales $x_1 \ldots x_n$ que forman x. Dicho árbol se denomina* árbol de análisis sintáctico concreto *de x. Una derivación determina una forma de recorrido del árbol de análisis sintáctico concreto.*

Definición 2.5.8. *Una gramática G se dice ambigua si existe alguna frase $x \in L(G)$ con al menos dos árboles sintácticos. Es claro que esta definición es equivalente a afirmar que existe alguna frase $x \in L(G)$ para la cual existen dos derivaciones a izquierda (derecha) distintas.*

2.5.1. Ejercicio

Dada la gramática con producciones:

program → declarations statements | statements
declarations → declaration ';' declarations | declaration ';'
declaration → INT idlist | STRING idlist
statements → statement ';' statements | statement
statement → ID '=' expression | P expression
expression → term '+' expression | term
term → factor '*' term | factor
factor → '(' expression ')' | ID | NUM | STR
idlist → ID ',' idlist | ID

En esta gramática, Σ esta formado por los caracteres entre comillas simples y los símbolos cuyos identificadores están en mayúsculas. Los restantes identificadores corresponden a elementos de V. El símbolo de arranque es $S =$ program.

Conteste a las siguientes cuestiones:

1. Describa con palabras el lenguaje generado.
2. Construya el árbol de análisis sintáctico concreto para cuatro frases del lenguaje.
3. Señale a que recorridos del árbol corresponden las respectivas derivaciones a izquierda y a derecha en el apartado 2.
4. ¿Es ambigua esta gramática?. Justifique su respuesta.

2.6. Análisis Sintáctico Predictivo Recursivo

La siguiente fase en la construcción del analizador es la fase de análisis sintáctico. Esta toma como entrada el flujo de terminales y construye como salida el árbol de análisis sintáctico abstracto.

El árbol de análisis sintáctico abstracto es una representación compactada del árbol de análisis sintáctico concreto que contiene la misma información que éste.

Existen diferentes métodos de análisis sintáctico. La mayoría caen en una de dos categorías: ascendentes y descendentes. Los ascendentes construyen el árbol desde las hojas hacia la raíz. Los descendentes lo hacen en modo inverso. El que describiremos aqui es uno de los mas sencillos: se denomina método de análisis predictivo descendente recursivo.

2.6.1. Introducción

En este método se asocia una subrutina con cada variable sintáctica $A \in V$. Dicha subrutina (que llamaremos `A`) reconocerá el lenguaje generado desde la variable A:

$$L_A(G) = \{x \in \Sigma^* : A \stackrel{*}{\Longrightarrow} x\}$$

En este método se escribe una rutina `A` por variable sintáctica $A \in V$. Se le da a la rutina asociada el mismo nombre que a la variable sintáctica asociada. La función de la rutina `A` asociada con la variable $A \in V$ es reconocer el lenguaje $L(A)$ generado por A. La estrategia general que sigue la rutina `A` para reconocer $L(A)$ es decidir en términos del terminal a en la entrada que regla de producción concreta $A \to \alpha$ se aplica para a continuación comprobar que la entrada que sigue pertenece al lenguaje generado por α. En un analizador predictivo descendente recursivo (APDR) se asume que el símbolo que actualmente esta siendo observado (denotado `lookahead`) permite determinar unívocamente que producción de A hay que aplicar. Una vez que se ha determinado que la regla por la que continuar la derivación es $A \to \alpha$ se procede a reconocer $L_\alpha(G)$, el lenguaje generado por α. Si $\alpha = X_1 \ldots X_n$, las apariciones de terminales X_i en α son emparejadas con los terminales en la entrada mientras que las apariciones de variables $X_i = B$ en α se traducen en llamadas a la correspondiente subrutina asociada con `B`.

Para ilustrar el método, simplificaremos la gramática presentada en el ejercicio 2.5.1 eliminando las declaraciones:

statements → statement ’;’ statements | statement
statement → ID ’=’ expression | P expression
expression → term ’+’ expression | term
term → factor ’*’ term | factor
factor → ’(’ expression ’)’ | ID | NUM

La secuencia de llamadas cuando se procesa la entrada mediante el siguiente programa construye “implícitamente” el árbol de análisis sintáctico concreto.

Dado que estamos usando `strict` se requiere prototipar las funciones al comienzo del fichero:

```
sub parse();
sub statements();
sub statement();
sub expression();
sub term();
sub factor();
sub idlist();
sub declaration();
sub declarations();
```

Para saber mas sobre prototipos consulte [2].

Programa 2.6.1.

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

  if ($lookahead eq $t) {
    ($lookahead, $value) = splice @tokens,0,2;
    if (defined($lookahead)) {
      $lookahead = $value if ($lookahead eq 'PUN');
    } else { $lookahead = 'EOI'; }
  }
  else { error("Se esperaba $t y se encontro $lookahead\n"); }
}

sub statement {
  if ($lookahead eq 'ID') { match('ID'); match('='); expression; }
  elsif ($lookahead eq 'P') { match('P'); expression; }
  else { error('Se esperaba un identificador'); }
}

sub term() {
  factor;
  if ($lookahead eq '*') { match('*'); term; }
}

sub expression() {
  term;
  if ($lookahead eq '+') { match('+'); expression; }
}

sub factor() {
  if ($lookahead eq 'NUM') { match('NUM'); }
  elsif ($lookahead eq 'ID') { match('ID'); }
  elsif ($lookahead eq '(') { match('('); expression; match(')'); }
  else { error("Se esperaba (, NUM o ID"); }
```

```
34 }
35
36 sub statements {
37   statement;
38   if ($lookahead eq ';') { match(';'); statements; }
39 }
40
41 sub parser {
42   ($lookahead, $value) = splice @tokens,0,2;
43   statements; match('EOI');
44 }
```

Como vemos en el ejemplo, el análisis predictivo confía en que, si estamos ejecutando la entrada del procedimiento `A`, el cuál está asociado con la variable $A \in V$, el símbolo terminal que esta en la entrada a determine de manera unívoca la regla de producción $A \rightarrow a\alpha$ que debe ser procesada.

Si se piensa, esta condición requiere que todas las partes derechas α de las reglas $A \rightarrow \alpha$ de A "comiencen" por diferentes símbolos. Para formalizar esta idea, introduciremos el concepto de conjunto $FIRST(\alpha)$:

Definición 2.6.1. *Dada una gramática* $G = (\Sigma, V, P, S)$ *y un símbolo* $\alpha \in (V \cup \Sigma)^*$ *se define el conjunto* $FIRST(\alpha)$ *como:*

$$FIRST(\alpha) = \left\{ b \in \Sigma : \alpha \stackrel{*}{\Longrightarrow} b\beta \right\} \cup N(\alpha)$$

donde:

$$N(\alpha) = \begin{cases} \{\epsilon\} & \textit{si } \alpha \stackrel{*}{\Longrightarrow} \epsilon \\ \emptyset & \textit{en otro caso} \end{cases}$$

Podemos reformular ahora nuestra afirmación anterior en estos términos: Si $A \rightarrow \gamma_1 \mid \ldots \mid \gamma_n$ y los conjuntos $FIRST(\gamma_i)$ son disjuntos podemos construir el procedimiento para la variable A siguiendo este seudocódigo:

```
sub A {
  if ($lookahead in FIRST(gamma_1)) { imitar gamma_1 }
  elsif ($lookahead in FIRST(gamma_2)) { imitar gamma_2 }
  ...
  else ($lookahead in FIRST(gamma_n)) { imitar gamma_n }
}
```

Donde si γ_j es $X_1 \ldots X_k$ el código `gamma_j` consiste en una secuencia $i = 1 \ldots k$ de llamadas de uno de estos dos tipos:

- Llamar a la subrutina `X_i` si X_i es una variable sintáctica
- Hacer una llamada a `match(X_i)` si X_i es un terminal

2.6.2. Ejercicio: Recorrido del árbol en un ADPR

¿En que forma es recorrido el árbol de análisis sintáctico concreto en un analizador descendente predictivo recursivo? ¿En que orden son visitados los nodos?

2.6.3. Ejercicio: Factores Comunes

En el programa 2.6.1 el reconocimiento de las categorías gramáticales statements, expression y term (líneas 19-27) difiere del resto. Observe las reglas:

statements → statement ';' statements | statement
expression → term '+' expression | term
term → factor '*' term | factor

¿Son disjuntos los conjuntos $FIRST(\gamma_i)$ para las partes derechas de las reglas de statements? ¿Son disjuntos los conjuntos $FIRST(\gamma_i)$ para las partes derechas de las reglas de expression? ¿Son disjuntos los conjuntos $FIRST(\gamma_i)$ para las partes derechas de las reglas de term?

Si se tiene una variable con producciones:

$A \rightarrow \alpha\beta \mid \alpha\gamma$

Las dos producciones tienen un *máximo factor común* en la izquierda de su parte derecha α. Asumimos que $FIRST(\beta) \cap FIRST(\gamma) = \emptyset$.

1. ¿Cómo puede modificarse la gramática para obtener una nueva gramática que cumpla la condición de que las partes derechas tienen conjuntos $FIRST(\gamma_i)$ disjuntos?
2. ¿Puede modificarse la técnica APDR para que funcione sobre gramáticas con este tipo de producciones?. Observe el código asociado con statements, expression y term. ¿Cómo sería el esquema general?

2.6.4. Derivaciones a vacío

Surge un problema cuando $A \rightarrow \gamma_1 \mid \ldots \mid \gamma_n$ y la palabra vacía está en alguno de los conjuntos $FIRST(\gamma_i)$. ¿Que hacer entonces?

Nótese que si $A \rightarrow \gamma$ y $\epsilon \in FIRST(\gamma)$ es porque existe una derivación $\gamma \stackrel{*}{\Longrightarrow} \epsilon$. ¿Que terminales podemos legalmente encontrarnos cuando estamos en la subrutina `A`? Consideremos una derivación desde el símbolo de arranque en la que se use la producción $A \rightarrow \gamma$. Dicha derivación forzosamente tendrá la forma:

$$S \stackrel{*}{\Longrightarrow} \beta A\ a\mu \Longrightarrow \beta\gamma\ a\mu \stackrel{*}{\Longrightarrow} \beta\ a\mu.$$

Cualquier terminal $a \in \Sigma$ que pueda aparecer en una derivación desde el símbolo de arranque inmediatamente a continuación de la variable A es susceptible de ser visto cuando se esta analizando A y se aplicó $A \rightarrow \gamma$ con $\gamma \stackrel{*}{\Longrightarrow} \epsilon$. Esto nos lleva a la definición del conjunto $FOLLOW(A)$ como conjunto de terminales que pueden aparecer a continuación de A en una derivación desde el símbolo de arranque:

Definición 2.6.2. *Dada una gramática* $G = (\Sigma, V, P, S)$ *y una variable* $A \in V$ *se define el conjunto* $FOLLOW(A)$ *como:*

$$FOLLOW(A) = \left\{ b \in \Sigma : \exists\ S \stackrel{*}{\Longrightarrow} \alpha A b \beta \right\} \cup E(A)$$

donde

$$E(A) = \left\{ \begin{array}{ll} \{\$\} & \text{si } S \stackrel{*}{\Longrightarrow} \alpha A \\ \emptyset & \text{en otro caso} \end{array} \right.$$

Aqui \$ denota el final de la entrada (que se corresponde en el código Perl anterior con el terminal `EOI`).

Si $A \rightarrow \gamma_1 \mid \ldots \mid \gamma_n$ dado que los conjuntos $FIRST(\gamma_i)$ han de ser disjuntos para que un analizador predictivo APDR funcione, sólo una parte derecha puede contener la palabra vacía en su $FIRST$. Supongamos que es γ_n. Podemos reformular la construcción del procedimiento para la variable A siguiendo este seudocódigo:

```
sub A {
  if ($lookahead in FIRST(gamma_1)) { imitar gamma_1 }
  elsif ($lookahead in FIRST(gamma_2)) { imitar gamma_2 }
  ...
  else ($lookahead in FIRST(gamma_n) or $lookahead in FOLLOW(A)) { imitar gamma_n }
}
```

Un caso particular de $\gamma_n \stackrel{*}{\Longrightarrow} \epsilon$ es que $\gamma_n = \epsilon$. En tal caso, y como es obvio, el significado de `imitar gamma_n` es equivalente a ejecutar una sentencia vacía.

2.6.5. Construcción de los conjuntos de Primeros y Siguientes

Algoritmo 2.6.1. *Construcción de los conjuntos* $FIRST(X)$

Repita el siguiente conjunto de reglas hasta que no se puedan añadir mas símbolos terminales o a ningún conjunto $FIRST(X)$*:*

1. *Si* $X \in \Sigma$ *entonces* $FIRST(X) = X$
2. *Si* $X \rightarrow \epsilon$ *entonces* $FIRST(X) = FIRST(X) \cup \{\epsilon\}$
3. *Si*$X \in V$ *y* $X \rightarrow Y_1 Y_2 \cdots Y_k \in P$ *entonces*

```
$i = 1;$
$hacer$
  $FIRST(X) = FIRST(X) \cup FIRST^*(Y_i);$
  $i++;$
$mientras\ (i \leq k\ y\ \epsilon \in FIRST(Y_i))$
```

4. *Añadir* ϵ *a* $FIRST(X)$ *si* $i \geq k$ *y* $\epsilon \in FIRST(Y_k)$

Aqui $FIRST^*(Y)$ denota al conjunto $FIRST(Y) - \{\epsilon\}$.

Este algoritmo puede ser extendido para calcular $FIRST(\alpha)$ para $\alpha = X_1 X_2 \cdots X_n \in (V \cup \Sigma)^*$. El esquema es anólogo al de un símbolo individual.

Algoritmo 2.6.2. *Construcción del conjunto* $FIRST(\alpha)$

Repita siguiente conjunto de reglas hasta que no se puedan añadir mas símbolos terminales o a ningún conjunto $FIRST(\alpha)$*:*

```
$i = 1;$
$FIRST(\alpha) = \emptyset;$
$hacer$
  $FIRST(\alpha) = FIRST(\alpha) \cup FIRST^*(X_i);$
  $i++;$
$mientras\ (i \leq n\ y\ \epsilon \in FIRST(X_i))$
```

Algoritmo 2.6.3. *Construcción de los conjuntos* $FOLLOW(A)$ $\forall A \in V$*:*

Repetir los siguientes pasos hasta que ninguno de los conjuntos $FOLLOW$ *cambie:*

1. $FOLLOW(S) = \{\$\}$ *($ representa el final de la entrada)*
2. *Si* $A \rightarrow \alpha B \beta$ *entonces*

$$FOLLOW(B) = FOLLOW(B) \cup (FIRST(\beta) - \{\epsilon\})$$

3. *Si* $A \rightarrow \alpha B$ *o* $A \rightarrow \alpha B \beta$ *y* $\epsilon \in FIRST(\beta)$ *entonces*

$$FOLLOW(B) = FOLLOW(B) \cup FOLLOW(A)$$

2.6.6. Ejercicio: Construir los $FIRST$

Construya los conjuntos $FIRST$ de las partes derechas de las reglas de producción de la gramática presentada en el ejercicio 2.5.1.

2.6.7. Ejercicio: Calcular los *FOLLOW*

Modificamos la gramática de la sección 2.6.1 para que admita la sentencia vacía:

statements → statement ’;’ statements | statement
statement → ID ’=’ expression | P expression | ϵ
expression → term ’+’ expression | term
term → factor ’*’ term | factor
factor → ’(’ expression ’)’ | ID | NUM

Calcule los conjuntos *FOLLOW*. ¿Es la nueva gramática susceptible de ser analizada por un analizador predictivo descendente recursivo? ¿Cómo sería el código para la subrutina `statements`?. Escríbalo.

2.6.8. Práctica: Construcción de los FIRST y los FOLLOW

He escrito un módulo llamado Grammar que provee la función `Grammar::Parse` la cual recibe una cadena conteniendo la gramática en formato `yacc` o `eyapp` y devuelve una referencia a un hash conteniendo la información pertinente para el tratamiento de la gramática. Para instalar el módulo tenga en cuenta que depende del módulo `Parse::Yapp`.

Para ilustrar el uso vea los ejemplos en el directorio `scripts`. En concreto veamos el programa `grammar.pl`.

```
Grammar/scripts$ cat -n grammar.pl
#!/usr/bin/perl -w -I../lib
use strict;
use Grammar;
use Data::Dumper;

sub usage {
  print <<"EOI";
usage:
$0 input_grammar
EOI
  die "\n";
}

usage() unless @ARGV;
my $filename = shift;

local $/ = undef;
open my $FILE, "$filename";
my $grammar = <$FILE>;
my $x = Grammar::Parse($grammar);

print Dumper($x);
```

Vamos a darle como entrada la gramática en el fichero `aSb.yp` conteniendo una gramática:

```
Grammar/scripts$ cat -n aSb.yp
%%
S:
    |   ’a’ S ’b’
;
%%
```

Las gramáticas aceptadas por `Grammar::Parse` se adaptan a la sintáxis de las gramáticas reconocidas por `Parse::Yapp`. Una gramática (normalmente con tipo `.yp`) consta de tres partes: la cabeza, el cuerpo y la cola. Cada una de las partes va separada de las otras por el símbolo `%%` en una línea aparte. Así, el `%%` de la línea 1 separa la cabeza del cuerpo. En la cabecera se colocan las declaraciones de terminales (directiva `%token`), cual es el símbolo de arranque (directiva `%start`), etc. El cuerpo contiene las reglas de la gramática y las acciones asociadas. Por último, la cola en nuestro caso no es usada y es vacía. En general, la cola contiene las rutinas de soporte al código que aparece en las acciones asi como, posiblemente, rutinas para el análisis léxico y el tratamiento de errores.

La salida de `Grammar::Parse` es una referencia a un hash cuyas entradas vienen explicadas por los comentarios.

```
Grammar/scripts$ grammar.pl aSb.yp
$VAR1 = {
    'SYMS' => { 'S' => 2, '"b"' => 3, '"a"' => 3 }, # Símbolo => línea
    'NULL' => { 'S' => 1 }, # símbolos que se anulan
    'RULES' => [
                 [ 'S', [] ], # S produce vacío
                 [ 'S', [ '"a"', 'S', '"b"' ] ] # S -> aSb
               ],
    'START' => 'S', # Símbolo de arranque
    'TERM' => [ '"b"', '"a"' ], # terminales /tokens
    'NTERM' => { 'S' => [ 0, 1 ] }  # índices de las reglas de las variables sintácticas
  };
```

Usando la estructura devuelta por la función `Grammar::Parse` escriba un módulo que provea funciones para computar los `FIRST` y los `FOLLOW` de las variables sintácticas de la gramática. No olvide escribir la documentación. Incluya una prueba por cada una de las gramáticas que figuran en el directorio `scripts` del módulo `Grammar`.

Puede encontrar la práctica *casi hecha* en PL::FirstFollow. Asegúrese de entender el algoritmo usado. Aumente el número de pruebas y haga un análisis de cubrimiento.

2.6.9. Gramáticas LL(1)

Una gramática $G = (\Sigma, V, P, S)$ cuyo lenguaje generado $L(G)$ puede ser analizado por un analizador sintáctico descendente recursivo predictivo se denomina *LL(1)*. Una gramática es LL(1) si y sólo si para cualesquiera dos producciones $A \rightarrow \alpha$ y $A \rightarrow \beta$ de G se cumple:

1. $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
2. Si $\epsilon \in FIRST(\alpha)$, entonces $FIRST(\alpha) \cap FOLLOW(A) = \emptyset$

¿De donde viene el nombre LL(1)? La primera L hace alusión al hecho de que el flujo de terminales se lee de izquierda a derecha, accediendo a la entrada por su izquierda (*Left*). La segunda L se refiere a que el método de análisis predictivo construye una derivación a izquierdas. El número entre paréntesis indica el número de terminales que debemos consultar para decidir que regla de producción se aplica. Asi, en una gramática LL(2) la decisión final de que producción elegir se hace consultando los dos terminales a la entrada.

2.6.10. Ejercicio: Caracterización de una gramática LL(1)

Cuando se dice que una gramática es LL(1) si, y sólo si:

1. $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
2. Si $\epsilon \in FIRST(\alpha)$, entonces $FIRST(\alpha) \cap FOLLOW(A) = \emptyset$

se asume que los conjuntos $FIRST(\alpha)$ no son vacíos.

- ¿Que se puede decir de la regla $A \to \alpha$ si $FIRST(\alpha) = \emptyset$?
- ¿Que se puede decir de la variable A si $FOLLOW(A) = \emptyset$?

2.6.11. Ejercicio: Ambiguedad y LL(1)

¿Puede una gramática LL(1) ser ambigua?. Razone su respuesta.

2.6.12. Práctica: Un analizador APDR

Siguiendo con la construcción del compilador para el lenguaje Tutu, escriba un analizador APDR para la siguiente gramática. Reutilice el código de las prácticas de las secciones anteriores (2.3 y 2.4).

program → declarations statements | statements
declarations → declaration ’;’ declarations | declaration ’;’
declaration → INT idlist | STRING idlist
statements → statement ’;’ statements | statement
statement → ID ’=’ expression | P expression | ϵ
expression → term ’+’ expression | term
term → factor ’*’ term | factor
factor → ’(’ expression ’)’ | ID | NUM | STR
idlist → ID ’,’ idlist | ID

2.6.13. Práctica: Generación Automática de Analizadores Predictivos

Objetivo Escriba un módulo `GAP.pm` que provea una subrutina `gap` para la generación automática de un APDR supuesto que la gramática de entrada es LL(1).

La subrutina `gap` recibe como entrada la gramática según la estructura de datos generada por la función `Grammar::Parse` de la versión 0.3 del módulo Grammar.

El Módulo `Grammar` La estructura de datos generada por la función `Grammar::Parse` se explicó en la práctica 2.6.8. La estructura ha sido extendida en esta versión para incluir el código que se sitúe en la zona de cola. Por ejemplo, dada la gramática de entrada:

```
Grammar/03/scripts$ cat -n aSb.yp
%%
S:
    |   ’a’ S ’b’
;
%%

sub Lex {
  local $_ = shift; # input
  my @tokens;


  while ($_) {
    s/^\s*//; # fuera blancos
    push @tokens, $1, $1 if s/^(.)//s
  }
  @tokens;
}

sub main {
```

```
  my $filename = shift;
  my $input;

  if (defined($filename)) {
    local $/ = undef;
    open my $FILE, $filename or die "No se pudo abrir $filename\n";
    $input = <$FILE>;
    close($FILE);
  }
  else { $input = <STDIN> }

  my @tokens = Lex($input);
  Parse(@tokens); # Llamada al analizador generado
  print "Sintácticamente correcto\n";
}
```

se genera la siguiente estructura de datos:

```
{
  'SYMS' => { 'S' => 2, 'b' => 3, 'a' => 3 },  # Símbolo => línea de aparición
  'NULL' => { 'S' => 1 }, # Símbolos que se anulan
  'RULES' => [ # Reglas
               [ 'S', [] ], # S produce vacío
               [ 'S', [ 'a', 'S', 'b' ] ] # S-> a S b
             ],
  'START' => 'S', # Símbolo de arranque
  'TERM' => [ 'b', 'a' ], # Terminales
  'NTERM' => { 'S' => [ 0, 1 ] } # Variables sintácticas e índices de las reglas de esa variab
  'TAIL' => [ # [ 'Código de cola', línea en la que está el segundo %% ]
  '

  sub Lex {
    local $_ = shift; # input
    my @tokens;


    while ($_) {
      s/^\\s*//; # fuera blancos
      push @tokens, $1, $1 if s/^(.)//s
    }
    @tokens;
  }

  sub main {
    my $filename = shift;
    my $input;

    if (defined($filename)) {
      local $/ = undef;
      open my $FILE, $filename or die "No se pudo abrir $filename\\n";
      $input = <$FILE>;
      close($FILE);
    }
    else { $input = <STDIN> }
```

```
    my @tokens = Lex($input);
    my $ok = Parse(@tokens); # Llamada al analizador generado
    print "Sintácticamente correcto\\n" if $ok;
  }

  ', 5 ], # línea en la que está el segundo %%
 };
```

Asi pues la entrada con clave `TAIL` contiene el código auxiliar de cola. Este código debe ser incluido por su programa dentro del texto del paquete generado por `gap`.

Descripción del objetivo: La función `gap` La función `gap` también recibe como entrada el nombre del package:

```
$package_text = &gap($grammar, 'Package_name');
```

La función `gap` retorna una cadena conteniendo el `package` en el que estan las subrutinas del analizador sintáctico.

La idea es que dicha cadena se salvará en un fichero con nombre `Package_name.pm` que podrá posteriormente ser usado (`use Package_name`) por un programa que necesite analizar entradas que se conforman de acuerdo a la especificación de la gramática.

Descripción del objetivo: La función `parser` La rutina principal del paquete generado se ha de llamar `parser` (esto es, su nombre completo es: `Package_name::parser`. Evidentemente `Package_name` debe ser un nombre Perl válido). Ninguna subrutina deberá ser exportada sino que deberán ser llamadas por su nombre completo.

La subrutina `parser` recibe como argumento el array de terminales, obtiene el primer terminal y llama a la subrutina asociada con el símbolo de arranque. Los terminales están representados como parejas $(terminal, atributo)$.

Observe que, una vez que la cadena `$package_text` conteniendo el paquete ha sido generada y salvada en un fichero con nombre `Package_name.pm`, podemos escribir un programa cliente:

```
use strict;
use Package_name;

&Package_name::main;
```

Este programa espera una entrada desde fichero o `STDIN` e informa si dicha entrada es sintácticamente correcta o no para la gramática en cuestión.

Cálculo de los First y los Follow con `PL::FirstFollow` Para facilitar la escritura de `GAP.pm` pueden hacer uso del módulo PL::FirstFollow el cual calcula los $FIRST$ y los $FOLLOW$. El módulo `PL::FirstFollow` depende de Set::Scalar escrito por Jarkko Hietaniemi: instálelo primero.

Deberá familiarizarse con `PL::FirstFollow`, rellenar la documentación de todas las subrutinas (apariciones de `????` en el texto) y escribir la documentación siguiendo el template que se provee. *Rellene los fragmentos de código que se han sustituido por signos de interrogación.* Haga un estudio de cubrimiento y añada pruebas para mejorar el actual. El actual cubrimiento es:

```
---------------------------- ------ ------ ------ ------ ------ ------ ------
File                           stmt   bran   cond    sub    pod   time  total
---------------------------- ------ ------ ------ ------ ------ ------ ------
...ammar-0.03/lib/Grammar.pm  100.0    n/a    n/a  100.0    0.0   75.3   97.2
blib/lib/PL/FirstFollow.pm    100.0   92.9   50.0  100.0    0.0   24.7   95.1
Total                         100.0   92.9   50.0  100.0    0.0  100.0   95.5
---------------------------- ------ ------ ------ ------ ------ ------ ------
```

Si observa un fallo en `PL::FirstFollow` háganoslo saber y además de resolverlo escriba una prueba para detectar el fallo.

Haga un estudio de profiling de su aplicación.

Uso de Templates Un módulo que puede facilitar la escritura de esta práctica es Text::Template debido a Mark Jason Dominus. El siguiente ejemplo de uso es un fragmento de un traductor - que nunca acabo de terminar - que toma con fuente un fichero en el formato que usa Moodle para los cuestionarios (conocido como formato GIFT) y lo convierte en un cuestionario LaTeX:

```
lhp@nereida:~/projects/Gift2LaTeX/Gift2LaTeX/lib$ cat -n Gift2LaTeX.pm
   1  package Gift2LaTeX;
   2
   3  use strict;
   4  use warnings;
   5  use Gift;
   6  use Text::Template;
   7  use HTML::Latex;
  ..    ......................................................
  49  package Gift::TRUEFALSE; # True-false questions belong to this class
  50
  51  { # closure
  52
  53    die "Can't find $TEMPLATE_DIR/TRUEFALSE_question.tep\n"
  54          unless -e "$TEMPLATE_DIR/TRUEFALSE_question.tep";
  55    my $tfq_tmpl = Text::Template->new( #tfq = true-false question
  56      DELIMITERS => ['%<', '%>'];
  57      SOURCE => "$TEMPLATE_DIR/TRUEFALSE_question.tep",
  58    );
  ..    ......................................................
  67    sub gen_latex {
  68      my $self = shift;
  69
  70      ########## Generate latex for question
  71      my $prefix  = $self->PREFIX;
  72
  73      my $sufix = $self->POSTSTATE;
  74
  75      $self->Error("Only HTML and PLAIN formats are supported\n")
  76          unless (!$self->FORMAT or ($self->FORMAT =~ m{html|plain}i));
  77
  78      my ($prefix_tex, $sufix_tex);
  79      if (defined($self->FORMAT) and $self->FORMAT =~ m{plain}i) {
  80        $prefix_tex = $prefix;
  81        $sufix_tex  = $sufix;
  82      }
  83      else { # HTML by default
  ..    ......................................................
  86      }
  87      my $params = {
  88        prefix  => $prefix_tex,
  89        sufix   => $sufix_tex,
  90        separator => $separator,
  91        label   => $label_prefix.$question_number,
  92        question_number => $question_number
```

```
 93      };
 94      my $question_tex = $tfq_tmpl->fill_in(HASH => $params);
 96      ########## Generate latex for answer
...      ...................................
101    }
102  }
```

En la línea 55 se crea el template. El template se lee desde el fichero `"$TEMPLATE_DIR/TRUEFALSE_question.tep"` cuyo contenido es una mezcla de texto (en este caso texto LaTeX y HTML) con código Perl: El código Perl aparece acotado entre los delimitadores `'%<'` y `'%>'`.

```
lhp@nereida:~/projects/Gift2LaTeX/Gift2LaTeX/etc/en$ cat -n TRUEFALSE_question.tep
 1  \ begin{latexonly}
 2    %<$separator%>
 3    \ label{question:%<$label%>}
 4    %<$prefix%>
 5
 6    \ begin{center}
 7      \ begin{tabular}{llll}
 8          $\ bigcirc$ & TRUE & $\ bigcirc$ & FALSE
 9      \ end{tabular}
10
11      \noindent %<$sufix%>
12    \ end{center}
13  \ end{latexonly}
14
15  \ begin{htmlonly}
16  %<$separator%>
17  \ label{question:%<$label%>}
18  %<$prefix%>
19
20  \ begin{center}
21    \ begin{tabular}{llll}
22        \ htmlref{$\bigcirc$}{answer:%<$label%>} & TRUE &
23        \ htmlref{$\bigcirc$}{answer:%<$label%>} & FALSE
24    \ end{tabular}
25
26    \ noindent %<$sufix%>
27  \ end{center}
28  \ end{htmlonly}
```

El template se rellena en las líneas 87-94. En esa llamada se ejecuta el código Perl incrustado en el esqueleto y su resultado se inserta en la posición que ocupa en el texto.

Concatenación y Documentos `HERE` Cuando concatene sangre adecuadamente las concatenaciones:

```
my $usage = "Usage: $0 <file> [-full] [-o] [-beans]\n"
            . "Options:\n"
            . "    -full  : produce a full dump\n"
            . "    -o     : dump in octal\n"
            . "    -beans : source is Java\n"
            ;
```

ponga el punto al principio de la siguiente línea, no al final.

Pero cuando el número de líneas es grande es mejor usar un *here document* o *documento aqui.* Veamos un ejemplo:

```
print <<"EOI";
El programa se deberá ejecutar con:

$0 numfiles $opciones initialvalue
EOI
```

Para definir un “documento aqui” se escribe la etiqueta entrecomillada y precedida de `<<` y sigue el texto que consituye el *here document* que se delimita por una línea en blanco que empieza por la etiqueta. Al documento aquí se le trata como una cadena de doble comilla si la etiqueta aparece en doble comilla y como de comilla simple si la etiqueta esta entre comillas simples. Observe que el punto y coma se escribe despues de la primera aparición de la etiqueta.

Un problema con el uso de los heredoc es que rompen la estructura normal del sangrado:

```
if ($usage_error) {
    warn <<'END_USAGE';
Usage: qdump <file> [-full] [-o] [-beans]
Options:
    -full  : produce a full dump
    -o     : dump in octal
    -beans : source is Java
END_USAGE
}
```

Es mejor que cada heredoc se aisle en una subrutina y se parametrice con las variables que van a ser interpoladas:

```
sub build_usage {
    my ($prog_name, $file_name) = @_;

    return <<"END_USAGE";
Usage: $prog_name $file_name [-full] [-o] [-beans]
Options:
    -full  : produce a full dump
    -o     : dump in octal
    -beans : source is Java
END_USAGE
}
```

que mas tarde puede ser llamado con los valores de interpolación adecuados:

```
if ($usage_error) {
    warn build_usage($PROGRAM_NAME, $requested_file);
}
```

Véase el libro de Conway Perl Best Practices [5] para mas detalles sobre buenas prácticas de programación con heredocs.

Descarga de los Módulos Necesarios

- El módulo `Grammar`: http://nereida.deioc.ull.es/~pl/perlexamples/Grammar-0.03.tar.gz
- El módulo `PL::FirstFollow`: http://nereida.deioc.ull.es/~pl/perlexamples/PL-FirstFollow-0.02.tar.gz

2.7. Esquemas de Traducción

Definición 2.7.1. *Un* esquema de traducción *es una gramática independiente del contexto en la cual se han insertado fragmentos de código en las partes derechas de sus reglas de producción. Los fragmentos de código asi insertados se denominan* acciones semánticas. *Dichos fragmentos actúan, calculan y modifican los atributos asociados con los nodos del árbol sintáctico. El orden en que se evalúan los fragmentos es el de un recorrido primero-profundo del árbol de análisis sintáctico.*

Obsérvese que, en general, para poder aplicar un esquema de traducción hay que construir el árbol sintáctico y después aplicar las acciones empotradas en las reglas en el orden de recorrido primero-profundo. Por supuesto, si la gramática es ambigua una frase podría tener dos árboles y la ejecución de las acciones para ellos podría dar lugar a diferentes resultados. Si se quiere evitar la multiplicidad de resultados (interpretaciones semánticas) es necesario precisar de que árbol sintáctico concreto se esta hablando.

Por ejemplo, si en la regla $A \rightarrow \alpha\beta$ insertamos un fragmento de código:

$$A \rightarrow \alpha\{action\}\beta$$

La acción $\{action\}$ se ejecutará después de todas las acciones asociadas con el recorrido del subárbol de α y antes que todas las acciones asociadas con el recorrido del subárbol β.

El siguiente esquema de traducción recibe como entrada una expresión en infijo y produce como salida su traducción a postfijo para expresiones aritmeticas con sólo restas de números:

$expr \rightarrow expr_1 - NUM$ `{ $expr{TRA} = $expr[1]{TRA}." ".$NUM{VAL}." - "}`
$expr \rightarrow NUM$ `{ $expr{TRA} = $NUM{VAL} }`

Las apariciones de variables sintácticas en una regla de producción se indexan como se ve en el ejemplo, para distinguir de que nodo del árbol de análisis estamos hablando. Cuando hablemos del atributo de un nodo utilizaremos una indexación tipo *hash*. Aquí `VAL` es un atributo de los nodos de tipo NUM denotando su valor numérico y para accederlo escribiremos `$NUM{VAL}`. Análogamente `$expr{TRA}` denota el atributo "traducción" de los nodos de tipo *expr*.

Ejercicio 2.7.1. *Muestre la secuencia de acciones a la que da lugar el esquema de traducción anterior para la frase 7 -5 -4.*

En este ejemplo, el cómputo del atributo `$expr{TRA}` depende de los atributos en los nodos hijos, o lo que es lo mismo, depende de los atributos de los símbolos en la parte derecha de la regla de producción. Esto ocurre a menudo y motiva la siguiente definición:

Definición 2.7.2. *Un atributo tal que su valor en un nodo puede ser computado en términos de los atributos de los hijos del nodo se dice que es un* atributo sintetizado.

Ejemplo 2.7.1. *Un ejemplo de atributo heredado es el tipo de las variables en las declaraciones:*

$decl \rightarrow type$ `{ $list{T} = $type{T} }` $list$
$type \rightarrow INT$ `{ $type{T} = $int }`
$type \rightarrow STRING$ `{ $type{T} = $string }`
$list \rightarrow ID$, `{ $ID{T} = $list{T}; $list_1{T} = $list{T} }` $list_1$
$list \rightarrow ID$ `{ $ID{T} = $list{T} }`

Definición 2.7.3. *Un* atributo heredado *es aquel cuyo valor se computa a partir de los valores de sus hermanos y de su padre.*

Ejercicio 2.7.2. *Escriba un esquema de traducción que convierta expresiones en infijo con los operadores* `+-*/()` *y números en expresiones en postfijo. Explique el significado de los atributos elegidos.*

2.8. Recursión por la Izquierda

Definición 2.8.1. *Una* gramática es recursiva por la izquierda *cuando existe una derivación* $A \stackrel{*}{\Longrightarrow} A\alpha$.

En particular, es recursiva por la izquierda si contiene una regla de producción de la forma $A \rightarrow A\alpha$. *En este caso se dice que la recursión por la izquierda es directa.*

Cuando la gramática es *recursiva por la izquierda*, el método de análisis recursivo descendente predictivo no funciona. En ese caso, el procedimiento `A` asociado con A ciclaría para siempre sin llegar a consumir ningún terminal.

2.8.1. Eliminación de la Recursión por la Izquierda en la Gramática

Es posible modificar la gramática para eliminar la recursión por la izquierda. En este apartado nos limitaremos al caso de recursión por la izquierda directa. La generalización al caso de recursión por la izquierda no-directa se reduce a la iteración de la solución propuesta para el caso directo.

Consideremos una variable A con dos producciones:

$$A \rightarrow A\alpha \quad | \, \beta$$

donde $\alpha, \beta \in (V \cup \Sigma)^*$ no comienzan por A. Estas dos producciones pueden ser sustituidas por:

$$\begin{array}{ll} A \rightarrow \beta R & \\ R \rightarrow \alpha R & | \, \epsilon \end{array}$$

eliminando así la recursión por la izquierda.

Definición 2.8.2. *La producción* $R \rightarrow \alpha R$ *se dice* recursiva por la derecha.

Las producciones recursivas por la derecha dan lugar a árboles que se hunden hacia la derecha. Es mas difícil traducir desde esta clase de árboles operadores como el menos, que son asociativos a izquierdas.

Ejercicio 2.8.1. *Elimine la recursión por la izquierda de la gramática*

$$\begin{array}{l} expr \rightarrow expr - NUM \\ expr \rightarrow NUM \end{array}$$

Ejercicio 2.8.2. *¿Que hay de erróneo en este esquema de traducción?*

$expr \rightarrow NUM - expr_1$ `{ $expr{T} = $NUM{VAL}." ".$expr[1]{T}." - "}`
$expr \rightarrow NUM$ `{ $expr{T} = $NUM{VAL} }`

Ejercicio 2.8.3. *Dado el esquema de traducción:*

$e \rightarrow NUM\ r$ `{ $e{TRA} = $NUM{VAL}." ".$r{TRA} }`
$r \rightarrow -e$ `{ $r{TRA} = $e{TRA}." - " }`
$r \rightarrow \epsilon$ `{ $r{TRA} = "" }`

¿Cuál es el lenguaje generado por la gramática? ¿Puede el lenguaje ser analizado por un APDR? ¿Cual es la traducción de 4-5-6? ¿Es un esquema de traducción adecuado para traducir de infijo a postfijo? ¿Cuál es la traducción si cambiamos el anterior esquema por este otro?:

$e \rightarrow NUM\ r$ `{ $e{TRA} = $NUM{VAL}." ".$r{TRA} }`
$r \rightarrow -e$ `{ $r{TRA} = " - ".$e{TRA} }`
$r \rightarrow \epsilon$ `{ $r{TRA} = "" }`

2.8.2. Eliminación de la Recursión por la Izquierda en un Esquema de Traducción

La eliminación de la recursión por la izquierda es sólo un paso: debe ser extendida a esquemas de traducción, de manera que no sólo se preserve el lenguaje sino la secuencia de acciones. Supongamos que tenemos un esquema de traducción de la forma:

$A \to A\alpha$ `{ alpha_action }`
$A \to A\beta$ `{ beta_action }`
$A \to \gamma$ `{ gamma_action }`

para una sentencia como $\gamma\beta\alpha$ la secuencia de acciones será:

```
gamma_action  beta_action alpha_action
```

¿Cómo construir un esquema de traducción para la gramática resultante de eliminar la recursión por la izquierda que ejecute las acciones asociadas en el mismo orden?. Supongamos para simplificar, que las acciones no dependen de atributos ni computan atributos, sino que actúan sobre variables globales. En tal caso, la siguiente ubicación de las acciones da lugar a que se ejecuten en el mismo orden:

$A \to \gamma$ `{ gamma_action }` R
$R \to \beta$ `{ beta_action }` R
$R \to \alpha$ `{ alpha_action }` R
$R \to \epsilon$

Si hay atributos en juego, la estrategia para construir un esquema de traducción equivalente para la gramática resultante de eliminar la recursividad por la izquierda se complica. Consideremos de nuevo el esquema de traducción de infijo a postfijo de expresiones aritméticas de restas:

$expr \to expr_1 - NUM$ `{ $expr{T} = $expr[1]{T}." ".$NUM{VAL}." - "}`
$expr \to NUM$ `{ $expr{T} = $NUM{VAL} }`

En este caso introducimos un atributo `H` para los nodos de la clase r el cuál acumula la traducción a postfijo hasta el momento. Observe como este atributo se computa en un nodo r a partir del correspondiente atributo del el padre y/o de los hermanos del nodo:

$expr \to NUM$ `{ $r{H} = $NUM{VAL} }` r `{ $expr{T} = $r{T} }`
$r \to -NUM$ `{ $r_1{H} = $r{H}." ".$NUM{VAL}." - " }` r_1 `{ $r{T} = $r_1{T} }`
$r \to \epsilon$ `{ $r{T} = $r{H} }`

El atributo `H` es un ejemplo de atributo heredado.

2.8.3. Ejercicio

Calcule los valores de los atributos cuando se aplica el esquema de traducción anterior a la frase `4 - 5 - 7`.

2.8.4. Convirtiendo el Esquema en un Analizador Predictivo

A partir del esquema propuesto, que se basa en una fase de descenso con un atributo heredado y una de ascenso con un atributo sintetizado:

$expr \to NUM$ `{ $r{H} = $NUM{VAL} }` r `{ $expr{T} = $r{T} }`
$r \to -NUM$ `{ $r_1{H} = $r{H}." ".$NUM{VAL}." - " }` r_1 `{ $r{T} = $r_1{T} }`
$r \to \epsilon$ `{ $r{T} = $r{H} }`

es posible construir un APDR que ejecuta las acciones semánticas en los puntos indicados por el esquema de traducción. El atributo heredado se convierte en un parámetro de entrada a la subrutina asociada con la variable sintáctica:

```
sub expression() {
  my $r = $value." "; #accion intermedia
  match('NUM');
  return rest($r); # accion final $expr{T} = $r{T}
}

sub rest($) {
  my $v = shift;

  if ($lookahead eq '-') {
    match('-');
    my $r = "$v $value -"; # accion intermedia
    match('NUM');
    return rest($r); # accion final $r{t} = $r_1{t}
  }
  elsif ($lookahead ne 'EOI') {
    error("Se esperaba un operador");
  }
  else { return $v; } # r -> epsilon { $r{t} = $r{h} }
}
```

2.8.5. Ejercicio

Generalize la estrategia anterior para eliminar la recursividad por la izquierda al siguiente esquema de traducción genérico recursivo por la izquierda y con un atributo sintetizado A^s:

$$\begin{array}{ll} A \to A_1 X_1 X_2 X_3 & \{A^s = f_X(A_1^s, X_1^s, X_2^s, X_3^s)\} \\ A \to A_1 Y_1 Y_2 Y_3 & \{A^s = f_Y(A_1^s, Y_1^s, Y_2^s, Y_3^s)\} \\ A \to Z_1 Z_2 Z_3 & \{A^s = f_Z(Z_1^s, Z_2^s, Z_3^s)\} \end{array}$$

donde f_X, f_Y y f_Z son funciones cualesquiera.

2.8.6. Práctica: Eliminación de la Recursividad por la Izquierda

En esta práctica vamos a extender las fases de análisis léxico y sintáctico del compilador del lenguaje Tutu cuya gramática se definió en el ejercicio 2.5.1 con expresiones que incluyen diferencias y divisiones. Además construiremos una representación del árbol sintáctico concreto. Para ello consideremos el siguiente esquema de traducción recursivo por la izquierda (en concreto las reglas recursivas por la izquierda son las 10, 11, 13 y 14):

0	p → ds ss	`{ $p{t}  = { n => 0, ds => $ds{t}, ss => $ss{t} } }`
1	p → ss	`{ $p{t}  = { n => 1, ss => $ss{t} } }`
2	ds → d ’;’ ds	`{ $ds{t} = { n => 2, d => $d{t}, ; => ’;’, ds => $ds{t} } }`
3	ds → d ’;’	`{ $ds{t} = { n => 3, d => $d{t} ; => ’;’ } }`
4	d → INT il	`{ $d{t}  = { n => 4, INT => ’INT’, il >$il{t} } }`
5	d → STRING il	`{ $d{t}  = { n => 5, STRING => ’STRING’, il >$il{t} } }`
6	ss → s ’;’ ss	`{ $ss{t} = { n => 6, s => $s{t}, ; => ’;’ ss => $ss{t} } }`
7	ss → s	`{ $ss{t} = { n => 7, s => $s{t} } }`
8	s → ID ’=’ e	`{ $s{t}  = { n => 8, ID => $ID{v}, = => ’=’, e => $e{t} } }`
9	s → P e	`{ $s{t}  = { n => 9, P => ’P’, e => $e{t} } }`
10	e → e1 ’+’ t	`{ $e{t}  = { n => 10, e => $e1{t}, + => ’+’, t => $t{t} } }`
11	e → e1 ’-’ t	`{ $e{t}  = { n => 11, e => $e1{t}, - => ’-’, t => $t{t} } }`
12	e → t	`{ $e{t}  = { n => 12, t => $t{t} } }`
13	t → t1 ’*’ f	`{ $t{t}  = { n => 13, t => $t1{t}, * => ’*’, f => $f{t} } }`
14	t → t ’/’ f	`{ $t{t}  = { n => 14, t => $t1{t}, / => ’/’, f => $f{t} } }`
15	t → f	`{ $t{t}  = { n => 15, f => $f{t} } }`
16	f → ’(’ e ’)’	`{ $f{t}  = { n => 16, ( => ’(’, e => $e{t}, ) => ’)’ } }`
17	f → ID	`{ $f{t}  = { n => 17, ID => $ID{v} } }`
18	f → NUM	`{ $f{t}  = { n => 18, NUM => $NUM{v} } }`
19	f → STR	`{ $f{t}  = { n => 19, STR => $STR{v} } }`
20	il → ID ’,’ il	`{ $il{t} = { n => 20, ID => $ID{v}, ’,’ => ’,’, il => $il{t} } }`
21	il → ID	`{ $il{t} = { n => 21, ID => $ID{v} } }`
22	s → ϵ	`{ $s{t} = { n => 22, s => ’’ }}`

Por razones de espacio hemos abreviado los nombres de las variables. El atributo `t` (por *tree*) es una referencia a un hash. La entrada `n` contiene el número de la regla en juego. Hay una entrada por símbolo en la parte derecha. El atributo `v` de `ID` es la cadena asociada con el identificador. El atributo `v` de `NUM` es el valor numérico asociado con el terminal. Se trata de, siguiendo la metodología explicada en la sección anterior, construir un analizador descendente predictivo recursivo que sea equivalente al esquema anterior. Elimine la recursión por la izquierda. Traslade las acciones a los lugares convenientes en el nuevo esquema e introduzca los atributos heredados que sean necesarios. Genere pruebas siguiendo la metodología explicada en la sección 2.4.1. ¡Note que el árbol que debe producir es el de la gramática inicial, ¡No el de la gramática transformada!

2.9. Árbol de Análisis Abstracto

2.9.1. Lenguajes Árbol y Gramáticas Árbol

Un *árbol de análisis abstracto* (denotado *AAA*, en inglés *abstract syntax tree* o *AST*) porta la misma información que el árbol de análisis sintáctico pero de forma mas condensada, eliminándose terminales y producciones que no aportan información.

Definición 2.9.1. *Un* alfabeto con función de aridad *es un par* (Σ, ρ) *donde* Σ *es un conjunto finito y* ρ *es una función* $\rho : \Sigma \rightarrow \mathbb{N}_0$, *denominada* función de aridad. *Denotamos por* $\Sigma_k = \{a \in \Sigma :\ \rho(a) = k\}$.

Definimos el lenguaje árbol homogéneo $B(\Sigma)$ *sobre* Σ *inductivamente:*

- *Todos los elementos de aridad 0 están en* $B(\Sigma)$*:* $a \in \Sigma_0$ *implica* $a \in B(\Sigma)$
- *Si* $b_1, \ldots, b_k \in B(\Sigma)$ *y* $f \in \Sigma_k$ *es un elemento k-ario, entonces* $f(b_1, \ldots, b_k) \in B(\Sigma)$

Los elementos de $B(\Sigma)$ *se llaman* árboles *o* términos.

Ejemplo 2.9.1. *Sea* $\Sigma = \{A, CONS, NIL\}$ *con* $\rho(A) = \rho(NIL) = 0, \rho(CONS) = 2$. *Entonces* $B(\Sigma) = \{A, NIL, CONS(A, NIL), CONS(NIL, A), CONS(A, A), CONS(NIL, NIL), \ldots\}$

Ejemplo 2.9.2. *Una versión simplificada del alfabeto con aridad en el que estan basados los árboles construidos por el compilador de Tutu es:*

$\Sigma = \{ID, NUM, LEFTVALUE, STR, PLUS, TIMES, ASSIGN, PRINT\}$
$\rho(ID) = \rho(NUM) = \rho(LEFTVALUE) = \rho(STR) = 0$
$\rho(PRINT) = 1$
$\rho(PLUS) = \rho(TIMES) = \rho(ASSIGN) = 2.$

Observe que los elementos en $B(\Sigma)$ no necesariamente son árboles "correctos". Por ejemplo, el árbol $ASSIGN(NUM, PRINT(ID))$ es un elemento de $B(\Sigma)$.

Definición 2.9.2. *Una* gramática árbol regular *es una cuadrupla $((\Sigma, \rho), N, P, S)$, donde:*

- *(Σ, ρ) es un alfabeto con aricidad $\rho : \Sigma \to \mathbb{N}$*
- *N es un conjunto finito de variables sintácticas o no terminales*
- *P es un conjunto finito de reglas de producción de la forma $X \to s$ con $X \in N$ y $s \in B(\Sigma \cup N)$*
- *$S \in N$ es la variable o símbolo de arranque*

Definición 2.9.3. *Dada una gramática (Σ, N, P, S), se dice que un árbol $t \in B(\Sigma \cup N)$ es del tipo $(X_1, \ldots X_k)$ si el j-ésimo noterminal, contando desde la izquierda, que aparece en t es $X_j \in N$.*

Si $p = X \to s$ es una producción y s es de tipo $(X_1, \ldots X_n)$, diremos que la producción p es de tipo $(X_1, \ldots X_n) \to X$.

Definición 2.9.4. *Consideremos un árbol $t \in B(\Sigma \cup N)$ que sea del tipo $(X_1, \ldots X_n)$, esto es las variables sintácticas en el árbol leídas de izquierda a derecha son $(X_1, \ldots X_n)$.*

- *Si $X_i \to s_i \in P$ para algún i, entonces decimos que el árbol t* deriva en un paso en el árbol t' *resultante de sustituir el nodo X_i por el árbol s_i y escribiremos $t \Longrightarrow t'$. Esto es, $t' = t\{X_i/s_i\}$*
- *Todo árbol deriva en cero pasos en si mismo $t \overset{0}{\Longrightarrow} t$.*
- *Decimos que un árbol t deriva en n pasos en el árbol t' y escribimos $t \overset{n}{\Longrightarrow} t'$ si t deriva en un paso en un árbol t'' el cuál deriva en $n-1$ pasos en t'. En general, si t deriva en un cierto número de pasos en t' escribiremos $t \overset{*}{\Longrightarrow} t'$.*

Definición 2.9.5. *Se define el* lenguaje árbol generado por una gramática $G = (\Sigma, N, P, S)$ *como el lenguaje $L(G) = \{t \in B(\Sigma) : \exists S \overset{*}{\Longrightarrow} t\}$.*

Ejemplo 2.9.3. *Sea $G = (\Sigma, V, P, S)$ con $\Sigma = \{A, CONS, NIL\}$ y $\rho(A) = \rho(NIL) = 0, \rho(CONS) = 2$ y sea $V = \{E, L\}$. El conjunto de producciones P es:*

$$P_1 = \{L \to NIL, L \to CONS(E, L), E \to a\}$$

La producción $L \to CONS(E, L)$ es del tipo $(E, L) \to L$.

Informalmente, el lenguaje generado por G se obtiene realizando sustituciones sucesivas (derivando) desde el símbolo de arranque hasta producir un árbol cuyos nodos estén etiquetados con elementos de Σ. Debería ser claro que, en este ejemplo, $L(G)$ es el conjunto de las listas en A, incluyendo la lista vacía:

$$L(G) = \{NIL, CONS(A, NIL), CONS(A, CONS(A, NIL)), \ldots\}$$

Ejercicio 2.9.1. *Construya una derivación para el árbol $CONS(A, CONS(A, NIL))$. ¿De que tipo es el árbol $CONS(E, CONS(A, CONS(E, L)))$?.*

Cuando hablamos del AAA producido por un analizador sintáctico, estamos en realidad hablando de un lenguaje árbol cuya definición precisa debe hacerse a través de una gramática árbol regular. Mediante las gramáticas árbol regulares disponemos de un mecanismo para describir formalmente el lenguaje de los AAA que producirá el analizador sintáctico para las sentencias Tutu.

Ejemplo 2.9.4. *Sea $G = (\Sigma, V, P, S)$ con*

$$\begin{array}{l}
\Sigma = \{ID, NUM, LEFTVALUE, STR, PLUS, TIMES, ASSIGN, PRINT\} \\
\rho(ID) = \rho(NUM) = \rho(LEFTVALUE) = \rho(STR) = 0 \\
\rho(PRINT) = 1 \\
\rho(PLUS) = \rho(TIMES) = \rho(ASSIGN) = 2 \\
V = \{st, expr\}
\end{array}$$

y las producciones:

$$\begin{array}{lll}
P = & \{ & \\
 & st & \rightarrow ASSIGN(LEFTVALUE, expr) \\
 & st & \rightarrow PRINT(expr) \\
 & expr & \rightarrow PLUS(expr, expr) \\
 & expr & \rightarrow TIMES(expr, expr) \\
 & expr & \rightarrow NUM \\
 & expr & \rightarrow ID \\
 & expr & \rightarrow STR \\
 & \} &
\end{array}$$

Entonces el lenguaje $L(G)$ contiene árboles como el siguiente:

$$\begin{array}{llll}
ASSIGN & (& & \\
 & LEFTVALUE, & & \\
 & PLUS & (& \\
 & & ID, & \\
 & & TIMES & (\\
 & & & NUM, \\
 & & & ID \\
 & & &) \\
 & &) & \\
 &) & &
\end{array}$$

El cual podría corresponderse con una sentencia como `a = b + 4 * c`.

El lenguaje de árboles descrito por esta gramática árbol es el lenguaje de los AAA de las sentencias de Tutu.

Ejercicio 2.9.2. *Redefina el concepto de árbol de análisis concreto dado en la definición 3.1.7 utilizando el concepto de gramática árbol. Con mas precisión, dada una gramática $G = (\Sigma, V, P, S)$ defina una gramática árbol $T = (\Omega, N, R, U)$ tal que $L(T)$ sea el lenguaje de los árboles concretos de G. Puesto que las partes derechas de las reglas de producción de P pueden ser de distinta longitud, existe un problema con la aricidad de los elementos de Ω. Discuta posibles soluciones.*

Ejercicio 2.9.3. *¿Cómo son los árboles sintácticos en las derivaciones árbol? Dibuje varios árboles sintácticos para las gramáticas introducidas en los ejemplos 2.9.3 y 2.9.4.*

Intente dar una definición formal del concepto de árbol de análisis sintáctico asociado con una derivación en una gramática árbol

Definición 2.9.6. *La notación de Dewey es una forma de especificar los subárboles de un árbol $t \in B(\Sigma)$. La notación sigue el mismo esquema que la numeración de secciones en un texto: es una palabra formada por números separados por puntos. Así t/2.1.3 denota al tercer hijo del primer hijo del segundo hijo del árbol t. La definición formal sería:*

- $t/\epsilon = t$
- *Si $t = a(t_1, \ldots t_k)$ y $j \in \{1 \ldots k\}$ y n es una cadena de números y puntos, se define inductivamente el subárbol $t/j.n$ como el subárbol n-ésimo del j-ésimo subárbol de t. Esto es: $t/j.n = t_j/n$*

Ejercicio 2.9.4. *Sea el árbol:*

$$\begin{array}{llll}
t = ASSIGN & (& & \\
& LEFTVALUE, & & \\
& PLUS & (& \\
& & ID, & \\
& & TIMES & (\\
& & & NUM, \\
& & & ID \\
& & &) \\
& &) & \\
&) & &
\end{array}$$

Calcule los subárboles t/ϵ, $t/2.2.1$, $t/2.1$ y $t/2.1.2$.

2.9.2. Realización del AAA para Tutu en Perl

En la sección 2.6.1 nos limitamos a realizar un recorrido del árbol de análisis sintáctico concreto. En esta sección construimos un *árbol de análisis sintáctico abstracto*. Este proceso puede verse como *la traducción desde el lenguaje de árboles concretos hasta el lenguaje de árboles abstractos.*

Definición 2.9.7. *La gramática árbol extendida que especifica los árboles AAA para el compilador de Tutu es esta:*

$$\begin{array}{r|ll}
1 & prog & \rightarrow PROGRAM(decls, sts) \\
2 & decls & \rightarrow \text{list } decl \\
3 & sts & \rightarrow \text{list } st \\
4 & decl & \rightarrow INT(idlist) \\
5 & decl & \rightarrow STRING(idlist) \\
6 & idlist & \rightarrow \text{list } SIMPLEID \\
7 & st & \rightarrow ASSIGN(LEFTVALUE, expr) \\
8 & st & \rightarrow PRINT(expr) \\
9 & expr & \rightarrow PLUS(expr, expr) \\
10 & expr & \rightarrow TIMES(expr, expr) \\
11 & expr & \rightarrow NUM \\
12 & expr & \rightarrow ID \\
13 & expr & \rightarrow STR
\end{array}$$

Hemos extendido el concepto de gramática árbol con el concepto de *lista de no terminales*. A la hora de construir las estructuras de datos las listas de variables se van a traducir por listas de árboles.

Por ejemplo, un árbol abstracto para el programa

```
int a,b;
string c, d;
a = 4;
p a
```

Sería de la forma:

```
PROGRAM(
        DECLS[INT[ID, ID], STRING[ID, ID]],
        STS[ASSIGN(LEFTVALUE, NUM), PRINT(ID)]
      )
```

Donde los corchetes indican listas y los paréntesis tuplas.

Para llevar a cabo la traducción deberemos tomar decisiones sobre que forma de representación nos conviene. Cada nodo del AAA va a ser un objeto y la clase indicará si es un nodo suma, producto, una declaración, una asignación, etc.

Cada nodo del árbol AAA va a ser un objeto. De este modo el acceso a los atributos del nodo se hará a través de los métodos asociados. Además, el procedimiento de traducción al lenguaje objetivo depende del tipo de nodo. Así por ejemplo, el método *traducción* es diferente para un nodo de tipo $PLUS$ que para otro de tipo $ASSIGN$.

Resumamos antes de entrar en detalle, la forma de manejar los objetos en Perl:

- Para crear una *clase* se construye un “package”:

  ```
  package NUM;
  ```

- Para crea un *método* se escribe una subrutina:

  ```
  package NUM;
  sub incr { my $self = shift; $self->{VAL}++ }
  ```

  el primer argumento de un método suele ser la referencia al objeto en cuestión.

- Para crear un *objeto*, se bendice (“bless”) una referencia. Los objetos Perl son datos normales como hashes y arrays que han sido “bendecidos” en un paquete. Por ejemplo:

  ```
  my $a = bless {VAL => 4}, 'NUM';
  ```

  crea un objeto referenciado por `$a` que pertenece a la clase `NUM`. Los métodos del objeto son las subrutinas que aparecen en el `package NUM`.

- Para referirse a un método de un objeto se usa la sintáxis “flecha”:

  ```
  $a->incr;
  ```

  Cuando se usa la sintáxis flecha, el primer argumento de la rutina es la referencia al objeto, esto es, la llamada anterior es equivalente a `NUM::incr($a)`

- Constructores: En Perl son rutinas que retornan una referencia a un objeto recién creado e inicializado

  ```
  sub new { my ($class, $value) = @_; return bless {VAL => $value}, $class; }
  ```

  Normalmente se llaman usando la sintáxis flecha, pero a la izquierda de la flecha va el nombre de la clase. Por ejemplo:

  ```
  my $a = NUM->new(4)
  ```

  En este caso, el primer argumento es el nombre de la clase. La llamada anterior es equivalente a

  ```
  NUM::new('NUM', 4)
  ```

Volviendo a nuestro problema de crear el AAA, para crear los objetos de las diferentes clases de nodos usaremos el módulo `Class::MakeMethods::Emulator::MethodMaker` (véase la línea 9):

```
1 package PL::Syntax::Analysis;
2
3 use 5.006;
4 use strict;
5 use warnings;
6 use Data::Dumper;
7 use IO::File;
8 use Class::MakeMethods::Emulator::MethodMaker '-sugar';
9
```

```
10 require Exporter;
11 our @ISA = qw(Exporter);
12 our @EXPORT = qw( );
13 our $VERSION = '0.02';
14
15 #######################################################
16
17 # Grammar:
18 # P : DD L      | L
19 # DD: D ';' DD  | D ';'
20 # D : int I     | string I
21 # L : S         | S ; L
22 # S : ID '=' E  | p E  | epsilon
23 # E : T '+' E   | T
24 # T : F '*' T   | F
25 # F : '(' E ')' | id | num | str
26 # I : id ',' I  | id
```

Hemos aislado la fase de análisis sintáctica en un módulo aparte denominado `PL::Syntax::Analysis`. La dependencia se actualiza en `Makefile.PL`:

```
PL0506/04sintactico/PL-Tutu$ cat -n Makefile.PL
  1  use 5.008004;
  2  use ExtUtils::MakeMaker;
  3  WriteMakefile(
  4      NAME              => 'PL::Tutu',
  5      VERSION_FROM      => 'lib/PL/Tutu.pm', # finds $VERSION
  6      PREREQ_PM         => {Class::MakeMethods::Emulator::MethodMaker => 0},1
  7      EXE_FILES         => [ 'scripts/tutu.pl', 'scripts/tutu' ],
  8      ($] >= 5.005 ?     ## Add these new keywords supported since 5.005
  9        (ABSTRACT_FROM  => 'lib/PL/Tutu.pm', # retrieve abstract from module
 10         AUTHOR         => 'Casiano Rodriguez Leon <casiano@ull.es>') : ()),
 11  );
```

Se actualiza también MANIFEST:

```
$ cat -n MANIFEST
 1  Changes
 2  lib/PL/Error.pm
 3  lib/PL/Lexical/Analysis.pm
 4  lib/PL/Syntax/Analysis.pm
 5  lib/PL/Tutu.pm
 6  Makefile.PL
 7  MANIFEST
 8  MANIFEST.SKIP
 9  README
10  scripts/test01.tutu
11  scripts/tutu
12  scripts/tutu.pl
13  t/01Lexical.t
```

Ahora `compile` llama a `Syntax::Analysis::parser` pasándole como argumento la lista de terminales `@tokens`. La función `parser` devuelve el AAA:

```
04sintactico/PL-Tutu/lib/PL$ sed -ne '/sub compile\>/,/^}/p' Tutu.pm | cat -n
  1  sub compile {
```

```
  my ($input) = @_;
  #my %symbol_table = ();
  #my $data = ""; # Contiene todas las cadenas en el programa fuente
  my $target = ""; # target code
  my @tokens = ();
  my $tree = undef; # abstract syntax tree
  #my $global_address = 0;


  ########lexical analysis
  @tokens = scanner($input);
  #print "@tokens\n";

  ########syntax (and semantic) analysis
  $tree = &Syntax::Analysis::parser(@tokens);
  print Dumper($tree);

  ########machine independent optimizations
  &Machine::Independent::Optimization::Optimize;

  ########code generation
  &Code::Generation::code_generator;

  ########peephole optimization
  &Peephole::Optimization::transform($target);

  return \$target;
}
```

El módulo `Class::MakeMethods::Emulator::MethodMaker` permite crear constructores y métodos de acceso. El módulo no viene con la distribución de Perl, así que, en general, deberá descargarlo desde CPAN e instalarlo. Así definimos que existe una clase de nodos `TYPE` que nuestro AAA va a tener:

```
package TYPE;
make methods
  get_set       => [ 'NAME', 'LENGTH' ],
  new_hash_init => 'new';
```

El uso de los argumentos `get_set => [ 'NAME', 'LENGTH' ]` hace que se cree un objeto de tipo hash con claves `'NAME'` y `'LENGTH'` así como métodos `NAME` y `LENGTH` que cuando se llaman con un argumento devuelven el valor y cuando se llaman con dos argumentos modifican el valor correspondiente. La clave `get_set` produce métodos de acceso y modificación de los atributos del objeto que tienen la forma:

```
sub TYPE::NAME {
  my ($self, $new) = @_;
  defined($new) and $self->{NAME} = $new;
  return $self->{NAME};
}
```

Asi mismo el uso de `new_hash_init => 'new'` genera un constructor cuyo nombre será `'new'` y que cuando es llamado inicializará el objeto con los argumentos con nombre especificados en la llamada. El constructor construído (vaya retruécano) cuando se usa la clave `new_hash_init` tiene el siguiente aspecto:

```
sub TYPE::new {
  my ($class, %args) = @_;
  my $self = {};

  bless $self, $class;
  foreach my $attribute (keys %args) {
    $self->$attribute($args{$attribute});
  }
  return $self;
}
```

Ahora podemos crear objetos de la clase `TYPE` haciendo:

```
my $int_type = TYPE->new(NAME => 'INTEGER', LENGTH => 1);
my $string_type = TYPE->new(NAME => 'STRING', LENGTH => 2);
my $err_type = TYPE->new(NAME => 'ERROR', LENGTH => 0);
```

Cada uno de estos objetos es un hash con las correspondientes claves para el nombre y el tipo.

Otros tipos de nodos del AAA son:

```
package PROGRAM; # raíz del AAA
make methods
  get_set       => [ 'DECLS', 'STS' ],
  new_hash_init => 'new';

package STRING; # tipo
make methods
  get_set       => [ 'TYPE', 'IDLIST' ],
  new_hash_init => 'new';

package INT; # tipo
make methods
  get_set       => [ 'TYPE', 'IDLIST' ],
  new_hash_init => 'new';

package ASSIGN; #sentencia
make methods
  get_set       => [ 'LEFT', 'RIGHT' ],
  new_hash_init => 'new';

package PRINT; #sentencia
make methods
  get_set       => [ 'EXPRESSION' ],
  new_hash_init => 'new';

package NUM; # para los números
make methods
  get_set       => [ 'VAL', 'TYPE' ],
  new_hash_init => 'new';

package ID; # Nodos identificador. Parte derecha
make methods
  get_set       => [ 'VAL', 'TYPE' ],
  new_hash_init => 'new';
```

```
package STR; # Clase para las constantes cadena
make methods
  get_set       => [ 'OFFSET', 'LENGTH', 'TYPE' ],
  new_hash_init => 'new';

package PLUS; # Nodo suma
make methods
  get_set       => [ 'LEFT', 'RIGHT', 'TYPE' ],
  new_hash_init => 'new';

package TIMES;
make methods
  get_set       => [ 'LEFT', 'RIGHT', 'TYPE' ],
  new_hash_init => 'new';

package LEFTVALUE; # Identificador en la parte izquierda
make methods       # de una asignación
  get_set       => [ 'VAL', 'TYPE' ],
  new_hash_init => 'new';
```

Hemos extendido el concepto de gramática árbol con el concepto de *lista de no terminales*. A la hora de construir las estructuras de datos las listas de variables se van a traducir por listas de árboles. Los tipos de nodos ($ASSIGN$, $PRINT$, ...) se traducen en nombres de clases. Hemos hecho una excepción con $SIMPLEID$ el cual es simplemente una variable cadena conteniendo el identificador correspondiente.

El siguiente esquema de traducción resume la idea para una gramática simplificada: cada vez que encontremos un nodo en el árbol sintáctico concreto con una operación crearemos un nodo en el AAA cuya clase viene definida por el tipo de operación. Para los terminales creamos igualmente nodos indicando de que clase de terminal se trata. El atributo nodo lo denotaremos por `n`:

$e \rightarrow e_1 + f$ `{ $e{n} = PLUS->new(LEFT=>$e_1{n}, RIGHT=>$f{n}) }`
$f \rightarrow NUM$ `{ $f{n} = NUM->new(VAL => $NUM{VAL}) }`
$f \rightarrow ID$ `{ $f{n} = ID->new(VAL => $ID{VAL}) }`

La estructura de cada rutina sigue siendo la misma, sólo que ampliada con las acciones para la construcción de los correspondientes nodos. Veamos por ejemplo, como modificamos la subrutina factor:

```
sub factor() {
  my ($e, $id, $str, $num);

  if ($lookahead eq 'NUM') {
    $num = $value;
    match('NUM');
    return NUM->new(VAL => $num, TYPE => $int_type);
  }
  elsif ($lookahead eq 'ID') {
    $id = $value;
    match('ID');
    return ID->new( VAL => $id, TYPE => undef);
  }
  elsif ($lookahead eq 'STR') {
    $str = $value;
    match('STR');
    return STR->new(OFFSET => undef, LENGTH => undef, TYPE => $string_type);
```

```
  }
  elsif ($lookahead eq '(') {
    match('(');
    $e = expression;
    match(')');
    return $e;
  }
  else {
    Error::fatal("Se esperaba (, NUM o ID");
  }
}
```

2.9.3. AAA: Otros tipos de nodos

Hemos optado por que las rutinas asociadas a variables sintácticas que describen listas de subcategorías devuelvan las correspondientes listas de nodos. Teníamos tres variables tipo lista. Las reglas para las listas eran:

Gramática de los Árboles de Tutu	Gramática del lenguaje Tutu
decls → *list decl*	declarations → declaration ';' declarations | declaration ';'
sts → *list st*	statements → statement ';' statements | statement
idlist → *list SIMPLEID*	idlist → ID ',' idlist | ID

En este caso las subrutinas asociadas no devuelven objetos sino listas de objetos. Esto da lugar a una compactación del AAA. Veánse los códigos de `statements` y `idlist`:

```
sub statements() {
  my @s;

  @s = (statement());
  if ($lookahead eq ';') {
    match(';');
    push @s, statements();
  }
  return @s;
}

sub idlist() {
  my @id;

  if ($lookahead eq 'ID') {
    @id = ($value); # no es un objeto
    match('ID');
    if ($lookahead eq ',') {
      match(',');
      push @id, idlist();
    }
  }
  else {
    Error::fatal('Se esperaba un identificador');
    @id = ('ERROR');
  }
  return @id;
}
```

2.9.4. Declaraciones

Los nodos del tipo declaration no existen propiamente, son nodos de la clase INT o de la clase $STRING$. La parte de la gramática árbol de la que hablamos es:

2	*decls*	→ list *decl*
4	*decl*	→ $INT(idlist)$
5	*decl*	→ $STRING(idlist)$
6	*idlist*	→ list $SIMPLEID$

Los nodos declaration son un hash con una clave `TYPE` la cual apunta a la estructura de datos/objeto describiendo el tipo. La otra clave del hash `IDLIST` apunta a una lista de identificadores. Los elementos de esta lista son simples identificadores (identificados en la gramática árbol anterior como $SIMPLEID$ y no como objetos ID). La parte de la gramática implicada en las declaraciones es:

> declaration → INT idlist | STRING idlist
> idlist → ID ',' idlist | ID

Así pues, el código construye un nodo de la clase `INT` o `STRING` según sea el caso.

```
sub declaration() {
  my ($t, $class, @il);

  if (($lookahead eq 'INT') or ($lookahead eq 'STRING')) {
    $class = $lookahead;
    $t = &type();
    @il = &idlist();
    return $class->new(TYPE => $t, IDLIST => \@il);
  }
  else {
    Error::fatal('Se esperaba un tipo');
  }
}
```

Observe la llamada `$class->new(TYPE => $t, IDLIST => \@il)` en la cual la clase se usa a través de una referencia simbólica.

2.9.5. Práctica: Arbol de Análisis Abstracto

Complete la fase de análisis sintáctico para la gramática de Tutu extendida con sentencias de bloque (vea las reglas 1,2,3 y 11) construyendo el AAA según el lenguaje árbol especificado por una gramática árbol que extienda la dada en la definición 2.9.7. Genere pruebas, usando `make test` para comprobar el correcto funcionamiento de su analizador sobre las mismas. Utilize el módulo `Data::Dumper` para volcar las estructuras de datos resultantes.

1	p → b
2	b → ds ss
3	b → ss
4	ds → d ’;’ ds
5	ds → d ’;’
6	d → INT il
7	d → STRING il
8	ss → s ’;’ ss
9	ss → s
10	s → ID = e
11	s → ’{’ b ’}’
12	s → P e
13	s → ϵ
14	e → e1 ’+’ t
15	e → e1 ’-’ t
16	e → t
17	t → t1 ’*’ f
18	t → t ’/’ f
19	t → f
20	f → ’(’ e ’)’
21	f → ID
22	f → NUM
23	f → STR
24	il → ID ’,’ il
25	il → ID

2.10. Análisis Semántico

Hay quien dice que el análisis semántico es la determinación de aquellas propiedades que, siendo dependientes del contexto, pueden ser computadas estáticamente en tiempo de compilación para cualquier programa correcto. Entre estas propiedades están: la comprobación de que las variables son declaradas, la compatibilidad de tipos en las expresiones, el correcto uso de las llamadas a función asi como el ámbito y visibilidad de las variables. La fase de análisis semántico puede verse como una fase de “adornado” o “etiquetado” del AAA, en la cual los atributos de los nodos del AAA son computados.

Aunque la veamos como una fase separada del análisis sintáctico, puede en numerosas ocasiones llevarse a cabo al mismo tiempo que se construye el árbol. Así lo hacemos en este ejemplo: incrustamos la acción semántica en la correspondiente rutina de análisis sintáctico. Así, en la rutina `term`, una vez que hemos obtenido los dos operandos, comprobamos que son de tipo numérico llamando (línea 8) a

`Semantic::Analysis::check_type_numeric_operator`:

Observe como aparece un nuevo atributo `TYPE` decorando el nodo creado (línea 9):

```
 1 sub term() {
 2   my ($t, $t2);
 3
 4   $t = factor;
 5   if ($lookahead eq ’*’) {
 6     match(’*’);
 7     $t2 = term;
 8     my $type = Semantic::Analysis::check_type_numeric_operator($t, $t2, ’*’);
 9     $t = TIMES->new( LEFT => $t, RIGHT => $t2, TYPE => $type);
10   }
11   return $t;
12 }
```

En el manejo de errores de tipo, un tipo especial `$err_type` es usado para indicar un error de tipo:

```
sub check_type_numeric_operator {
  my ($op1, $op2, $operator) = @_;

  my $type = numeric_compatibility($op1, $op2, $operator);
  if ($type == $err_type) {
    Error::fatal("Operandos incompatibles para el operador $operator")
  }
  else {
    return $type;
  }
}
```

La subrutina `numeric_compatibility` comprueba que los dos operandos son de tipo numérico y devuelve el correspondiente tipo. Si ha ocurrido un error de tipo, intenta encontrar un tipo conveniente para el operando:

```
sub numeric_compatibility {
  my ($op1, $op2, $operator) = @_;

  if (($op1->TYPE == $op2->TYPE) and is_numeric($op1->TYPE)) {
    return $op1->TYPE; # correct
  }
  ... # código de recuperación de errores de tipo
}

sub is_numeric {
  my $type = shift;

  return ($type == $int_type); # añadir long, float, double, etc.
}
```

Es parte del análisis semántico la declaración de tipos:

```
sub declaration() {
  my ($t, $class, @il);

  if (($lookahead eq 'INT') or ($lookahead eq 'STRING')) {
    $class = $lookahead;
    $t = &type();
    @il = &idlist();
    &Semantic::Analysis::set_types($t, @il);
    &Address::Assignment::compute_address($t, @il);
    return $class->new(TYPE => $t, IDLIST => \@il);
  }
  else {
    Error::fatal('Se esperaba un tipo');
  }
}
```

Para ello se utiliza una tabla de símbolos que es un hash `%symbol_table` indexado en los identificadores del programa:

```
sub set_types {
  my $type = shift;
  my @vars = @_;

  foreach my $var (@vars) {
    if (!exists($symbol_table{$id})) { $symbol_table{$var}->{TYPE} = $type; }
    else { Error::error("$id declarado dos veces en el mismo ámbito"); }
  }
}
```

Cada vez que aparece una variable en el código, bien en un factor o en una asignación, comprobamos que ha sido declarada:

```
sub factor() {
  my ($e, $id, $str, $num);

  if ($lookahead eq 'NUM') { ... }
  elsif ($lookahead eq 'ID') {
    $id = $value;
    match('ID');
    my $type = Semantic::Analysis::check_declared($id);
    return ID->new( VAL => $id, TYPE => $type);
  }
  elsif ($lookahead eq 'STR') { ... }
  elsif ($lookahead eq '(') { ... }
  else { Error::fatal("Se esperaba (, NUM o ID"); }
}
```

La función `check_declared` devuelve el atributo `TYPE` de la correspondiente entrada en la tabla de símbolos.

```
sub check_declared {
  my $id = shift;

  if (!exists($symbol_table{$id})) {
    Error::error("$id no ha sido declarado!");
    # auto-declaración de la variable a err_type
    Semantic::Analysis::set_types($err_type, ($id));
  }
  return $symbol_table{$id}->{TYPE};
}
```

2.10.1. Práctica: Declaraciones Automáticas

Modifique la subrutina `check_declared` para que cuando una variable no haya sido declarada se declare “sobre la marcha”. ¿Puede utilizar información dependiente del contexto para decidir cual es la mejor forma de declararla?

2.10.2. Práctica: Análisis Semántico

Extienda el código de la práctica 2.9.5 para comprobar la compatibilidad de tipos.

1	p → b
2	b → ds ss
3	b → ss
4	ds → d ’;’ ds
5	ds → d ’;’
6	d → INT il
7	d → STRING il
8	ss → s ’;’ ss
9	ss → s
10	s → ID ’=’ e
11	s → ’{’ b ’}’
12	s → P e
13	s → ϵ
14	e → e1 ’+’ t
15	e → e1 ’-’ t
16	e → t
17	t → t1 ’*’ f
18	t → t ’/’ f
19	t → f
20	f → ’(’ e ’)’
21	f → ID
22	f → NUM
23	f → STR
24	il → ID ’,’ il
25	il → ID

En cuanto a las sentencias de bloque, se pretende que el ámbito y visibilidad de las variables sea como en el lenguaje C, esto es, las declaraciones mas internas con el mismo identificador ocultan las mas externas. Así:

```
int a;
a = 4;
{
  int a;
  a = 5;
  p a
}; /* el ; es necesario */
p a
```

Imprimiría 5 y 4. Para traducir esta sentencia es necesario *usar una lista/pila de referencias a tablas de símbolos. Cada sentencia compuesta o bloque tendrá su propia tabla de símbolos.* Los identificadores se búscan en la lista de referencias a tablas de símbolos, primero en la última tabla de símbolos insertada y sino se encuentra se busca en la penúltima insertada, etc.

Guarde como un atributo del identificador ($SYMTABLE$) la referencia a la tabla de símbolos a la que pertenece. Guarde como un atributo del nodo bloque ($BLOCK$) la referencia a la tabla de símbolos asociada.

2.11. Optimización Independiente de la Máquina

En esta fase se hace un análisis del árbol, sometiéndolo a transformaciones que aumenten la eficiencia del código final producido.

Ejemplos de tareas que se pueden llevar a cabo en esta fase son:

- Extracción del interior de un bucle de cálculos que son invariantes del bucle
- Plegado de constantes: computar las expresiones constantes en tiempo de compilación, no de ejecución

- Propagación de las constantes: si se sabe que una variable en un punto del programa tiene un valor constante `a = 4`, se puede sustituir su uso por el de la constante
- Eliminación de computaciones redundantes, cuando la misma expresión aparece repetidas veces con los mismos valores de las variables
- La eliminación de código “muerto”: código que se sabe que nunca podrá ser ejecutado

En nuestro primer ejemplo, reduciremos esta fase a realizar una tarea de plegado de las constantes. Primero lo haremos mediante la rutina

`&Machine::Independent::Optimization::fold`

En esta fase transformamos los AAA: si tenemos un árbol de la forma OPERATION(left, right), esto es, su raíz es una operación, primero plegamos los subárboles left y right, y si se han transformado en constantes numéricas, entonces plegamos el nodo que pasa a ser numérico:

```
 1 sub operator_fold { # Obsérvese el uso del aliasing
 2
 3   if ($_[0]->LEFT->is_operation) {
 4     $_[0]->{LEFT}->fold;
 5   }
 6   if ($_[0]->RIGHT->is_operation) {
 7     $_[0]->{RIGHT}->fold;
 8   }
 9   if (ref($_[0]->LEFT) eq "NUM" and ref($_[0]->RIGHT) eq "NUM") {
10     $_[0] = reduce_children($_[0]);
11   }
12 }
13
14 sub PLUS::fold {
15  operator_fold(@_);
16 }
17
18 sub TIMES::fold {
19  operator_fold(@_);
20 }
```

El plegado de las operaciones binarias se ha delegado en la subrutina `operator_fold`. En las líneas 3 y 6 se comprueba que se trata de un nodo de tipo operación. Si es así se procede a su plegado. Una vez plegados los dos subárboles hijo comprobamos en la línea 9 que los hijos actuales son de la clase `NUM`. Si es el caso, en la línea 10 cambiamos el nodo por el resultado de operar los dos hijos. Los nodos han sido extendidos con un método `is_operation` que determina si se trata de un nodo operación binaria o no. Para ello se han introducido nuevas clases de nodos: la clase `Node` está en la raíz de la jerarquía de herencia, las clases `Leaf` y `Binary` se usan para representar los nodos hoja y binarios y heredan de la anterior. Una clase informa a Perl que desea heredar de otra clase añadiendo el nombre de esa clase a la variable `@ISA` de su paquete. La herencia en Perl determina la forma de búsqueda de un método. Si el objeto no se puede encontrar en la clase, recursivamente y en orden primero-profundo se busca en las clases de las cuales esta hereda, esto es en las clases especificadas en el vector `@ISA`.

```
package Node;

sub is_operation {
  my $node = shift;
```

```
  return ref($node) =~ /^(TIMES)|(PLUS)$/;
}

package Leaf; # hoja del AAA
our @ISA = ("Node");
sub children {
  return ();
}

package Binary;
our @ISA = ("Node");
sub children {
  my $self = shift;

  return (LEFT => $self->{LEFT}, RIGHT => $self->{RIGHT});
}
```

Así pues, los objetos de la clase `Leaf` tienen acceso al método `is_operation`.

Ahora hacemos que las clases `PLUS` y `TIMES` hereden de la clase `BINARY`:

```
package PLUS;
our @ISA = ("Binary");

sub operator {
  my $self = shift;

  $_[0]+$_[1];
}

....

package TIMES;
our @ISA = ("Binary");

sub operator {
  my $self = shift;

  $_[0]*$_[1];
}

....
```

Obsérvese que en las líneas 4 y 7 del código del plegado de nodos de operación se ha accedido directamente al dato en vez de usar el método para modificar el atributo, saltándonos lo que la buena programación orientada a objetos indica. La forma en la que esta escrito hace que, por ejemplo, `$_[0]->{LEFT}` sea modificado. Recuérdese que en Perl *los argumentos son alias de los parámetros.*

La subrutina `reduce_children` es la encargada de crear el nuevo nodo con el resultado de operar los hijos izquierdo y derecho:

```
sub reduce_children {
  my ($node) = @_;

  my $value = $node->operator($node->LEFT->VAL, $node->RIGHT->VAL);
  NUM->new(VAL => $value, TYPE => $PL::Tutu::int_type);
}
```

En la línea 4 se usa el método `operator` asociado con un nodo operación.

Plegar una sentencia de impresión es plegar la expresión a imprimir:

```
sub PRINT::fold {
  $_[0]->{EXPRESSION}->fold;
}
```

Plegar una sentencia de asignación es plegar la parte derecha de la asignación:

```
sub ASSIGN::fold {
  $_[0]->{RIGHT}->fold;
}
```

de nuevo, hemos accedido a los campos en vez de usar los métodos.

Las restantes operaciones de plegado son triviales:

```
sub ID::fold { }

sub NUM::fold { }

sub STR::fold { }
```

Por último, para plegar todas las expresiones recorremos la lista de sentencias del programa y las plegamos una a una.

```
sub fold {
  my @statements = @{$tree->{STS}};
  for my $s (@statements) {
    $s->fold;
  }
}
```

2.11.1. Práctica: Plegado de las Constantes

Complete su proyecto de compilador de Tutu con la fase de plegado de las constantes siguiendo la metodología explicada en los párrafos previos. Mejore la jerarquía de clases con una clase abstracta `Operation` que represente a los nodos que se corresponden con operaciones binarias. Defina el método abstracto `operation` en dicha clase. Un *método abstracto* es uno que, mas que proveer un servicio representa un servicio o categoría. La idea es que al definir un clase base abstracta se indica un conjunto de métodos que deberían estar definidos en todas las clases que heredan de la clase base abstracta. Es como una declaración de interfaz que indica la necesidad de definir su funcionalidad en las clases descendientes, pero que no se define en la clase base. Un método abstracto debe producir una excepción con el mensaje de error adecuado si no se ha redefinido en la clase desendiente.

Para ello use la clave `abstract` del módulo `Class::MethodMaker`. Consulte la documentación del módulo `Class::MethodMaker`. Consulte [2] para saber más sobre clases abstractas.

2.12. Patrones Árbol y Transformaciones Árbol

En la fase de optimización presentada en la sección 2.11 transformabamos el programa en su representación intermedia, como un AAA decorado, para obtener otro AAA decorado.

Una transformación de un programa puede ser descrita como un conjunto de *reglas de transformación* o *esquema de traducción árbol* sobre el árbol abstracto que representa el programa.

Antes de seguir, es conveniente que repase los conceptos en la sección 2.9.1 sobre lenguajes y gramáticas árbol.

En su forma mas sencilla, estas reglas de transformación vienen definidas por ternas $(p, e, action)$, donde la primera componente de la terna p es un *patrón árbol* que dice que árboles deben ser seleccionados. La segunda componente e dice cómo debe transformarse el árbol que casa con el patrón p.

La acción *action* indica como deben computarse los atributos del árbol transformado a partir de los atributos del árbol que casa con el patrón p. Una forma de representar este esquema sería:

$$p \Longrightarrow e \texttt{ \{ action \}}$$

Por ejemplo:

$PLUS(NUM_1, NUM_2) \Longrightarrow NUM_3$ `{ $NUM_3{VAL} = $NUM_1{VAL} + $NUM_2{VAL} }`

cuyo significado es que dondequiera que haya un nódo del AAA que case con el *patrón de entrada* $PLUS(NUM, NUM)$ deberá sustituirse el subárbol $PLUS(NUM, NUM)$ por el subárbol NUM. Al igual que en los esquemas de traducción, enumeramos las apariciones de los símbolos, para distinguirlos en la parte semántica. La acción indica como deben recomputarse los atributos para el nuevo árbol: El atributo `VAL` del árbol resultante es la suma de los atributos `VAL` de los operandos en el árbol que ha casado. La transformación se repite hasta que se produce la *normalización del árbol.*

Las reglas de "casamiento" de árboles pueden ser mas complejas, haciendo alusión a propiedades de los atributos, por ejemplo

$ASSIGN(LEFTVALUE, x)$ *and* `{ notlive($LEFTVALUE{VAL}) }` $\Longrightarrow NIL$

indica que se pueden eliminar aquellos árboles de tipo asignación en los cuáles la variable asociada con el nodo $LEFTVALUE$ no se usa posteriormente.

Otros ejemplos con variables S_1 y S_2:

$IFELSE(NUM, S_1, S_2)$ and `{ $NUM{VAL} != 0 }` $\Longrightarrow S_1$
$IFELSE(NUM, S_1, S_2)$ and `{ $NUM{VAL} == 0 }` $\Longrightarrow S_2$

Observe que en el patrón de entrada $ASSIGN(LEFTVALUE, x)$ aparece un "comodín": la variable-árbol x, que hace que el árbol patrón $ASSIGN(LEFTVALUE, x)$ case con cualquier árbol de asignación, independientemente de la forma que tenga su subárbol derecho.

Las siguientes definiciones formalizan una aproximación simplificada al significado de los conceptos *patrones árbol* y *casamiento de árboles.*

Definición 2.12.1. *Sea* (Σ, ρ) *un alfabeto con función de aridad y un conjunto (puede ser infinito) de variables* $V = \{x_1, x_2, \ldots\}$. *Las variables tienen aridad cero:*

$$\rho(x) = 0 \ \forall x \in V.$$

Un elemento de $B(V \cup \Sigma)$ *se denomina* patrón *sobre* Σ.

Definición 2.12.2. *Se dice que un patrón es un* patrón lineal *si ninguna variable se repite.*

Definición 2.12.3. *Se dice que un patrón es de tipo* $(x_1, \ldots x_k)$ *si las variables que aparecen en el patrón leidas de izquierda a derecha en el árbol son* $x_1, \ldots x_k$.

Ejemplo 2.12.1. *Sea* $\Sigma = \{A, CONS, NIL\}$ *con* $\rho(A) = \rho(NIL) = 0, \rho(CONS) = 2$ *y sea* $V = \{x\}$. *Los siguientes árboles son ejemplos de patrones sobre* Σ*:*

$$\{\ x, CONS(A, x), CONS(A, CONS(x, NIL)), \ldots\}$$

El patrón $CONS(x, CONS(x, NIL))$ *es un ejemplo de patrón no lineal. La idea es que un patrón lineal como éste "fuerza" a que los árboles* t *que casen con el patrón deben tener iguales los dos correspondientes subárboles* $t/1$ *y* $t/2.1$ *situados en las posiciones de las variables* [1]

Ejercicio 2.12.1. *Dado la gramática árbol:*

$$S \to S_1(a, S, b)$$
$$S \to S_2(NIL)$$

---

[1] Repase la notación de Dewey introducida en la definición 2.9.6

la cuál genera los árboles concretos para la gramática

$$S \to aSb \mid \epsilon$$

¿Es $S_1(a, X(NIL), b)$ un patrón árbol sobre el conjunto de variables $\{X, Y\}$? ¿Lo es $S_1(X, Y, a)$? ¿Es $S_1(X, Y, Y)$ un patrón árbol?

Ejemplo 2.12.2. *Ejemplos de patrones para el AAA definido en el ejemplo 2.9.2 para el lenguaje Tutu son:*

$$x, y, PLUS(x, y), ASSIGN(x, TIMES(y, ID)), PRINT(y) \ldots$$

considerando el conjunto de variables $V = \{x, y\}$. El patrón $ASSIGN(x, TIMES(y, ID))$ es del tipo (x, y).

Definición 2.12.4. *Una* sustitución *es una aplicación θ que asigna variables a patrones $\theta : V \to B(V \cup \Sigma)$.*

Tal función puede ser naturalmente extendida de las variables a los árboles: los nodos (hoja) etiquetados con dichas variables son sustituidos por los correspondientes subárboles.

$$\begin{array}{l} \theta : B(V \cup \Sigma) \to B(V \cup \Sigma) \\ t\theta = \left\{ \begin{array}{ll} x\theta & si\ t = x \in V \\ a(t_1\theta, \ldots, t_k\theta) & si\ t = a(t_1, \ldots, t_k) \end{array} \right. \end{array}$$

Obsérvese que, al revés de lo que es costumbre, la aplicación de la sustitución θ al patrón se escribe por detrás: $t\theta$.

También se escribe $t\theta = t\{x_1/x_1\theta, \ldots x_k/x_k\theta\}$ si las variables que aparecen en t de izquierda a derecha son $x_1, \ldots x_k$.

Ejemplo 2.12.3. *Si aplicamos la sustitución $\theta = \{x/A, y/CONS(A, NIL)\}$ al patrón $CONS(x, y)$ obtenemos el árbol $CONS(A, CONS(A, NIL))$. En efecto:*

$$CONS(x, y)\theta = CONS(x\theta, y\theta) = CONS(A, CONS(A, NIL))$$

Ejemplo 2.12.4. *Si aplicamos la sustitución $\theta = \{x/PLUS(NUM, x), y/TIMES(ID, NUM)\}$ al patrón $PLUS(x, y)$ obtenemos el árbol $PLUS(PLUS(NUM, x), TIMES(ID, NUM))$:*

$$PLUS(x, y)\theta = PLUS(x\theta, y\theta) = PLUS(PLUS(NUM, x), TIMES(ID, NUM))$$

Definición 2.12.5. *Se dice que un patrón $\tau \in B(V \cup \Sigma)$ con variables $x_1, \ldots x_k$* casa con un árbol *$t \in B(\Sigma)$ si existe una sustitución de τ que produce t, esto es, si existen $t_1, \ldots t_k \in B(\Sigma)$ tales que $t = \tau\{x_1/t_1, \ldots x_k/t_k\}$. También se dice que τ* casa con la sustitución *$\{x_1/t_1, \ldots x_k/t_k\}$.*

Ejemplo 2.12.5. *El patrón $\tau = CONS(x, NIL)$ casa con el árbol $t = CONS(CONS(A, NIL), NIL)$ y con el subárbol $t.1$. Las respectivas sustituciones son $t\{x/CONS(A, NIL)\}$ y $t.1\{x/A\}$.*

$$\begin{array}{c} t = \tau\{x/CONS(A, NIL)\} \\ t.1 = \tau\{x/A\} \end{array}$$

Ejercicio 2.12.2. *Sea $\tau = PLUS(x, y)$ y $t = TIMES(PLUS(NUM, NUM), TIMES(ID, ID))$. Calcule los subárboles t' de t y las sustituciones $\{x/t_1, y/t_2\}$ que hacen que τ case con t'.*

Por ejemplo es obvio que para el árbol raíz t/ϵ no existe sustitución posible:
$t = TIMES(PLUS(NUM, NUM), TIMES(ID, ID)) = \tau\{x/t_1, y/t_2\} = PLUS(x, y)\{x/t_1, y/t_2\}$
ya que un término con raíz $TIMES$ nunca podrá ser igual a un término con raíz $PLUS$.

El problema aquí es equivalente al de las expresiones regulares en el caso de los lenguajes lineales. En aquellos, los autómatas finitos nos proveen con un mecanismo para reconocer si una determinada cadena "casa"' o no con la expresión regular. Existe un concepto análogo, el de *autómata árbol* que resuelve el problema del "casamiento" de patrones árbol. Al igual que el concepto de autómata permite la construcción de software para la búsqueda de cadenas y su posterior modificación, el concepto de autómata árbol permite la construcción de software para la búsqueda de los subárboles que casan con un patrón árbol dado.

Estamos ahora en condiciones de plantear una segunda aproximación al problema de la optimización independiente de la máquina utilizando una subrutina que busque por aquellos árboles que queremos optimizar (en el caso del plegado los árboles de tipo operación) y los transforme adecuadamente.

La función `match_and_transform_list` recibe una lista de árboles los cuales recorre sometiéndolos a las transformaciones especificadas. La llamada para producir el plegado sería:

```
Tree::Transform::match_and_transform_list(
  NODES => $tree->{STS}, # lista de sentencias
  PATTERN => sub {
     $_[0]->is_operation and $_[0]->LEFT->isa("NUM")
     and $_[0]->RIGHT->isa("NUM")
  },
  ACTION => sub {
    $_[0] = Machine::Independent::Optimization::reduce_children($_[0])
  }
);
```

Además de la lista de nodos le pasamos como argumentos una referencia a la subrutina encargada de reconocer los patrónes árbol (clave `PATTERN`) y una referencia a la subrutina que describe la acción que se ejecutará (clave `ACTION`) sobre el árbol que ha casado. Ambas subrutinas asumen que el primer argumento que se les pasa es la referencia a la raíz del árbol que está siendo explorado.

Los métodos *isa*, *can* y *VERSION* son proporcionados por una clase especial denominada clase `UNIVERSAL`, de la cual implícitamente hereda toda clase. El método `isa` nos permite saber si una clase hereda de otra.

La subrutina `match_and_transform_list` recibe los argumentos y da valores por defecto a los mismos en el caso de que no hayan sido establecidos. Finalmente, llama a `match_and_transform` sobre cada uno de los nodos "sentencia" del programa.

```
sub match_and_transform_list {
  my %arg = @_;
  my @statements = @{$arg{NODES}} or
   Error::fatal("Internal error. match_and_transform_list ".
   "espera una lista anónima de nodos");
  local $pattern = ($arg{PATTERN} or sub { 1 });
  local @pattern_args = @{$arg{PATTERN_ARGS}} if defined $arg{PATTERN_ARGS};
  local $action = ($arg{ACTION} or sub { print ref($_[0]),"\n" });
  local @action_args = @{$arg{ACTION_ARGS}} if defined $arg{ACTION_ARGS};

  for (@statements) {
    match_and_transform($_);
  }
}
```

La subrutina `match_and_transform` utiliza el método `can` para comprobar que el nodo actual dispone de un método para calcular la lista con los hijos del nodo. Una vez transformados los subárboles del nodo actual procede a comprobar que el nodo casa con el patrón y si es el caso le aplica la acción definida:

```
package Tree::Transform;

our $pattern;
our @pattern_args;
our $action;
our @action_args;
our @statements;

sub match_and_transform {
  my $node = $_[0] or Error::fatal("Error interno. match_and_transform necesita un nodo");
  Error::fatal("Error interno. El nodo de la clase",ref($node),
             " no dispone de método 'children'") unless $node->can("children");

  my %children = $node->children;

  for my $k (keys %children) {
    $node->{$k} = match_and_transform($children{$k});
  }

  if ($pattern->($node, @pattern_args)) {
    $action->( $node, @action_args);
  }
  return $node;
}
```

Recordemos el esquema de herencia que presentamos en la sección anterior. Las clases `Leaf` y `Binary` proveen versiones del método `children`. Teníamos:

```
package Node;

sub is_operation {
  my $node = shift;

  return ref($node) =~ /^(TIMES)|(PLUS)$/;
}

package Leaf; # hoja del AAA
our @ISA = ("Node");
sub children {
  return ();
}

package Binary;
our @ISA = ("Node");
sub children {
  my $self = shift;

  return (LEFT => $self->{LEFT}, RIGHT => $self->{RIGHT});
}
```

Los objetos de la clase `Leaf` tienen acceso al método `is_operation`.

Las clases `PLUS` y `TIMES` heredan de la clase `BINARY`:

```
package PLUS;
our @ISA = ("Binary");
```

```
sub operator {
  my $self = shift;

  $_[0]+$_[1];
}

....

package TIMES;
our @ISA = ("Binary");

sub operator {
  my $self = shift;

  $_[0]*$_[1];
}

....
```

La subrutina `reduce_children` introducida en la sección 2.11 es la encargada de crear el nuevo nodo con el resultado de operar los hijos izquierdo y derecho:

```
1 sub reduce_children {
2   my ($node) = @_;
3
4   my $value = $node->operator($node->LEFT->VAL, $node->RIGHT->VAL);
5   NUM->new(VAL => $value, TYPE => $PL::Tutu::int_type);
6 }
```

En la línea 4 se usa el método `operator` asociado con un nodo operación.

2.12.1. Práctica: Casando y Transformando Árboles

Complete su proyecto para el compilador de Tutu completando las subrutinas `match_and_transform` tal y como se explicó en la sección 2.12.

Ademas del plegado de constantes use las nuevas subrutinas para aplicar simultáneamente las siguientes transformaciones algebraicas:

$$\begin{array}{lll}
PLUS(NUM, x) & \wedge\ \texttt{\{ \$NUM\{VAL\} == 0 \}} & \Longrightarrow x \\
PLUS(x, NUM) & \wedge\ \texttt{\{ \$NUM\{VAL\} == 0 \}} & \Longrightarrow x \\
TIMES(NUM, x) & \wedge\ \texttt{\{ \$NUM\{VAL\} == 1 \}} & \Longrightarrow x \\
TIMES(x, NUM) & \wedge\ \texttt{\{ \$NUM\{VAL\} == 1 \}} & \Longrightarrow x
\end{array}$$

1. Dado un programa como

   ```
   int a; a = a * 4 * 5;
   ```

   ¿Será plegado el `4 * 5`? Sin embargo si que se pliega si el programa es de la forma:

   ```
   int a; a = a * (4 * 5);
   ```

   No intente en esta práctica que programas como el primero o como `4*a*5*b` sean plegados. Para lograrlo sería necesario introducir transformaciones adicionales y esto no se requiere en esta práctica.

2. ¿Existe un orden óptimo en el que ejecutar las transformaciones?
3. Ponga un ejemplo en el que sea beneficioso ejecutar el plegado primero.
4. Ponga otro ejemplo en el que sea beneficioso ejecutar el plegado después.
5. ¿Es necesario aplicar las transformaciones reiteradamente?
6. ¿Cuál es la condición de parada?
7. Como es habitual la pregunta 6 tiene una respuesta TIMTOWTDI: una posibilidad la da el módulo Data::Compare el cual puede obtenerse desde CPAN y que permite comparar estructuras de datos, pero existe una solución mas sencilla. ¿Cuál?

2.13. Asignación de Direcciones

Esta suele ser considerada la primera de las fases de síntesis. Las anteriores lo eran de análisis. La fase de análisis es una transformación $texto\ fuente \rightarrow arbol$, mientras que la fase síntesis es una transformación inversa $arbol \rightarrow texto\ objeto$ que produce una “linealización” del árbol. En general, se ubican en la fase de síntesis todas las tareas que dependan del lenguaje objeto.

La asignación de direcciones depende de la máquina objetivo en cuanto conlleva consideraciones sobre la longitud de palabra, las unidades de memoria direccionables, la compactación de objetos pequeños (por ejemplo, valores lógicos), el alineamiento a fronteras de palabra, etc.

En nuestro caso debemos distinguir entre las cadenas y las variables enteras.

Las constantes literales (como `"hola"`) se almacenan concatenadas en orden de aparición textual. Una variable de tipo cadena ocupa dos palabras, una dando su dirección y otra dando su longitud.

```
sub factor() {
  my ($e, $id, $str, $num);

  if ($lookahead eq 'NUM') { ... }
  elsif ($lookahead eq 'ID') { ... }
  elsif ($lookahead eq 'STR') {
    $str = $value;
    my ($offset, $length) = Address::Assignment::str($str);
    match('STR');
    return STR->new(OFFSET => $offset, LENGTH => $length, TYPE => $string_type);
  }
  elsif ($lookahead eq '(') { ... }
  else { Error::fatal("Se esperaba (, NUM o ID"); }
}
```

El código de la subrutina `Address::Assignment::str` es:

```
  sub str {
    my $str = shift;
    my $len = length($str);
    my $offset = length($data);
    $data .= $str;
    return ($offset, $len);
  }
```

Una posible mejora es la que se describe en el punto 2 de la práctica 2.13.1.

Hemos supuesto que las variables enteras y las referencias ocupan una palabra. Cada vez que una variable es declarada, se le computa su dirección relativa:

```
# declaration -> type idlist
# type        -> INT | STRING
sub declaration() {
  my ($t, $decl, @il);

  if (($lookahead eq 'INT') or ($lookahead eq 'STRING')) {
    $t = &type(); @il = &idlist();
    &Semantic::Analysis::set_types($t, @il);
    &Address::Assignment::compute_address($t, @il);
    $decl = [$t, \@il];
    return bless $decl, 'DECL';
  }
  else { Error::fatal('Se esperaba un tipo'); }
}
```

Se usa una variable `$global_address` para llevar la cuenta de la última dirección utilizada y se introduce un atributo `ADDRESS` en la tabla de símbolos:

```
sub compute_address {
  my $type = shift;
  my @id_list = @_;

  for my $i (@id_list) {
    $symbol_table{$i}->{ADDRESS} = $global_address;
    $global_address += $type->LENGTH;
  }
}
```

Por último situamos todas las cadenas después de las variables del programa fuente:

```
...
##### En compile, despues de haber calculado las direcciones
Tree::Transform::match_and_transform_list(
  NODES => $tree->{STS},
  PATTERN => sub {
    $_[0]->isa('STR')
  },
  ACTION => sub { $_[0]->{OFFSET} += $global_address; }
);
```

Esta aproximación es bastante simplista en cuanto que usa una palabra por carácter. El punto 3 de la práctica 2.13.1 propone una mejora.

2.13.1. Práctica: Cálculo de las Direcciones

Modifique el cálculo de las direcciones para la gramática de Tutu extendida con sentencias compuestas que fué presentada en la sección 2.10.2.

1. Resuelva el problema de calcular las direcciones de las variables declaradas en los bloques. La memoria ocupada por dichas variables se libera al terminar el bloque. Por tanto la variable `$global_address` disminuye al final de cada bloque.
2. En el código de la subrutina `Address::Assignment::str`

```
sub str {
  my $str = shift;
  my $len = length($str);
  my $offset = length($data);
  $data .= $str;
  return ($offset, $len);
}
```

no se comprueba si la cadena `$str` ya está en `$data`. Es natural que si ya está no la insertemos de nuevo. Introduzca esa mejora. Para ello use la función `pos`, la cual devuelve el desplazamiento en `$data` donde quedó la última búsqueda `$data =~ m/regexp/g`. El siguiente ejemplo con el depurador le muestra como trabaja. Escriba `perldoc -f pos` para obtener mas información sobre la función `pos`. Vea un ejemplo de uso:

```
DB<1> $str = "world"
#              0123456789012345678901234567890l2
DB<2> $data = "hello worldjust another statement"
DB<3> $res = $data =~ m{$str}g
DB<4> x pos($data) # la siguiente busqueda comienza en la 11
0  11
DB<5> $s = "hello"
DB<6> $res = $data =~ m{$s}g
DB<7> x pos($data) # No hay "hello" despues de world en $data
0  undef
DB<8> $res = $data =~ m{$str}g # Repetimos ...
DB<9> x pos($data)
0  11
DB<10> pos($data) = 0  # Podemos reiniciar pos!
DB<11> $res = $data =~ m{$s}g
DB<12> x pos($data) # Ahora si se encuentra "hello"
0  5
```

3. Empaquete las cadenas asumiendo que cada carácter ocupa un octeto o byte. Una posible solución al problema de alineamiento que se produce es rellenar con ceros los bytes que faltan hasta completar la última palabra ocupada por la cadena. Por ejemplo, si la longitud de palabra de la máquina objeto es de 4 bytes, la palabra *procesador* se cargaría asi:

```
   0    |    1    |    2
0 1 2 3 | 4 5 6 7 | 8 9 10  11
p r o c | e s a d | o r \0  \0
```

Existen dos posibles formas de hacer esta empaquetado. Por ejemplo, si tenemos las cadenas `lucas` y `pedro` las podemos introducir en la cadena `$data` asi:

```
   0    |     1       |    2      |     3
0 1 2 3 | 4  5  6  7  | 8 9 10 11 | 12 13 14 15
l u c a | s \0 \0 \0  | p e  d  r |  o \0 \0 \0
```

o bien:

```
   0    |    1    |    2
0 1 2 3 | 4 5 6 7 | 8 9 10 11
l u c a | s p e d | r o \0 \0
```

La segunda forma aunque compacta más tiene la desventaja de hacer luego mas lento el código para el direccionamiento de las cadenas, ya que no empiezan en frontera de palabra. Utilice el primer modo.

Observe que este empaquetado introduce un efecto en lo que se considera un éxito en la búsqueda de una cadena `$str` en `$data`. ¿Que ocurre si `$str` aparece en `$data` como subcadena de una cadena previamente empaquetada ocupando una posición que no es frontera de palabra?

Cuando rellene con `\0` la cadena use el operador `x` (letra equis) para multiplicar número por cadena:

```
DB<1> $x = "hola"x4
DB<2> p $x
holaholaholahola
```

esto le evitará escribir un bucle.

4. Mantega atributos distintos en el nodo `STR` para el desplazamiento y longitud de `$str` en `$data` (en caracteres) y su dirección y tamaño final en memoria (en palabras).

2.14. Generación de Código: Máquina Pila

El generador de código emite el código para el lenguaje objeto, utilizando las direcciones calculadas en la fase anterior. Hemos simplificado esta fase considerando una máquina orientada a pila: una memoria de pila en la que se computan las expresiones, un segmento de datos en el que se guardan las variables y un segmento para guardar las cadenas. La estrategia utilizada es similar a la que vimos en el plegado de las constantes. Definimos un método para la traducción de cada clase de nodo del AAA. Entonces, para traducir el programa basta con llamar al correspondiente método:

```
$tree = Syntax::Analysis::parser;
... # otras fases
########code generation
local $target = ""; # target code
$tree->translate;
...
```

en la cadena `$target` dejamos el código emitido. Cada método visita los hijos, traduciéndolos y añadiendo el código que fuera necesario para la traducción del nodo.

```
sub PROGRAM::translate {
  my $tree = shift;

  $target .= "DATA ". $data."\n" if $data;
  $tree->STS->translate;
}
```

Traducir la lista de sentencias es concatenar la traducción de cada una de las sentencias en la lista:

```
sub STATEMENTS::translate {
  my $statements = shift;
  my @statements = @{$statements};

  for my $s (@statements) {
    $s->translate;
  }
}
```

Si suponemos que disponemos en el ensamblador de nuestra máquina objeto de instrucciones `PRINT_INT` y `PRINT_STR` que imprimen el contenido de la expresión que esta en la cima de la pila, la traducción será:

```
sub PRINT::translate {
  my $self = shift;

  $self->EXPRESSION->translate;
  if ($self->EXPRESSION->TYPE == $int_type) { emit "PRINT_INT\n"; }
  else {emit "PRINT_STR\n"; }
}
```

Asi, si la sentencia era `P c`, donde `c` es del tipo cadena, se debería eventualmente llamar al método de traducción del identificador:

```
sub ID::translate {
  my $self = shift;

  my $id = $self->VAL;
  my $type =  Semantic::Analysis::get_type($id);
  if ($type == $int_type) {
    emit "LOAD ".$symbol_table{$id}->{ADDRESS}."\n";
  }
  else {
    emit "LOAD_STRING ".$symbol_table{$id}->{ADDRESS}."\n";
  }
}
```

la función `emit` simplemente concatena el código producido a la salida:

```
sub emit { $target .= shift; }
```

Para la traducción de una sentencia de asignación supondremos de la existencia de isntrucciones `STORE_INT` y `STORE_STRING`. La instrucción `STORE_STRING` asume que en la cima de la pila están la dirección y la cadena a almacenar.

```
sub ASSIGN::translate {
  my $self = shift;

  $self->RIGHT->translate;
  my $id = $self->LEFT;
  $id->translate;
  my $type =  Semantic::Analysis::get_type($id->VAL);

  if ($type == $int_type) {
    emit "STORE_INT\n";
  }
  else {
    emit "STORE_STRING\n";
  }
}
```

Si se está traduciendo una sentencia como `a = "hola"`, se acabará llamando al método de traducción asociado con la clase `STR` el cual actúa empujando la dirección y la longitud de la cadena en la pila:

```
sub STR::translate {
  my $self = shift;

  emit "PUSHSTR ".$self->OFFSET." ".$self->LENGTH."\n";
}
```

Así la traducción de este fuente:

```
$ cat test06.tutu
string a;
a = "hola";
p a
```

es:

```
$ ./main.pl test06.tutu test06.ok
$ cat test06.ok
DATA 2, hola
PUSHSTR 2 4
PUSHADDR 0
STORE_STRING
LOAD_STRING 0
PRINT_STR
```

El resto de los métodos de traducción es similar:

```
sub LEFTVALUE::translate {
  my $id = shift ->VAL;

  emit "PUSHADDR ".$symbol_table{$id}->{ADDRESS}."\n";
}

sub NUM::translate {
  my $self = shift;

  emit "PUSH ".$self->VAL."\n";
}

sub PLUS::translate {
  my $self = shift;

  $self->LEFT->translate;
  $self->RIGHT->translate;
  emit "PLUS\n";
}

sub TIMES::translate {
  my $self = shift;

  $self->LEFT->translate;
  $self->RIGHT->translate;
  emit "MULT\n";
}
```

Veamos un ejemplo. Dado el código fuente:

```
$ cat test02.tutu
int a,b; string c;
a = 2+3; b = 3*4; c = "hola"; p c;
c = "mundo"; p c; p 9+2; p a+1; p b+1
```

el código generado es:

```
$ ./main.pl test02.tutu test02.ok
$ cat test02.ok
DATA 4, holamundo
PUSH 5
PUSHADDR 0
STORE_INT
PUSH 12
PUSHADDR 1
STORE_INT
PUSHSTR 4 4
PUSHADDR 2
STORE_STRING
LOAD_STRING 2
PRINT_STR
PUSHSTR 8 5
PUSHADDR 2
STORE_STRING
LOAD_STRING 2
PRINT_STR
PUSH 11
PRINT_INT
LOAD 0
INC
PRINT_INT
LOAD 1
INC
PRINT_INT
```

2.15. Generación de Código: Máquina Basada en Registros

La máquina orientada a pila para la que generamos código en la sección 2.14 es un ejemplo de la clase de máquinas que es usada por la mayoría de los lenguajes interpretados: Perl, Python; java, etc.

En esta sección introduciremos una máquina basada en registros. Suponemos que la máquina tiene k registros $R_0 \ldots R_{k-1}$. Las instrucciones toman dos argumentos, dejando el resultado en el primer argumento. Son las siguientes:

`LOADM Ri, [a]`	$R_i = M_a$
`LOADC Ri, c`	$R_i = c$
`STORE [a], Ri`	$M_a = R_i$
`ADDR Ri, Rj`	$R_i += R_j$
`ADDM Ri, [a]`	$R_i += M_a$
`ADDC Ri, c`	$R_i += c$
...	...

El problema es generar el código con el menor número de instrucciones posible, teniendo en cuenta la limitación existente de registros.

Supongamos que queremos traducir un subárbol $OP(t_1, t_2)$ y que la traducción del subárbol t_1 requiere r_1 registros y que la traducción de t_2 requiere r_2 registros, con $r_1 < r_2 \leq k$. Si realizamos primero la evaluación de t_1, debemos dejar el resultado en un registro que no podrá ser utilizado en la evaluación de t_2. Si $r_2 = k$, la evaluación de t_2 podría dar lugar a la necesidad de recurrir a almacenamiento temporal. Esta situación no se da si evaluamos primero t_2. En tal caso, dado que hay un registro en el que se guarda el resultado de t_2, quedan libres al menos $r_2 - 1$ registros. Como $r_2 - 1 \geq r_1$ se sigue que tenemos suficientes registros para traducir t_1. Como regla general es mejor evaluar primero el subárbol que mayores requerimientos de registros tiene.

La siguiente cuestión es como calcular los requerimientos en registros de una expresión dada. No consideraremos en esta fase límites en el número de registros disponibles. Obsérvese que si los requerimientos para los subárboles son distintos, $r_1 \neq r_2$ la traducción puede realizarse usando el máximo de ambos $\text{máx}\{r_1, r_2\}$ siguiendo la estrategia de traducir primero el que mayores requerimentos tenga. Si son iguales entonces se necesitan $r_1 + 1$ registros ya que es necesario un registro para guardar el resultado de la primera traducción.

Nótese que, como el juego de instrucciones para un operando puede tener como segundo argumento una dirección de memoria, los "segundos operandos" no necesitan registro. Por ejemplo, el árbol $PLUS(a, b)$ se traduce por

```
LOADM R0, a
PLUSM R0, b
```

Asi b no requiere registro, mientras que a si lo requiere. Por tanto, las hojas izquierdas requieren de registro mientras que las hojas derechas no.

Si t es un nodo de la forma $OP(t_1, t_2)$ el número de registros r_t requeridos por t viene dado por la fórmula:

$$r_t = \begin{cases} \text{máx}\{r_1, r_2\} & \text{si } r_1 \neq r_2 \\ r_1 + 1 & \text{si } r_1 = r_2 \end{cases}$$

Dotaremos a cada nodo del AST de un método `required_registers` que computa la demanda en registros de dicho nodo. Lo que haremos es introducir en la clase `Operation` de la cual heredan las operaciones binarias el correspondiente método `required_registers`:

```
package Operation;
our @ISA = ("Binary");

sub required_registers {
  my $self = shift;

  my $rl = $self->LEFT->required_registers('LEFT');
  my $rr = $self->RIGHT->required_registers('RIGHT');
  $self->{REQ_REG} = ($rl == $rr)? $rl+1: Aux::max($rl, $rr);
  return $self->REQ_REG;
}
```

El segundo argumento que recibe `required_registers` es su posición (izquierda o derecha) entre los hijos de su padre. dicha información no es usada en los nodos binarios. Su necesidad queda clara cuando se considera el cómputo del número de registros requeridos por las hojas.

El cómputo en las hojas corre a cargo del correspondiente método en la clase `Value`. Los nodos de tipo número (clase `NUM`), cadena (clase `STR`) y variable (clase `ID`) heredan de la clase `Value`.

```
package Value;
our @ISA = ("Leaf");

sub required_registers {
  my $self = shift;
```

```
  my $position = shift;

  $self->{REQ_REG} = ($position eq 'LEFT') ? 1 : 0;
  return $self->REQ_REG;
}
```

El atributo `REQ_REG` se computa para cada una de las sentencias del programa:

```
package STATEMENTS;

sub required_registers {
  my $self = shift;
  my @sts = @{$self};

  for (@sts) {
    $_->required_registers;
  }
}
```

Por supuesto los nodos `ASSIGN` y `PRINT` poseen sus propios métodos `required_registers`.

Una vez computados los requerimientos en registros de cada nódo, la generación de código para un nodo gestiona la asignación de registros usando una cola en la que se guardan los registros disponibles. Se siguen básicamente dos reglas para la traducción de un nodo `Operation`:

1. Realizar primero la traducción del hijo con mayores requerimientos y luego el otro
2. El resultado queda siempre en el registro que ocupa la primera posición en la cola

Hay cuatro casos a considerar: el primero es que el operando derecho sea una hoja. La generación de código para este caso es:

```
package Operation;
our @ISA = ("Binary");
...

sub gen_code {
  my $self = shift;

  if ($self->RIGHT->isa('Leaf')) {
    my $right = $self->RIGHT;
    my $a = $right->VAL;
    my $rightoperand = $right->gen_operand; # valor o dirección
    my $key = $right->key;                  # M, C, etc.
    $self->LEFT->gen_code;
    Aux::emit($self->nemonic."$key $RSTACK[0], $rightoperand # $a\n");
  }
  ...
}
```

La generación del nemónico se basa en tres métodos:

- El método `nemonic` devuelve el nemónico asociado con el nodo. Por ejemplo, para la clase `TIMES` el código es:

  ```
  sub  nemonic {
    return "MULT";
  }
  ```

- El método `key` devuelve el sufijo que hay que añadir para completar el nemónico, en términos de como sea el operando: `C` para los números, `M` para los identificadores, etc.
- El método `gen_operand` genera el operando. Así para las clases número e identificador su código es:

```
package NUM;
...
sub gen_operand {
  my $self = shift;

  return $self->VAL;
}
```

```
package ID;
...
sub gen_operand {
  my $self = shift;

  return $symbol_table{$self->VAL}->{ADDRESS},
}
```

El resto del código distingue tres casos, según sean r_1, r_2 y el número de registros disponibles. Los dos primeros casos desglosan la posibilidad de que uno de los dos subárboles pueda realizarse con el número de registros disponible ($\min\{r_1, r_2\} < k$). El tercer caso corresponde a que se necesiten temporales: $\min\{r_1, r_2\} \geq k$.

```
 1  ...
 2  if ($self->RIGHT->isa('Leaf')) { ...  }
 3  else { # Hijo derecho no es una hoja
 4    my ($t1, $t2) = ($self->LEFT, $self->RIGHT);
 5    my ($r1, $r2) = ($t1->REQ_REG, $t2->REQ_REG);
 6
 7    if ($r1 < Aux::min($r2, $NUM_REG)) {
 8      $t2->gen_code;
 9      my $R = shift @RSTACK;
10      $t1->gen_code;
11      Aux::emit($self->nemonic."R $RSTACK[0], $R\n");
12      push @RSTACK, $R;
13    }
14    ...
15  }
```

En este caso debemos realizar primero la traducción del hijo derecho. Salvando su resultado en `$R`. El registro es retirado de la cola y traducimos el lado izquierdo. El resultado ha quedado en el primer registro de la cola. Emitimos la operación, añadiendo el sufijo `R`, ya que se trata de una operación entre registros y posteriormente devolvemos el registro a la cola.

Ejercicio 2.15.1. *Responda a las siguientes preguntas:*

1. *Si en el código anterior sustituimos la línea 12*

   `push @RSTACK, $R`

   por

   `unshift @RSTACK, $R`

   ¿Seguiría funcionando el código?

2. *¿Podemos asegurar en este subcaso que el código generado para* `$t2` *(línea 8) se ha realizado íntegramente en los registros?*

Los otros dos casos tienen similar tratamiento:

```
  if ($self->RIGHT->isa('Leaf')) { ...  }
  else { ...
    if ($r1 < Aux::min($r2, $NUM_REG)) { ... }
    elsif (($r1 >= $r2) and ($r2 < $NUM_REG)) {
      $t1->gen_code;
      my $R = shift @RSTACK;
      $t2->gen_code;
      Aux::emit($self->nemonic."R $R, $RSTACK[0]\n");
      unshift @RSTACK, $R;
    }
    elsif (($r1 >= $NUM_REG) and ($r2 >= $NUM_REG)) {
      $t2->gen_code;
      Aux::emit("STORE $T, $RSTACK[0]\n");
      $T++;
      $t1->gen_code;
      $T--;
      Aux::emit($self->nemonic."M $RSTACK[0], $T\n");
    }
  }
}
```

Antes de comenzar a generar el código, la variable `$T` debe ser inicializada a un valor apropiado, de manera que se usen direcciones no ocupadas por los datos. Por ejemplo:

```
local $T =  $final_global_address+length($data);
```

El método `gen_code` sólo debería ser llamado sobre una hoja si se trata de una hoja izquierda (en cuyo caso el número de registros requeridos es uno):

```
package Value;
our @ISA = ("Leaf");
...

sub gen_code {
  my $self = shift;
  my $a = $self->VAL;

  if ($self->REQ_REG == 1) {
    if (ref($self) eq "NUM") { Aux::emit("LOADC $RSTACK[0], $a\n"); }
    else {
      my $address = $symbol_table{$a}->{ADDRESS};
      Aux::emit("LOADM $RSTACK[0], $address # $a\n");
    }
  }
  else {
    croak("gen_code visita hoja izquierda con REQ_REG = ".$self->REQ_REG);
  }
}
```

La pila de registros es inicializada al número de registros disponibles:

```
use constant LAST_REG => 1;
our @RSTACK = map "R$_", 0..LAST_REG; # Registros disponibles
```

Ejercicio 2.15.2. *Responda a las siguientes preguntas:*

- *¿Cuáles son los requerimientos de registros para un nodo de la clase ASSIGN?*
- *¿Cuáles son los requerimientos de registros para un nodo de la clase PRINT?*
- *¿Se puede lograr la funcionalidad proveída por el método* `required_registers` *usando* `match_and_transform`*?*

  ¿Sería necesario introducir modificaciones en `match_and_transform`*? Si es así, ¿Cuáles?.*

2.15.1. Práctica: Generación de Código

1. Complete la generación de código para la máquina basada en registros. Recuerde que debe escribir el método `required_registers` para las diferentes clases `Value`, `Operation`, `ASSIGN`, `PRINT`, `STATEMENTS`, etc. Asi mismo deberá escribir el método `gen_code` para las diversas clases: `Value`, `Operation`, `ASSIGN`, `PRINT`, `STATEMENTS`, etc. Recuerde que los temporales usados durante la generación de código deben ubicarse en una zona que no esté en uso.

2. En la sección anterior no se consideraba la generación de código para las cadenas. Amplíe y/o modifique el juego de instrucciones como considere conveniente. El siguiente ejemplo de traducción sugiere como puede ser la generación de código para las cadenas:

Fuente	Objeto

`string a,b;`
`a = "hola";`
`b = a;`
`p b`
	`1 LSTRG  R0, 4, 4`
	`2 STORES  0, R0 # a`
	`3 LOADS  R0, 0 # a`
	`4 STORES  2, R0 # b`
	`5 LOADS  R0, 2 # b`
	`6 PRNTS  R0`

Asuma que los registros pueden contener dos direcciones de memoria (línea 1). La instrucción `LSTRG R0, a, b` carga las constantes (direcciones) `a` y `b` en el registro. La constante `"hola"` ocupa en la posición final en la que se colocan los contenidos de `$data` un desplazamiento de 4 y ocupa 4 palabras. Las instrucción `LOADS R, a` carga las dos palabras en las direcciones `a` y `a+1` en el registro `R`. La instrucción `STORES a, R` se encarga de que las dos palabras en la dirección `a` queden referenciando una cadena igual a la apuntada por el registro `R`. La instrucción `PRNTS` imprime la cadena apuntada por el registro. En una situación mas realista instrucciones como `STORES a, R` y `PRNTS` probablemente serían llamadas a funciones/servicios del sistema o de la librería para soporte en tiempo de ejecución asociada al lenguaje.

3. Se puede mejorar el código generado si hacemos uso de las propiedades algebraicas de los operadores. Por ejemplo, cuando se tiene un operador conmutativo que ha sido asociado a derechas, como ocurre en este programa fuente:

```
$ cat test18.tutu
int a,b,c;

a = a + (b + c)
```

El código producido por el compilador es:

```
LOADM R0, 0 # a
LOADM R1, 1 # b
PLUSM R1, 2 # c
PLUSR R0, R1
STORE  0, R0 # a
```

En este caso, la expresión `a + (b + c)` corresponde a un árbol que casa con el patrón árbol

$$PLUS(ID, t) \text{ and } \{r_t \geq 1\}$$

Donde r_t es el número de registros requeridos por t. En tales casos es posible sacar ventaja de la conmutatividad de la suma y transformar el árbol

$$PLUS(ID, t) \text{ and } \{r_t \geq 1\} \Longrightarrow PLUS(t, ID)$$

Observe que mientras el primer árbol requiere $\max\{2, r_t\}$ registros, el segundo requiere r_t registros, que en general es menor. Esta transformación invierte la traducción:

```
traduce(t)
ADDM $RSTACK[0], dirección de ID
```

que daría lugar a:

```
LOADM R0, 1 # b
PLUSM R0, 2 # c
PLUSM R0, 0 # a
STORE  0, R0 # a
```

la cual usa una instrucción y un registro menos.

Usando `match_and_transform` modifique el generador de código para que, después de la fase de cálculo del número de registros requeridos, aplique esta transformación sobre los nodos conmutativos cuyo hijo izquierdo sea un identificador y su hijo derecho requiera al menos un registro.

2.16. Optimización de Código

Aunque en esta fase se incluyen toda clase de optimizaciones, es aquí donde se hacen las optimizaciones de código dependientes del sistema objeto. Normalmente se recorre el código generado buscando secuencias de instrucciones que se puedan sustituir por otras cuya ejecución sea mas eficiente. El nombre *Peephole optimization* hace alusión a esta especie de “ventana de visión” que se desplaza sobre el código. En nuestro caso, supongamos que disponemos de una instrucción `INC` que permite incrementar eficientemente una expresión. Recorremos el código buscando por un patrón ”sumar 1z lo reeemplazamos adecuadamente.

```
package Peephole::Optimization;

sub transform {
  $target = shift;
  $target =~ s/PUSH 1\nPLUS/INC/g;
}
```

Otro ejemplo de optimización peephole consiste en reemplazar las operaciones flotantes de división por una constante por la multiplicación por la inversa de la misma (aquí se pueden introducir diferencias en el resultado, debido a la inexactitud de las operaciones en punto flotante y a que, si se trata de un *compilador cruzado* la aritmética flotante en la máquina en la que se ejecuta el compilador puede ser diferente de la de la máquina que ejecutará el código objeto).

2.16.1. Práctica: Optimización Peephole

1. Optimice el código generado para la máquina de registros sustituyendo las operaciones de multiplicación y división enteras por una constante que sea potencia de dos (de la forma 2^n) por operaciones de desplazamiento. Repase el capítulo de expresiones regulares. Es posible que aquí quiera emplear una sustitución en la que la cadena de reemplazo sea evaluada sobre la marcha. Si es así, repase la sección 1.1.6. La siguiente sesión con el depurador pretende ilustrar la idea:

   ```
   $ perl -de 0
   DB<1> $a = "MUL R2, 16"
   DB<2> $a =~ s/MUL R(\d), (\d+)/($2 == 16)?"SHL R$1, 4":$&/e
   DB<3> p $a
   SHL R2, 4
   DB<5> $a = "MUL R2, 7"
   DB<6> $a =~ s/MUL R(\d), (\d+)/($2 == 16)?"SHL R$1, 4":$&/e
   DB<7> p $a
   MUL R2, 7
   DB<8>
   ```

2. El plegado de constantes realizado durante la optimización de código independiente de la máquina (véase la sección 2.11) es parcial. Si los árboles para el producto se hunden a izquierdas, una expresión como `a = a * 2 * 3` no será plegada, ya que produce un árbol de la forma

   $$t = TIMES(TIMES(a, 2), 3)$$

   Dado que el algoritmo no puede plegar $t/1$ tampoco plegará t. Busque en el código objeto secuencias de multiplicaciones por constantes y abrévielas en una. Haga lo mismo para las restantes operaciones.

3. Dado el siguiente programa:

   ```
   $ cat test14.tutu
   int a,b; a = 2; b = a*a+1
   ```

   El código producido por el traductor para la máquina de registros es:

   ```
   1 LOADC R0, 2
   2 STORE  0, R0 # a
   3 LOADM R0, 0 # a
   4 MULTM R0, 0 # a
   5 PLUSC R0, 1 # 1
   6 STORE  1, R0 # b
   ```

   Se ve que la instrucción de carga `LOADM R0, 0` de la línea 3 es innecesaria por cuanto el contenido de la variable `a` ya está en el registro `R0`, ya que fué cargada en el registro en la línea 2. Nótese que esta hipótesis no es necesariamente cierta si la línea 3 fuera el objetivo de un salto desde otro punto del programa. Esta condición se cumple cuando nos movemos dentro de un *bloque básico*: una secuencia de instrucciones que no contiene instrucciones de salto ni es el objetivo de instrucciones de salto, con la excepción de las instrucciones inicial y final. Mejore el código generado intentando detectar patrones de este tipo, eliminando la operación de carga correspondiente.

Capítulo 3

Análisis Sintáctico con `Parse::Eyapp`

3.1. Conceptos Básicos para el Análisis Sintáctico

Suponemos que el lector de esta sección ha realizado con éxito un curso en teoría de autómatas y lenguajes formales. Las siguientes definiciones repasan los conceptos mas importantes.

Definición 3.1.1. *Dado un conjunto* A*, se define* A^* *el cierre de Kleene de* A *como:* $A^* = \cup_{n=0}^{\infty} A^n$

Se admite que $A^0 = \{\epsilon\}$*, donde* ϵ *denota la palabra vacía, esto es la palabra que tiene longitud cero, formada por cero símbolos del conjunto base* A*.*

Definición 3.1.2. *Una gramática* G *es una cuaterna* $G = (\Sigma, V, P, S)$*.* Σ *es el conjunto de terminales.* V *es un conjunto (disjunto de* Σ*) que se denomina conjunto de* variables sintácticas *o* categorías gramáticales*,* P *es un conjunto de pares de* $V \times (V \cup \Sigma)^*$*. En vez de escribir un par usando la notación* $(A, \alpha) \in P$ *se escribe* $A \to \alpha$*. Un elemento de* P *se denomina producción. Por último,* S *es un símbolo del conjunto* V *que se denomina símbolo de arranque.*

Definición 3.1.3. *Dada una gramática* $G = (\Sigma, V, P, S)$ *y* $\mu = \alpha A \beta \in (V \cup \Sigma)^*$ *una frase formada por variables y terminales y* $A \to \gamma$ *una producción de* P*, decimos que* μ *deriva en un paso en* $\alpha\gamma\beta$*. Esto es, derivar una cadena* $\alpha A \beta$ *es sustituir una variable sintáctica* A *de* V *por la parte derecha* γ *de una de sus reglas de producción. Se dice que* μ *deriva en* n *pasos en* δ *si deriva en* $n-1$ *pasos en una cadena* η *la cual deriva en un paso en* δ*. Se escribe entonces que* $\mu \overset{n}{\Longrightarrow} \delta$*. Una cadena deriva en 0 pasos en si misma. Se escribe entonces que* $\mu \overset{*}{\Longrightarrow} \delta$*.*

Definición 3.1.4. *Dada una gramática* $G = (\Sigma, V, P, S)$ *se denota por* $L(G)$ *o lenguaje generado por* G *al lenguaje:*

$$L(G) = \{x \in \Sigma^* : S \overset{*}{\Longrightarrow} x\}$$

Esto es, el lenguaje generado por la gramática G *esta formado por las cadenas de terminales que pueden ser derivados desde el símbolo de arranque.*

Definición 3.1.5. *Una derivación que comienza en el símbolo de arranque y termina en una secuencia formada por sólo terminales de* Σ *se dice* completa.

Una derivación $\mu \overset{*}{\Longrightarrow} \delta$ *en la cual en cada paso* $\alpha A x$ *la regla de producción aplicada* $A \to \gamma$ *se aplica en la variable sintáctica mas a la derecha se dice* una derivación a derechas

Una derivación $\mu \overset{*}{\Longrightarrow} \delta$ *en la cual en cada paso* $x A \alpha$ *la regla de producción aplicada* $A \to \gamma$ *se aplica en la variable sintáctica mas a la izquierda se dice* una derivación a izquierdas

Definición 3.1.6. *Observe que una derivación puede ser representada como un árbol cuyos nodos están etiquetados en* $V \cup \Sigma$*. La aplicación de la regla de producción* $A \to \gamma$ *se traduce en asignar como hijos del nodo etiquetado con* A *a los nodos etiquetados con los símbolos* $X_1 \ldots X_n$ *que constituyen la frase* $\gamma = X_1 \ldots X_n$*. Este árbol se llama* árbol sintáctico concreto *asociado con la derivación.*

Definición 3.1.7. *Observe que, dada una frase $x \in L(G)$ una derivación desde el símbolo de arranque da lugar a un árbol. Ese árbol tiene como raíz el símbolo de arranque y como hojas los terminales $x_1 \ldots x_n$ que forman x. Dicho árbol se denomina* árbol de análisis sintáctico concreto *de x. Una derivación determina una forma de recorrido del árbol de análisis sintáctico concreto.*

Definición 3.1.8. *Una gramática G se dice ambigua si existe alguna frase $x \in L(G)$ con al menos dos árboles sintácticos. Es claro que esta definición es equivalente a afirmar que existe alguna frase $x \in L(G)$ para la cual existen dos derivaciones a izquierda (derecha) distintas.*

Definición 3.1.9. *Un* esquema de traducción *es una gramática independiente del contexto en la cual se han insertado fragmentos de código en las partes derechas de sus reglas de producción.*

$$A \rightarrow \alpha\{action\}$$

Los fragmentos de código asi insertados se denominan acciones semánticas*.*

En un esquema de traducción los nodos del árbol sintáctico tienen asociados atributos*. Si pensamos que cada nodo del árbol es un objeto, entonces los atributos del nodo son los atributos del objeto. Las reglas semánticas determinan la forma en la que son evaluados los atributos.*

Los fragmentos de código de un esquema de traducción calculan y modifican los atributos asociados con los nodos del árbol sintáctico. El orden en que se evalúan los fragmentos es el de un recorrido primero-profundo del árbol de análisis sintáctico. Esto significa que si en la regla $A \rightarrow \alpha\beta$ insertamos un fragmento de código:

$$A \rightarrow \alpha\{action\}\beta$$

La acción $\{action\}$ se ejecutará después de todas las acciones asociadas con el recorrido del subárbol de α y antes que todas las acciones asociadas con el recorrido del subárbol β.

Obsérvese que para poder aplicar un esquema de traducción hay que - al menos conceptualmente - construir el árbol sintáctico y después aplicar las acciones empotradas en las reglas en el orden de recorrido primero-profundo. Por supuesto, si la gramática es ambigua una frase podría tener dos árboles y la ejecución de las acciones para ellos podría dar lugar a diferentes resultados. *Si se quiere evitar la multiplicidad de resultados (interpretaciones semánticas) es necesario precisar de que árbol sintáctico concreto se esta hablando.*

Definición 3.1.10. *Un atributo tal que su valor en un nodo puede ser computado en términos de los atributos de los hijos del nodo se dice que es un* atributo sintetizado*.*

El siguiente esquema de traducción recibe como entrada una expresión en infijo y produce como salida su traducción a postfijo para expresiones aritmeticas con sólo restas de números:

$expr \rightarrow expr_1 - NUM$ `{ $expr{TRA} = $expr[1]{TRA}." ".$NUM{VAL}." - "}`
$expr \rightarrow NUM$ `{ $expr{TRA} = $NUM{VAL} }`

Que para una entrada `2 - 3 - 7` daría lugar a la siguiente evaluación:

```
e ’2 3 - 7 -’
‘-- --- e ’2 3 -’
    |   ‘------ e ’2’
    |       |    ‘-- N 2
    |       |-- ’-’
    |       ‘-- N 3
    |-- ’-’
    ‘-- N 7
```

Definición 3.1.11. *Un* atributo heredado *es aquel cuyo valor se computa a partir de los valores de sus hermanos y de su padre.*

Ejemplo 3.1.1. *Un ejemplo de atributo heredado es el tipo de las variables en las declaraciones:*

$decl \to type$ `{ $list{T} = $type{T} }` $list$
$type \to INT$ `{ $type{T} = $int }`
$type \to STRING$ `{ $type{T} = $string }`
$list \to ID\ ,$ `{ $ID{T} = $list{T}; $list_1{T} = $list{T} }` $list_1$
$list \to ID$ `{ $ID{T} = $list{T} }`

3.2. `Parse::Eyapp`: Un Generador de Analizadores Sintácticos

El generador de analizadores sintácticos `Parse::Eyapp` es un analizador LALR inspirado en `yacc`. Parse::Eyapp es una extensión de Parse::Yapp escrita por Casiano Rodriguez-Leon. Puede descargarlo desde CPAN e instalarlo en su máquina siguiendo el procedimiento habitual.

El generador de analizadores sintácticos `Parse::Eyapp` que estudiaremos en las siguientes secciones funciona de manera similar a un esquema de traducción. Las reglas de producción de la gramática son aumentadas con reglas semánticas. Los símbolos que aparecen en la regla de producción tienen *atributos* asociados y las reglas semánticas dicen como deben ser computados dichos atributos. Consideremos, por ejemplo, el siguiente fragmento de programa `eyapp`:

```
exp:  exp '+' exp         { $_[1] + $_[3] }
```

que dice que asociado con el símbolo *exp* de la regla de producción $exp \to exp' +' exp$ tenemos el atributo valor numérico y que para computar el atributo valor de la variable sintáctica *exp* en la parte izquierda tenemos que sumar los atributos asociados con los símbolos primero y tercero de la parte derecha.

Por defecto `Parse::Eyapp` no provee un esquema de traducción ya que - aunque el orden de ejecucíon de las acciones es de abajo-arriba y de izquierda a derecha como en un esquema - no es posible acceder a atributos de nodos que aún no han sido visitados.

Para ilustrar el uso de `Parse::Eyapp` veamos un ejemplo en el que se implanta una gramática cuyas frases son secuencias (separadas por retornos de carro) de expresiones aritméticas.

Los contenidos del programa `eyapp` los hemos guardado en un fichero denominado `CalcSyntax.eyp`

Partes de un Programa `Eyapp`

Un programa `eyapp` consta de tres partes:

- la cabeza,
- el cuerpo
- y la cola.

Cada una de las partes va separada de las otras por el símbolo `%%` en una línea aparte.
Así, el `%%` de la línea 8 separa la cabeza del cuerpo y el de la línea 31 el cuerpo de la cola.

- En la cabecera se colocan el código de inicialización, las declaraciones de terminales, las reglas de precedencia, etc.
- El cuerpo contiene las reglas de la gramática y las acciones asociadas.
- Por último, la cola de un program `eyapp` contiene las rutinas de soporte al código que aparece en las acciones asi como, posiblemente, rutinas para el análisis léxico y el tratamiento de errores.

```
pl@nereida:~/LEyapp/examples$ cat -n CalcSyntax.eyp
 1  # CalcSyntax.eyp
 2  %right  ’=’
 3  %left   ’-’ ’+’
 4  %left   ’*’ ’/’
 5  %left   NEG
 6  %right  ’^’
 7
 8  %%
 9
10  input:  line *  { print "input -> line *\n" }
11  ;
12
13  line:
14    ’\n’         { print "line -> \\n\n" }
15    | exp ’\n’   { print "line -> exp \\n\n"}
16  ;
17
18  exp:
19      NUM               { print "exp -> NUM ($_[1])\n"; }
20    | VAR               { print "exp -> VAR ($_[1])\n"; }
21    | VAR ’=’ exp       { print "exp -> VAR ’=’ exp\n"; }
22    | exp ’+’ exp       { print "exp -> exp ’+’ exp\n"; }
..    . ..............................................
29  ;
30
31  %%
32
33  sub _Error {
..    ...................................
41  }
42
..  ......................................
44
45  sub _Lexer {
46      my($parser)=shift;
..      .................
58  }
59
60  sub Run {
61      my($self)=shift;
62
63      $input = shift;
64      return $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error );
65  }
```

Las Reglas

Todas las partes derechas de las reglas de producción de una misma variable sintáctica se escriben juntas separadas mediante la barra vertical `|`.

```
10  input:  line *  { print "input -> line *\n" }
11  ;
12
13  line:
```

```
14    ’\n’          { print "line -> \\n\n" }
15    | exp ’\n’    { print "line -> exp \\n\n"}
16  ;
```

En este ejemplo hemos simplificado las acciones semánticas reduciéndolas a mostrar la regla de producción encontrada.

Reglas de Producción Vacías

Un asterisco (como en la línea 10) indica repetición cero o mas veces de la expresión a la que se aplica. De hecho la línea 10 es casi equivalente a:

```
input:  temp
;
temp:
    /* vacio */
  | temp line
;
```

Observe como en el código anterior hemos codificado la regla de producción *temp* $\rightarrow \epsilon$ como:

```
temp:
    /* vacio */
```

Es buena costumbre de programación cuando se tiene una regla de producción que produce vacío ponerla la primera del grupo y añadir un comentario como este. Dado que vacío se representa en `Eyapp` mediante la cadena vacía es fácil que pase desapercibida. Es por ello que se recomienda que *una regla vacía sea siempre la primera y que este comentada* como en el ejemplo.

Tratamiento de las Ambiguedades

Hay numerosas ambiguedades en esta gramática. Observe las reglas para los diferentes tipos de expresiones:

```
18 exp:
19      NUM                { print "exp -> NUM ($_[1])\n"; }
20    | VAR                { print "exp -> VAR ($_[1])\n"; }
21    | VAR ’=’ exp        { print "exp -> VAR ’=’ exp\n"; }
22    | exp ’+’ exp        { print "exp -> exp ’+’ exp\n"; }
23    | exp ’-’ exp        { print "exp -> exp ’-’ exp\n"; }
24    | exp ’*’ exp        { print "exp -> exp ’*’ exp\n"; }
25    | exp ’/’ exp        { print "exp -> exp ’/’ exp\n"; }
26    | ’-’ exp %prec NEG  { print "exp -> ’-’ exp\n"; }
27    | exp ’^’ exp        { print "exp -> exp ’^’ exp\n"; }
28    | ’(’ exp ’)’        { print "exp -> ’(’ exp ’)’\n"; }
29 ;
```

Surgen preguntas como:

- ¿Como debo interpretar la expresión `4 - 5 - 2`? ¿Como `(4 - 5) - 2`? ¿o bien `4 - (5 - 2)`? La respuesta la da la asignación de asociatividad a los operadores que hicimos en la cabecera:

  ```
  1 # CalcSyntax.eyp
  2 %right  ’=’
  3 %left   ’-’ ’+’
  4 %left   ’*’ ’/’
  5 %left   NEG
  6 %right  ’^’
  7
  8 %%
  ```

La Asociatividad de Terminales y la Ambiguedad

Las declaraciones `%left` y `%right` expresan la asociatividad y precedencia de los terminales, permitiendo decidir que árbol construir en caso de ambiguedad.

Los terminales declarados en líneas posteriores tienen mas prioridad que los declarados en las líneas anteriores.

Por defecto, *una regla de producción tiene la prioridad del último terminal que aparece en su parte derecha.*

Al declarar como asociativo a izquierdas al terminal `-` hemos resuelto la ambiguedad en `4 -5 -2`. Lo que estamos haciendo es indicarle al analizador que a la hora de elegir entre los árboles abstractos $-(-(4,5),2)$ y $-(4,-(5,2))$ elija siempre el árbol que se hunde a izquierdas.

- ¿Como debo interpretar la expresión `4 - 5 * 2`? ¿Como `(4 - 5) * 2`? ¿o bien `4 - (5 * 2)`? Al declarar que `*` tiene mayor prioridad que `-` estamos resolviendo esta otra fuente de ambiguedad. Esto es así pues `*` fué declarado en la línea 11 y `-` en la 10. Por tanto el árbol será `-(4, *(5,2))`.

La declaración de `^` como asociativo a derechas y con un nivel de prioridad alto resuelve las ambiguedades relacionadas con este operador:

```
18  exp:
..    ...............................................
28    | ’(’ exp ’)’        { print "exp -> ’(’ exp ’)’\n"; }
```

Ejercicio 3.2.1. *¿Como se esta interpretando la expresión* `-2^2`*? ¿Cómo* `(-2)^2`*? ¿o bien* `-(2^2)`*?*

Modificación de la Prioridad Implícita de una Regla

Una regla de producción puede ir seguida de una directiva `%prec` la cual le da una prioridad explícita. Esto puede ser de gran ayuda en ciertos casos de ambiguedad.

```
26    | ’-’ exp %prec NEG  { print "exp -> ’-’ exp\n"; }
```

¿Cual es la ambiguedad que surge con esta regla? La ambiguedad de esta regla esta relacionada con el doble significado del menos como operador unario y binario: hay frases como `-y-z` que tiene dos posibles interpretaciones: Podemos verla como `(-y)-z` o bien como `-(y-z)`. Hay dos árboles posibles. El analizador, cuando este analizando la entrada `-y-z` y vea el segundo `-` (después de haber leído `-y`) deberá escoger uno de los dos árboles. ¿Cuál?. El conflicto puede verse como una “lucha” entre la regla `exp: ’-’ exp` la cual interpreta la frase como `(-y)-z` y la segunda aparición del terminal `-` el cuál “quiere entrar” para que gane la regla `exp: exp ’-’ exp` y dar lugar a la interpretación `-(y-z)`. En este caso, las dos reglas $E \to -E$ y $E \to E - E$ tienen, en principio la prioridad del terminal `-`, el cual fué declarado en la zona de cabecera:

```
1 # CalcSyntax.eyp
2 %right  ’=’
3 %left   ’-’ ’+’
4 %left   ’*’ ’/’
5 %left   NEG
6 %right  ’^’
7
8 %%
```

La prioridad expresada explícitamente para la regla por la declaración `%prec NEG` de la línea 41 hace que la regla tenga la prioridad del terminal `NEG` y por tanto mas prioridad que el terminal `-`. Esto hará que `eyapp` finalmente opte por la regla `exp: ’-’ exp` dando lugar a la interpretación `(-y)-z`.

La Cola

Después de la parte de la gramática, y separada de la anterior por el símbolo `%%`, sigue la parte en la que se suelen poner las rutinas de apoyo. Hay al menos dos rutinas de apoyo que el analizador sintáctico requiere le sean pasados como argumentos: la de manejo de errores y la de análisis léxico.

```
%%

sub _Error {
        exists $_[0]->YYData->{ERRMSG}
    and do {
        print $_[0]->YYData->{ERRMSG};
        delete $_[0]->YYData->{ERRMSG};
        return;
    };
    print "Syntax error.\n";
}

my $input;

sub _Lexer {
    my($parser)=shift;

    # topicalize $input
    for ($input) {
      s/^[ \t]+//;      # skip whites
      return('',undef) unless $input;

      return('NUM',$1) if s{^([0-9]+(?:\.[0-9]+)?)}{};
      return('VAR',$1) if s/^([A-Za-z][A-Za-z0-9_]*)//;
      return($1,$1)    if s/^(.)//s;
    }
}

sub Run {
    my($self)=shift;

    $input = shift;
    return $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error );
}
```

El Método `Run`

El método `Run` ilustra como se hace la llamada al método de análisis sintáctico generado, utilizando la técnica de llamada con argumentos con nombre y pasándole las referencias a las dos subrutinas (en Perl, es un convenio que si el nombre de una subrutina comienza por un guión bajo es que el autor la considera privada):

```
sub Run {
    my($self)=shift;

    $input = shift;
    return $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error );
}
```

El Atributo `YYData` y el Método `_Error`

El método `YYData` provee acceso a un hash que contiene los datos que están siendo analizados.

La subrutina de manejo de errores `_Error` imprime el mensaje de error proveído por el usuario, el cual, si existe, fué guardado en `$_[0]->YYData->{ERRMSG}`.

```
sub _Error {
        exists $_[0]->YYData->{ERRMSG}
    and do {
        print $_[0]->YYData->{ERRMSG};
        delete $_[0]->YYData->{ERRMSG};
        return;
    };
    print "Syntax error.\n";
```

El Método _Lexer

A continuación sigue el método que implanta el análisis léxico `_Lexer`.

```
45  sub _Lexer {
46      my($parser)=shift;
47
48      # topicalize $input
49      for ($input) {
50        s/^[ \t]+//;      # skip whites
51        return('',undef) unless $input;
52
53        return('NUM',$1) if s{^([0-9]+(?:\.[0-9]+)?)}{};
54        return('VAR',$1) if s/^([A-Za-z][A-Za-z0-9_]*)//;
55        return($1,$1)    if s/^(.)//s;
56      }
```

El bucle `for` constituye en este caso una ”frase hecha”: el efecto es hacer que `$_` sea un alias de `$input`. Se simplifica la escritura (obsérvese que no es necesario explicitar el operador de binding en las líneas 50-55, no teniendo que escribir `$input =~ s{^([0-9]+(?:\.[0-9]+)?)}{}`) y que la computación será mas eficiente al acceder a través de `$_` en vez de `$input`.

El bucle `for ($input)` se ejecutará mientras la cadena en `$input` no sea vacía, lo que ocurrirá cuando todos los terminales hayan sido consumidos. sin embargo es un ”falso for”: no hay iteración. El interior del bucle es ejecutado una sola vez en cada llamada.

Eliminamos los blancos iniciales (lo que en inglés se conoce por *trimming*) y a continuación vamos detectando los números, identificadores y los símbolos individuales.

En primer lugar se comprueba la existencia de datos. Si no es el caso, estamos ante el final de la entrada. Cuando el analizador léxico alcanza el final de la entrada debe devolver la pareja `('',undef)`.

Ejercicio 3.2.2. 1. *¿Quién es la variable índice en la línea 49?*

2. *¿Sobre quién ocurre el binding en las líneas 50-54?*

3. *¿Cual es la razón por la que* `$input` *se ve modificado aún cuando no aparece como variable para el binding en las líneas 50-54?*

4. *Dada la entrada ’*`4 * 3   `*’ con blancos al final: como termina el analizador léxico. ¿Funciona correctamente en ese caso?*

Compilación con eyapp

Construimos el módulo `CalcSyntax.pm` a partir del fichero `CalcSyntax.eyp` especificando la gramática, usando elejecutable `eyapp`:

```
pl@nereida:~/LEyapp/examples$ eyapp -m CalcSyntax CalcSyntax.eyp
pl@nereida:~/LEyapp/examples$ ls -ltr | tail -3
-rw-r--r-- 1 pl users   1545 2007-10-24 09:03 CalcSyntax.eyp
-rwxr-xr-x 1 pl users    329 2007-10-24 09:05 usecalcsyntax.pl
-rw-r--r-- 1 pl users   7848 2007-10-24 09:36 CalcSyntax.pm
```

Esta compilación genera el fichero `CalcSyntax.pm` conteniendo el analizador.

El script `eyapp` es un *frontend* al módulo `Parse::Eyapp`. Admite diversas formas de uso:

- `eyapp [options]` *grammar*`[.eyp]`

  Los sufijos `.eyp` io `.yp` son opcionales.

- `eyapp -`*V*

  Nos muestra la versión:

  ```
  pl@nereida:~/LEyapp/examples$ eyapp -V
  This is Parse::Eyapp version 1.081.
  ```

- `eyapp -`*h*

  Nos muestra la ayuda:

  ```
  pl@nereida:~/LEyapp/examples$ eyapp -h

  Usage:  eyapp [options] grammar[.yp]
    or    eyapp -V
    or    eyapp -h

      -m module   Give your parser module the name <module>
                  default is <grammar>
      -v          Create a file <grammar>.output describing your parser
      -s          Create a standalone module in which the driver is included
      -n          Disable source file line numbering embedded in your parser
      -o outfile  Create the file <outfile> for your parser module
                  Default is <grammar>.pm or, if -m A::Module::Name is
                  specified, Name.pm
      -t filename Uses the file <filename> as a template for creating the parser
                  module file.  Default is to use internal template defined
                  in Parse::Eyapp::Output
      -b shebang  Adds '#!<shebang>' as the very first line of the output file

      grammar     The grammar file. If no suffix is given, and the file
                  does not exists, .yp is added

      -V          Display current version of Parse::Eyapp and gracefully exits
      -h          Display this help screen
  ```

La opción *-o outfile* da el nombre del fichero de salida. Por defecto toma el nombre de la gramática, seguido del sufijo `.pm`. sin embargo, si hemos especificado la opción *-m A::Module::Name* el valor por defecto será *Name.pm*.

La Jerarquía de un Módulo y La Opción *-m module*

La opción *-m module* da el nombre al paquete o espacio de nombres o clase encapsulando el analizador. Por defecto toma el nombre de la gramática. En el ejemplo anterior podría haberse omitido. Sin embargo es necesaria cuando se esta desarrollando un módulo con un nombre complejo. Construyamos una distribución con `h2xs`:

```
$ h2xs -XA -n Calc::Syntax
Writing Calc-Syntax/lib/Calc/Syntax.pm
Writing Calc-Syntax/Makefile.PL
Writing Calc-Syntax/README
Writing Calc-Syntax/t/Calc-Syntax.t
Writing Calc-Syntax/Changes
Writing Calc-Syntax/MANIFEST
```

Ahora añadimos el fichero `.eyp` en el directorio de la librería y producimos el módulo `Syntax.pm` al compilarlo. Para darle al paquete el nombre `Calc::Syntax` usamos la opción `-m`:

```
$ cd Calc-Syntax/lib/Calc/
$ cp ~/LEyapp/examples/CalcSyntax.eyp .
$ eyapp -m Calc::Syntax  CalcSyntax.eyp
$  head -12 Syntax.pm | cat -n
     1  ##################################################################################
     2  #
     3  #    This file was generated using Parse::Eyapp version 1.081.
     4  #
     5  # (c) Parse::Yapp Copyright 1998-2001 Francois Desarmenien.
     6  # (c) Parse::Eyapp Copyright 2006 Casiano Rodriguez-Leon. Universidad de La Laguna.
     7  #        Don't edit this file, use source file "CalcSyntax.eyp" instead.
     8  #
     9  #             ANY CHANGE MADE HERE WILL BE LOST !
    10  #
    11  ##################################################################################
    12  package Calc::Syntax;
```

La opción que recomiendo para documentar el módulo es escribir la documentación en un fichero aparte `Calc/Syntax.pod`.

El Programa Cliente

A continuación escribimos el programa cliente:

```
$ cd ../..
$ mkdir scripts
$ cd scripts/
$ vi usecalcsyntax.pl
$ cat -n usecalcsyntax.pl
 1  #!/usr/bin/perl -w -I../lib
 2  use strict;
 3  use Calc::Syntax;
 4  use Carp;
 5
 6  sub slurp_file {
 7    my $fn = shift;
 8    my $f;
 9
10    local $/ = undef;
```

```
  if (defined($fn)) {
    open $f, $fn
  }
  else {
    $f = \*STDIN;
  }
  my $input = <$f>;
  return $input;
}

my $parser = Calc::Syntax->new();

my $input = slurp_file( shift() );
$parser->Run($input);
```

Ejecución

La ejecución muestra la *antiderivación a derechas construida por* `eyapp`:

```
$ cat prueba.exp
a=2*3

$ usecalcsyntax.pl prueba.exp
exp -> NUM (2)
exp -> NUM (3)
exp -> exp '*' exp
exp -> VAR '=' exp
line -> exp \n
input -> line *
```

Orden de Ejecución de las Acciones Semánticas

¿En que orden ejecuta `YYParse` las acciones? La respuesta es que el analizador generado por `eyapp` construye una derivación a derechas inversa y ejecuta las acciones asociadas a las reglas de producción que se han aplicado. Así, para la frase `2*3` la antiderivación es:

$$NUM + NUM \overset{NUM \leftarrow E}{\Longleftarrow} E + NUM \overset{NUM \leftarrow E}{\Longleftarrow} E + E \overset{E+E \leftarrow E}{\Longleftarrow} E$$

por tanto las acciones ejecutadas son las asociadas con las correspondientes reglas de producción:

1. La acción asociada con $E \rightarrow NUM$:

   ```
   NUM                  { print "exp -> NUM ($_[1])\n"; }
   ```

   Esta instancia de `exp` tiene ahora como atributo `2` (pasado por el analizador léxico).

2. De nuevo la acción asociada con $E \rightarrow NUM$:

   ```
   NUM                  { print "exp -> NUM ($_[1])\n"; }
   ```

   Esta nueva instancia de `exp` tiene como atributo `3`.

3. La acción asociada con $E \rightarrow E * E$:

   ```
    | exp '*' exp        { print "exp -> exp '*' exp\n"; }
   ```

Obsérvese que la antiderivación a derechas da lugar a un recorrido ascendente y a izquierdas del árbol:

$$E_3(E_1(NUM[2]), *, E_2(NUM[2]))$$

Los subíndices indican el orden de visita de las nodos/producciones.

3.3. Depuración de Errores

Las fuentes de error cuando se programa con una herramienta como `eyapp` son diversas. En esta sección trataremos tres tipos de error:

- Conflictos en la gramática: la gramática es ambigua o quizá - posiblemente debido a que en `eyapp` se usa un sólo símbolo de predicción - no queda claro para el generador que analizador sintáctico producir
- Ya no tenemos conflictos pero el analizador sintáctico generado no se comporta como esperabamos: no acepta frases correctas o construye un árbol erróneo para las mismas
- Ya hemos resuelto los conflictos y además el análisis sintáctico es correcto pero tenemos errores semánticos. Los errores se producen durante la ejecución de las acciones semánticas

Resolución de Conflictos

El siguiente programa `eyapp` contiene algunos errores. El lenguaje generado esta constituido por listas de `D`s (por declaraciones) seguidas de listas de `S`s (por sentencias) separadas por puntos y coma:

```
pl@nereida:~/LEyapp/examples$ cat -n Debug.eyp
 1  %token D S
 2
 3  %{
 4  our $VERSION = '0.01';
 5  %}
 6
 7  %%
 8  p:
 9                  ds ';' ss
10          |       ss
11  ;
12
13  ds:
14            D ';' ds
15          | D
16            {
17              print "Reducing by rule:\n";
18              print "\tds -> D\n";
19              $_[1];
20            }
21  ;
22
23  ss:
24            S ';' ss
25          | S
26  ;
27
28  %%
29
30  my $tokenline = 0;
31
32  sub _Error {
33          my $parser = shift;
34          my ($token) = $parser->YYCurval;
35          my ($what) = $token ? "input: '$token'" : "end of input";
```

```
        die "Syntax error near $what line num $tokenline\n";
}

my $input;

sub _Lexer {

        for ($input) {
          s{^(\s)}{} and $tokenline += $1 =~ tr{\n}{};
          return (’’,undef) unless $_;
          return ($1,$1) if s/^(.)//;
        }
        return (’’,undef);
}

sub Run {
        my ($self) = shift;

        $input = shift;

        return $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
                         yydebug => 0xF
  );
}
```

Al compilar este programa con `eyapp` produce un mensaje de advertencia anunciándonos la existencia de un conflicto.

```
pl@nereida:~/LEyapp/examples$ eyapp Debug.eyp
1 shift/reduce conflict (see .output file)
State 4: shifts:
  to state    8 with ’;’
```

La existencia de advertencias da lugar a la creación de un fichero `Debug.output` conteniendo información sobre la gramática y el analizador construido.

Veamos los contenidos del fichero:

```
pl@nereida:~/LEyapp/examples$ cat -n Debug.output
   1  Warnings:
   2  ---------
   3  1 shift/reduce conflict (see .output file)
   4  State 4: shifts:
   5    to state    8 with ’;’
   6
   7  Conflicts:
   8  ----------
   9  State 4 contains 1 shift/reduce conflict
  10
  11  Rules:
  12  ------
  13  0:      $start -> p $end
  14  1:      p -> ds ’;’ ss
  15  2:      p -> ss
  16  3:      ds -> D ’;’ ds
  17  4:      ds -> D
```

```
5:      ss -> S ';' ss
6:      ss -> S

States:
-------
State 0:

        $start -> . p $end      (Rule 0)

        D       shift, and go to state 4
        S       shift, and go to state 1

        p       go to state 2
        ss      go to state 3
        ds      go to state 5

.........................................
State 4:

        ds -> D . ';' ds        (Rule 3)
        ds -> D .       (Rule 4)

        ';'     shift, and go to state 8

        ';'     [reduce using rule 4 (ds)]

.........................................
State 8:

        ds -> D ';' . ds        (Rule 3)

        D       shift, and go to state 4

        ds      go to state 11

.........................................
State 12:

        p -> ds ';' ss .        (Rule 1)

        $default        reduce using rule 1 (p)


Summary:
--------
Number of rules         : 7
Number of terminals     : 4
Number of non-terminals : 4
Number of states        : 13
```

El problema según se nos anuncia ocurre en el estado 4. Como veremos mas adelante el analizador sintáctico generado por `Parse::Eyapp` es un autómata finito. Cada estado guarda información sobre las reglas que podrían aplicarse durante el análisis de la entrada. Veamos en detalle la información asociada con el estado 4:

```
55  State 4:
56
57          ds -> D . ’;’ ds        (Rule 3)
58          ds -> D .       (Rule 4)
59
60          ’;’     shift, and go to state 8
61
62          ’;’     [reduce using rule 4 (ds)]
63
```

Un estado es un conjunto de reglas de producción con un marcador en su parte derecha. La idea es que - si estamos en un estado dado - esas reglas son potenciales candidatos para la construcción del árbol sintáctico. Que lo sean o no dependerá de los terminales que se vean a continuación.

El punto que aparece en la parte derecha de una regla indica ”posición” de lectura. Así el hecho de que en el estado cuatro aparezcan los items:

```
57          ds -> D . ’;’ ds        (Rule 3)
58          ds -> D .       (Rule 4)
```

Significa que si estamos en este estado es por que se leyó una `D` y se espera ver un punto y coma.

El comentario de la línea 60 indica que si el siguiente terminal es ; podríamos ir al estado 8. Obsérvese que el estado 8 contiene un item de la forma `ds -> D ’;’ . ds`. La marca recuerda ahora que se vió una `D` y un punto y coma. El comentario de la línea 62:

```
62          ’;’     [reduce using rule 4 (ds)]
```

indica que `Parse::Eyapp` considera factible otro árbol cuando el token actual es punto y coma: Reducir por la regla `ds -> D`.

Para ilustrar el problema consideremos las frases `D;S` y `D;D;S`.

Para ambas frases, después de consumir la `D` el analizador irá al estado 4 y el terminal actual será punto y coma. Para la primera frase `D;S` la decisión correcta es utilizar (”reducir” en la jerga) la regla 4 `ds -> D`.

Para la segunda frase `D;D;S` la decisión correcta es conjeturar la regla 3 `ds -> D . ’;’ ds`. El analizador podría saber que regla es la correcta si consumiera el terminal que sigue al punto y coma: si es `S` se trata de la regla 4 y si es `D` de la regla 3. Los analizadores generados por Eyapp no ”miran” mas allá del siguiente terminal y por tanto no están en condiciones de decidir.

Cualquier solución a este tipo de conflictos implica una reformulación de la gramática modificando prioridades o reorganizando las reglas. Reescribiendo la regla para `ds` recursiva por la derecha desaparece el conflicto:

```
l@nereida:~/LEyapp/examples$ sed -ne ’/^ds:/,/^;/p’ Debug1.eyp | cat -n
 1  ds:
 2            ds ’;’ D
 3          | D
 4      {
 5        print "Reducing by rule:\n";
 6        print "\tds -> D\n";
 7        $_[1];
 8      }
 9  ;
```

Ahora ante una frase de la forma `D ; ...` siempre hay que reducir por `ds -> D`. La antiderivación a derechas para `D;D;S` es:

Derivación	Árbol
`D;D;S <= ds;D;S <= ds;S <= ds;ss <= p`	p(ds(ds(D),';',D),';',ss(S))
o	

Mientras que la antiderivación a derechas para `D;S` es:

Derivación	Árbol
`D;S <= ds;S <= ds;ss <= p`	p(ds(D),';',ss(S))

Recompilamos la nueva versión de la gramática. Las advertencias han desaparecido:

```
pl@nereida:~/LEyapp/examples$ eyapp Debug1.eyp
pl@nereida:~/LEyapp/examples$
```

Errores en la Construcción del Arbol

Escribimos el típico programa cliente:

```
pl@nereida:~/LEyapp/examples$ cat -n usedebug1.pl
#!/usr/bin/perl -w
# usetreebypass.pl prueba2.exp
use strict;
use Debug1;

sub slurp_file {
  my $fn = shift;
  my $f;

  local $/ = undef;
  if (defined($fn)) {
    open $f, $fn  or die "Can’t find file $fn!\n";
  }
  else {
    $f = \*STDIN;
  }
  my $input = <$f>;
  return $input;
}

my $input = slurp_file( shift() );

my $parser = Debug1->new();

$parser->Run($input);
```

y ejecutamos. Cuando damos la entrada `D;S` introduciendo algunos blancos y retornos de carro entre los terminales ocurre un mensaje de error:

```
casiano@cc111:~/LPLsrc/Eyapp/examples/debuggingtut$ usedebug1.pl
D

;
```

```
S
Reducing by rule:
        ds -> D
Syntax error near end of input line num 1
```

Como esta conducta es anómala activamos la opción `yydebug => 0xF` en la llamada a `YYParser`.

Es posible añadir un parámetro en la llamada a `YYParse` con nombre *yydebug* y valor el nivel de depuración requerido. Ello nos permite observar la conducta del analizador. Los posibles valores de depuración son:

Bit	Información de Depuración
0x01	Lectura de los terminales
0x02	Información sobre los estados
0x04	Acciones (shifts, reduces, accept . . .)
0x08	Volcado de la pila
0x10	Recuperación de errores

Veamos que ocurre cuando damos la entrada `D;S` introduciendo algunos blancos y retornos de carro entre los terminales:

```
pl@nereida:~/LEyapp/examples$ usedebug1.pl
D

;

S

----------------------------------------
In state 0:
Stack:[0]
Need token. Got >D<
Shift and go to state 4.
----------------------------------------
In state 4:
Stack:[0,4]
Don't need token.
Reduce using rule 4 (ds --> D): Reducing by rule:
        ds -> D
Back to state 0, then go to state 5.
----------------------------------------
In state 5:
Stack:[0,5]
Need token. Got ><
Syntax error near end of input line num 1
```

¿Que está pasando? Vemos que después de leer `D` el analizador sintáctico recibe un `end of file`. Algo va mal en las comunicaciones entre el analizador léxico y el sintáctico. Repasemos el analizador léxico:

```
pl@nereida:~/LEyapp/examples$ sed -ne '/sub.*_Lexer/,/^}/p' Debug1.eyp | cat -n
 1  sub _Lexer {
 2
 3          for ($input) {
```

```
 4            s{^(\s)}{} and $tokenline += $1 =~ tr{\n}{};
 5            return ('',undef) unless $_;
 6            return ($1,$1) if s/^(.)//;
 7         }
 8         return ('',undef);
 9  }
```

El error está en la línea 4. ¡Sólo se consume un blanco!. Escribimos una nueva versión `Debug2.eyp` corrigiendo el problema:

```
pl@nereida:~/LEyapp/examples$ sed -ne '/sub.*_Lexer/,/^}/p' Debug2.eyp | cat -n
 1  sub _Lexer {
 2
 3         for ($input) {
 4            s{^(\s+)}{} and $tokenline += $1 =~ tr{\n}{};
 5            return ('',undef) unless $_;
 6            return ($1,$1) if s/^(.)//;
 7         }
 8         return ('',undef);
 9  }
```

Ahora el análisis parece funcionar correctamente:

```
pl@nereida:~/LEyapp/examples$ usedebug2.pl
D

;

S

----------------------------------------
In state 0:
Stack:[0]
Need token. Got >D<
Shift and go to state 4.
----------------------------------------
In state 4:
Stack:[0,4]
Don't need token.
Reduce using rule 4 (ds --> D): Reducing by rule:
        ds -> D
Back to state 0, then go to state 5.
----------------------------------------
In state 5:
Stack:[0,5]
Need token. Got >;<
Shift and go to state 8.
----------------------------------------
In state 8:
Stack:[0,5,8]
Need token. Got >S<
Shift and go to state 1.
----------------------------------------
In state 1:
Stack:[0,5,8,1]
```

```
Need token. Got ><
Reduce using rule 6 (ss --> S): Back to state 8, then go to state 10.
----------------------------------------
In state 10:
Stack:[0,5,8,10]
Don't need token.
Reduce using rule 1 (p --> ds ; ss): Back to state 0, then go to state 2.
----------------------------------------
In state 2:
Stack:[0,2]
Shift and go to state 7.
----------------------------------------
In state 7:
Stack:[0,2,7]
Don't need token.
Accept.
```

Errores en las Acciones Semánticas

Un tercer tipo de error ocurre cuando el código de una acción semántica no se conduce como esperamos.

Las acciones semánticas son convertidas en métodos anónimos del objeto analizador sintáctico generado por `Parse::Eyapp`. Puesto que son subrutinas anónimas no podemos utilizar comandos de ruptura del depurador como

```
b nombre # parar cuando se entre en la subrutina ''nombre''
```

o bien

```
c nombredesub  # continuar hasta alcanzar la subrutina ''nombre''
```

Además el fichero cargado durante la ejecución es el `.pm` generado. El código en `Debug.pm` nos es ajeno - fue generado automáticamente por `Parse::Eyapp` - y puede resultar difícil encontrar en él las acciones semánticas que insertamos en nuestro esquema de traducción.

La estrategia a utilizar para poner un punto de parada en una acción semántica se ilustra en la siguiente sesión con el depurador. Primero arrancamos el depurador con el programa y usamos la opción `f file` para indicar el nuevo "fichero de vistas" por defecto:

```
pl@nereida:~/LEyapp/examples$ perl -wd usedebug2.pl

Loading DB routines from perl5db.pl version 1.28
Editor support available.

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

main::(usedebug2.pl:21):        my $input = slurp_file( shift() );
  DB<1> f Debug2.eyp
1       2       #line 3 "Debug2.eyp"
3
4:      our $VERSION = '0.01';
5
6       7       8       9       10
```

Ahora usamos la orden `b 18` para poner un punto de ruptura en la línea 18. El comando `l` muestra las correspondientes líneas del fichero `.eyp`:

```
  DB<2> b 18
  DB<3> l
11      12      13      14      15      16      #line 17 "Debug2.eyp"
17
18:b          print "Reducing by rule:\n";
19:           print "\tds -> D\n";
20:           $_[1];
```

La orden `c` (continuar) hace que el programa se ejecute hasta alcanzar el único punto de ruptura en la línea 18 de `Debug2.eyp`:

```
  DB<3> c
D


;


S

Debug2::CODE(0x85129d8)(Debug2.eyp:18):
18:           print "Reducing by rule:\n";
  DB<3> n
Reducing by rule:
```

En este momento podemos usar cualesquiera comandos del depurador para visualizar el estado interno de nuestro programa y determinar la causa de una potencial conducta anómala de la acción semántica:

```
Debug2::CODE(0x85129d8)(Debug2.eyp:19):
19:           print "\tds -> D\n";
  DB<3> x $_[0]{GRAMMAR}
0  ARRAY(0x8538360)
   0  ARRAY(0x855aa88)
      0  '_SUPERSTART'
      1  '$start'
      2  ARRAY(0x855ab60)
         0  'p'
         1  '$end'
      3  0
   1  ARRAY(0x855a890)
      0  'p_1'
      1  'p'
      2  ARRAY(0x855a8fc)
         0  'ds'
         1  ';'
         2  'ss'
      3  0
   2  ARRAY(0x855a800)
      0  'p_2'
      1  'p'
      2  ARRAY(0x855a830)
         0  'ss'
      3  0
   3  ARRAY(0x855a764)
      0  'ds_3'
      1  'ds'
      2  ARRAY(0x855a7a0)
```

```
         0  'ds'
         1  ';'
         2  'D'
      3  0
   4  ARRAY(0x85421d4)
      0  'ds_4'
      1  'ds'
      2  ARRAY(0x855a6e0)
         0  'D'
      3  0
   5  ARRAY(0x8538474)
      0  'ss_5'
      1  'ss'
      2  ARRAY(0x854f9c8)
         0  'S'
         1  ';'
         2  'ss'
      3  0
   6  ARRAY(0x85383b4)
      0  'ss_6'
      1  'ss'
      2  ARRAY(0x85383f0)
         0  'S'
      3  0
 DB<4>
```

Con el comando `c` podemos hacer que la ejecución continúe, esta vez hasta a finalización del programa

```
  DB<3> c
Debugged program terminated.  Use q to quit or R to restart,
  use o inhibit_exit to avoid stopping after program termination,
  h q, h R or h o to get additional info.
  DB<3>
```

3.4. Acciones y Acciones por Defecto

En el ejemplo anterior la acción semántica asociada con cada una de las reglas de producción es básicamente la misma: escribir la regla. Parece lógico intentar factorizar todo ese código común.

Las Acciones en `eyapp`

En `Parse::Eyapp` las acciones semánticas son convertidas en subrutinas anónimas. Mas bien en métodos anónimos dado que el primer argumento (`$_[0]`) de toda acción semántica es una referencia al objeto analizador sintáctico. Los restantes parámetros se corresponden con los *atributos de los símbolos* en la parte derecha de la regla de producción (`$_[1]` ...). Esto es, si la regla es:

$$A \rightarrow X_1 X_2 \dots X_n \quad \texttt{\{ action(@\_) \}}$$

cuando se "reduzca" por la regla $A \rightarrow X_1 X_2 \dots X_n$ la acción será llamada con argumentos

```
action(parserref, $X_1, $X_2, ..., $X_n)
```

donde `$X_1`, `$X_2`, etc. denotan los atributos de X_1, X_2, etc. y `parserref` es una referencia al objeto analizador sintáctico.

El valor retornado por la acción semántica es asignado como valor del atributo del lado izquierdo de la regla A.

Cuando una regla $A \rightarrow X_1X_2\ldots X_n$ no tiene acción semántica asociada se ejecuta la *acción por defecto*. La acción por defecto en `eyapp` es `{ $_[1] }`:

$$A \rightarrow X_1X_2\ldots X_n \quad \texttt{\{ \$\_[1] \}}$$

El efecto de tal acción es asignar al atributo de la variable sintáctica en la parte izquierda el atributo del primer símbolo en la parte derecha. Si la parte derecha fuera vacía se asigna `undef`.

La Directiva `%defaultaction`

La directiva `%defaultaction` permite especificar una acción por defecto alternativa. Observe esta nueva versión del programa anterior:

```
pl@nereida:˜/LEyapp/examples$ cat -n CalcSyntaxDefaultWithHOPLexer.eyp
# CalcSyntaxDefaultWithHOPLexer.eyp
%right  ’=’
%left   ’-’ ’+’
%left   ’*’ ’/’
%left   NEG
%right  ’ˆ’

%defaultaction {
  my $parser = shift;

  print $parser->YYLhs." --> ";
  my $i = 0;
  for ($parser->YYRightside) {
     print "$_";
     print "[$_[$i]]" if /NUM|VAR/;
     print " ";
     $i++;
  }
  print "\n";
}

%{
use HOP::Lexer qw(string_lexer);
%}
%%
line: (exp ’\n’)*
;

exp:
    NUM
  | VAR
  | VAR ’=’ exp
  | exp ’+’ exp
  | exp ’-’ exp
  | exp ’*’ exp
  | exp ’/’ exp
  | ’-’ exp %prec NEG
  | exp ’ˆ’ exp
  | ’(’ exp ’)’
;
```

El método `YYLhs` (línea 11) retorna el nombre de la variable sintáctica asociada con la regla actual. El método `YYRightside` (línea 13) devuelve la lista de nombres de los símbolos de la parte derecha de la regla de producción actual.

Observe como la modificación de la acción con defecto nos permite eliminar todas las acciones del cuerpo de la gramática.

Al ejecutar el cliente obtenemos una salida similar a la del programa anterior:

```
pl@nereida:~/LEyapp/examples$ eyapp CalcSyntaxDefaultWithHOPLexer.eyp
pl@nereida:~/LEyapp/examples$ cat -n prueba4.exp
     1  4*3
pl@nereida:~/LEyapp/examples$ usecalcsyntaxdefaultwithhoplexer.pl prueba4.exp | cat -n
     1  exp --> NUM[4]
     2  exp --> NUM[3]
     3  exp --> exp * exp
     4  line --> STAR-2
```

La Opción `-v` de `eyapp`

Posiblemente llame su atención la línea `line --> STAR-2` en la salida. Es consecuencia de la existencia de la línea 26 que contiene una aplicación del operador de repetición `*` a una expresión entre paréntesis:

```
26  line: (exp ’\n’)*
```

De hecho la gramática anterior es desplegada como se indica en el fichero `CalcSyntaxDefaultWithHOPLexer.outpu` obtenido al compilar usando la opción `-v` de `eyapp`:

```
pl@nereida:~/LEyapp/examples$ eyapp -v CalcSyntaxDefaultWithHOPLexer.eyp
pl@nereida:~/LEyapp/examples$ ls -ltr | tail -1
-rw-r--r-- 1 pl users   8363 2007-10-30 11:45 CalcSyntaxDefaultWithHOPLexer.output
pl@nereida:~/LEyapp/examples$ sed -ne ’/Rules/,/^$/p’ CalcSyntaxDefaultWithHOPLexer.output
Rules:
------
0:      $start -> line $end
1:      PAREN-1 -> exp ’\n’
2:      STAR-2 -> STAR-2 PAREN-1
3:      STAR-2 -> /* empty */
4:      line -> STAR-2
5:      exp -> NUM
6:      exp -> VAR
7:      exp -> VAR ’=’ exp
8:      exp -> exp ’+’ exp
9:      exp -> exp ’-’ exp
10:     exp -> exp ’*’ exp
11:     exp -> exp ’/’ exp
12:     exp -> ’-’ exp
13:     exp -> exp ’^’ exp
14:     exp -> ’(’ exp ’)’
```

Análisis Léxico con `HOP::Lexer`

En esta ocasión hemos elegido un método alternativo para construir el analizador léxico: Usar el módulo HOP::Lexer.

El módulo `HOP::Lexer` es cargado en las líneas 22-24. Los delimitadores `%{` y `%}` permiten introducir código Perl en cualquier lugar de la sección de cabecera.

Estudie la documentación del módulo `HOP::Lexer` y compare la solución que sigue con la dada en la sección anterior (véase 3.2). Este es el código en la sección de cola:

```
42  %%
43
44  sub _Error {
..      .................................
52  }
53
54  sub Run {
55      my($self)=shift;
56      my $input = shift;
57
58      my @lexemes = (
59        ['SPACE', qr{[ \t]+}, sub { () } ],
60        ['NUM',   qr{[0-9]+(?:\.[0-9]+)?}],
61        ['VAR',   qr{[A-Za-z][A-Za-z0-9_]*}],
62        ["\n",    qr{\n}],
63        ['ANY',   qr{.},
64                    sub {
65                      my ($label, $val) = @_;
66                      return [ $val, $val ];
67                    }
68        ]
69
70      );
71      my $lexer = string_lexer($input, @lexemes);
72
73      return $self->YYParse(
74        yylex => sub {
75          my $parser = shift;
76          my $token = $lexer->();
77          return @$token if defined($token);
78          return ('', undef);
79        },
80        yyerror => \&_Error,
81        #yydebug => 0xF
82      );
83  }
```

La Opción `yydebug`

Cuando `YYParse` se llama con la opción `yydebug` activada (descomente la línea 81 si quiere ver el efecto) la ejecución del analizador produce un informe detallado (version de eyapp 0.83 o superior) de su avance durante el análisis:

```
pl@nereida:~/LEyapp/examples$ usecalcsyntaxdefaultwithhoplexer.pl prueba4.exp >& warn
pl@nereida:~/LEyapp/examples$ head -30 warn
----------------------------------------
In state 0:
Stack:[0]
Don't need token.
Reduce using rule 3 (STAR-2 --> /* empty */): Back to state 0, then go to state 1.
----------------------------------------
In state 1:
Stack:[0,1]
Need token. Got >NUM<
Shift and go to state 4.
```

```
----------------------------------------
In state 4:
Stack:[0,1,4]
Don't need token.
Reduce using rule 5 (exp --> NUM): Back to state 1, then go to state 5.
----------------------------------------
In state 5:
Stack:[0,1,5]
Need token. Got >*<
Shift and go to state 13.
----------------------------------------
In state 13:
Stack:[0,1,5,13]
Need token. Got >NUM<
Shift and go to state 4.
----------------------------------------
In state 4:
Stack:[0,1,5,13,4]
Don't need token.
Reduce using rule 5 (exp --> NUM): Back to state 13, then go to state 21.
```

3.5. Traducción de Infijo a Postfijo

Las acción por defecto (en yacc `$$ = $1`) puede ser modificada de manera parecida a como lo hace la variable `$::RD_AUTOACTION` en `Parse::RecDescent` usando la directiva `%defaultaction`:

```
%defaultaction { perl code }
```

El código especificado en la acción por defecto se ejecutará siempre que se produzca una reducción por una regla en la que no se haya explicitado código.

El siguiente ejemplo traduce una expresión en infijo como `a=b*-3` a una en postfijo como `a b 3 NEG * =` :

```
# File Postfix.yp
%right  '='
%left   '-' '+'
%left   '*' '/'
%left   NEG
%{
use IO::Interactive qw(interactive);
%}

%defaultaction {
  my $self = shift;

  return  "$_[0]" if (@_==1);  # NUM and VAR
  return  "$_[0] $_[2] $_[1]"  if (@_==3); # other operations
  die "Fatal Error\n";
}

%%
line: (exp  { print "$_[1]\n" })?
;

exp:        NUM
```

```
        |   VAR
        |   VAR '=' exp
        |   exp '+' exp
        |   exp '-' exp
        |   exp '*' exp
        |   exp '/' exp
        |   '-' exp %prec NEG { "$_[2] NEG" }
        |   '(' exp ')' { $_[2] }
;

%%

sub _Error {
  my($token)=$_[0]->YYCurval;
  my($what)= $token ? "input: '$token'" : "end of input";
  my @expected = $_[0]->YYExpect();

  local $" = ', ';
  die "Syntax error near $what. Expected one of these tokens: @expected\n";
}

my $x;

sub _Lexer {
  my($parser)=shift;

  for ($x) {
    s/^\s+//;
    $_ eq '' and return('',undef);

    s/^([0-9]+(?:\.[0-9]+)?)//   and return('NUM',$1);
    s/^([A-Za-z][A-Za-z0-9_]*)// and return('VAR',$1);
    s/^(.)//s                    and return($1,$1);
  }
}

sub Run {
  my($self)=shift;
  print {interactive} "Write an expression: ";
  $x = <>;
  $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
    #yydebug => 0xFF
  );
}
```

3.6. Práctica: Gramática Simple en `Parse::Eyapp`

Escriba usando `Parse::Eyapp` un analizador sintáctico `PL::Simple` para la siguiente gramática:

0	p → ds ss
1	p → ss
2	ds → d ';' ds
3	ds → d ';'
4	d → INT il
5	d → STRING il
6	ss → s ';' ss
7	ss → s
8	s → ID '=' e
9	s → PRINT nel
10	s → ϵ
11	e → e '+' t
12	e → e '-' t
13	e → t
14	t → t '*' f
15	t → t '/' f
16	t → f
17	f → '(' e ')'
18	f → ID
19	f → NUM
20	f → STR
21	f → FUNCTIONID '(' el ')'
22	il → ID ',' il
23	il → ID
24	el → nel
25	el → ϵ
26	nel → e ',' nel
27	nel → e

El terminal `ID` corresponde a identificador y `NUM` a un número entero sin signo. El terminal `STR` a cadenas de dobles comillas. El terminal `FUNCTIONID` viene dado por la expresión regular `min|max|concat`. Declare dichos terminales en la cabecera usando la directiva `%token` (lea la documentación de `Parse::Eyapp`). Una directiva como:

```
%token ID FUNCTIONID NUM
```

declara los terminales pero no les da prioridad.

- Describa mediante ejemplos el lenguaje generado por esta gramática. ¿Es ambigua la gramática?
- Admita comentarios como en `C`.
- Almacene los números de línea asociados con los terminales y utilice esa información en los diagnósticos de error.
- Para la escritura de las reglas del forma $s \rightarrow \epsilon$ repase los consejos dados en el párrafo 3.2
- Expanda la gramática para que incluya operaciones lógicas (`a < b`, `a == b`, etc.). Observe que tales expresiones tiene menor prioridad que los operadores aditivos: `a = b < c + 2` es interpretada como `a = b < (c + 2)`
- Escriba la documentación en un fichero de documentación aparte `PL/Simple.pod`.
- Escriba las pruebas. En las pruebas asegúrese de poner tanto entradas correctas como erróneas.
- Adjunte un estudio de cubrimiento con su práctica. Añada los ficheros con el informe al `MANIFEST`. Ubiquelos en un directorio de informes.

3.7. El Método `YYName` y la Directiva `%name`

Construyendo una Representación del Árbol

En `eyapp` toda regla de producción tiene un nombre. El nombre de una regla puede ser dado explícitamente por el programador mediante la directiva `%name`. Por ejemplo, en el fragmento de código que sigue se da el nombre `ASSIGN` a la regla `exp: VAR ’=’ exp`. Si no se da un nombre explícito, la regla tendrá un nombre implícito. El nombre implícito de una regla se forma concatenando el nombre de la variable sintáctica en la parte izquierda con un subguión y el número de orden de la regla de producción: `Lhs_#`. Evite dar nombres con ese patrón a sus reglas de producción. Los patrones de la forma `/${lhs}_\d+$/` donde `${lhs}` es el nombre de la variable sintáctica estan reservados por `eyapp`.

```
pl@nereida:~/LEyapp/examples$ cat -n Lhs.eyp
 1  # Lhs.eyp
 2
 3  %right    ’=’
 4  %left     ’-’ ’+’
 5  %left     ’*’ ’/’
 6  %left   NEG
 7
 8  %defaultaction {
 9    my $self = shift;
10    my $name = $self->YYName();
11    bless { children => [ grep {ref($_)} @_] }, $name;
12  }
13
14  %%
15  input:
16              /* empty */
17                { [] }
18          |   input line
19                {
20                  push @{$_[1]}, $_[2] if defined($_[2]);
21                  $_[1]
22                }
23  ;
24
25  line:     ’\n’       { }
26          | exp ’\n’   {  $_[1] }
27  ;
28
29  exp:
30              NUM   { $_[1] }
31          |   VAR   { $_[1] }
32          |   %name ASSIGN
33              VAR ’=’ exp
34          |   %name PLUS
35              exp ’+’ exp
36          |   %name MINUS
37              exp ’-’ exp
38          |   %name TIMES
39              exp ’*’ exp
40          |   %name DIV
41              exp ’/’ exp
42          |   %name UMINUS
```

```
43              ’-’ exp %prec NEG
44          |  ’(’ exp ’)’  { $_[2] }
45  ;
```

Dentro de una acción semántica el nombre de la regla actual puede ser recuperado mediante el método `YYName` del analizador sintáctico.

La acción por defecto (lineas 8-12) computa como atributo de la parte izquierda una referencia a un objeto que tiene como clase el nombre de la regla. Ese objeto tiene un atributo `children` que es una referencia a la lista de hijos del nodo. La llamada a `grep`

```
11    bless { children => [ grep {ref($_)} @_] }, $name;
```

tiene como efecto obviar (no incluir) aquellos hijos que no sean referencias. Observe que el analizador léxico sólo retorna referencias a objetos para los terminales `NUM` y `VAR`.

La Cola

```
47  %%
48
49  sub _Error {
50          exists $_[0]->YYData->{ERRMSG}
51      and do {
52          print $_[0]->YYData->{ERRMSG};
53          delete $_[0]->YYData->{ERRMSG};
54          return;
55      };
56      print "Syntax error.\n";
57  }
58
59  sub _Lexer {
60      my($parser)=shift;
61
62      for ($parser->YYData->{INPUT}) {
63          s/^[ \t]+//;
64          return(’’,undef) unless $_;
65          s/^([0-9]+(?:\.[0-9]+)?)//
66                  and return(’NUM’, bless { attr => $1}, ’NUM’);
67          s/^([A-Za-z][A-Za-z0-9_]*)//
68                  and return(’VAR’,bless {attr => $1}, ’VAR’);
69          s/^(.)//s
70                  and return($1, $1);
71      }
72      return(’’,undef);
73  }
74
75  sub Run {
76      my($self)=shift;
77
78      $self->YYData->{INPUT} = <>;
79      return $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error  );
80  }
```

Para los terminales `NUM` y `VAR` el analizador léxico devuelve una referencia a un objeto. No así para el resto de los terminales.

El Programa Cliente

```
pl@nereida:~/LEyapp/examples$ cat -n uselhs.pl
     1  #!/usr/bin/perl -w
     2  use Lhs;
     3  use Data::Dumper;
     4
     5  $parser = new Lhs();
     6  my $tree = $parser->Run;
     7  $Data::Dumper::Indent = 1;
     8  if (defined($tree)) { print Dumper($tree); }
     9  else { print "Cadena no válida\n"; }
```

Ejecución

Al darle la entrada `a=(2+3)*b` el analizador produce el árbol

```
ASSIGN(TIMES(PLUS(NUM[2],NUM[3]), VAR[b]))
```

Veamos el resultado de una ejecución:

```
pl@nereida:~/LEyapp/examples$ uselhs.pl
a=(2+3)*b
$VAR1 = [
  bless( {
    'children' => [
      bless( { 'attr' => 'a' }, 'VAR' ),
      bless( {
        'children' => [
          bless( {
            'children' => [
              bless( { 'attr' => '2' }, 'NUM' ),
              bless( { 'attr' => '3' }, 'NUM' )
            ]
          }, 'PLUS' ),
          bless( { 'attr' => 'b' }, 'VAR' )
        ]
      }, 'TIMES' )
    ]
  }, 'ASSIGN' )
];
```

Cambiando el Nombre de la Regla Dinámicamente

Es posible cambiar en tiempo de ejecución el nombre de una regla. Observe la siguiente variante del ejemplo anterior en el que se cambia - en tiempo de ejecución - el nombre de la regla del menos binario.

```
exp:
            NUM   { $_[1] }
        |   VAR   { $_[1] }
        |   %name ASSIGN
            VAR '=' exp
        |   %name PLUS
            exp '+' exp
        |   %name MINUS
```

```
      exp '-' exp
       {
        my $self = shift;
        $self->YYName('SUBSTRACT'); # rename it
        $self->YYBuildAST(@_); # build the node
       }
    | %name TIMES
      exp '*' exp
    | %name DIV
      exp '/' exp
    | %name UMINUS
      '-' exp %prec NEG
    | '(' exp ')'  { $_[2] }
;
```

Al ejecutar el cliente observamos como los nodos han sido bendecidos en la clase `SUBSTRACT`:

```
pl@nereida:~/LEyapp/examples$ useyynamedynamic.pl
2-b
$VAR1 = [
  bless( {
    'children' => [
      bless( {
        'attr' => '2'
      }, 'NUM' ),
      bless( {
        'attr' => 'b'
      }, 'VAR' )
    ]
  }, 'SUBSTRACT' )
];
```

3.8. Construyendo el Arbol de Análisis Sintactico Mediante Directivas

Directivas para la Construcción del Arbol Sintáctico

La directiva `%tree` hace que `Parse::Eyapp` genere una estructura de datos que representa al árbol sintáctico. Haciendo uso de las directivas adecuadas podemos controlar la forma del árbol.

Los nodos del árbol sintáctico generado por la producción $A \rightarrow X_1 \dots X_n$ son objetos (de tipo hash) bendecidos en una clase con nombre `A_#`, esto es, el de la variable sintáctica en el lado izquierdo seguida de un número de orden. Los nodos retornados por el analizador léxico son bendecidos en la clase especial `TERMINAL`. Los nodos tienen además un atributo `children` que referencia a la lista de nodos hijos $X_1 \dots X_n$ del nodo.

La directiva `%name CLASSNAME` permite modificar el nombre por defecto de la clase del nodo para que sea `CLASSNAME`. Por ejemplo, cuando es aplicada en el siguiente fragmento de código:

```
exp:
 ...........
 | %name PLUS
  exp '+' exp
```

Hace que el nodo asociado con la regla $exp \rightarrow exp + exp$ pertenezca a la clase `PLUS` en vez de a una clase con un nombre poco significativo como `exp_25`:

```
pl@nereida:~/LEyapp/examples$ cat -n CalcSyntaxTree.eyp
 1  # CalcSyntaxTree.eyp
 2  %right  '='
 3  %left   '-' '+'
 4  %left   '*' '/'
 5  %left   NEG
 6  %right  '^'
 7
 8  %tree
 9
10  %{
11  sub TERMINAL::info {
12    my $self = shift;
13
14    $self->attr;
15  }
16
17  %}
18  %%
19
20  line:
21      %name EXP
22      exp '\n'
23  ;
24
25  exp:
26      %name NUM
27      NUM
28    | %name VAR
29      VAR
30    | %name ASSIGN
31      VAR '=' exp
32    | %name PLUS
33      exp '+' exp
34    | %name MINUS
35      exp '-' exp
36    | %name TIMES
37      exp '*' exp
38    | %name DIV
39      exp '/' exp
40    | %name UMINUS
41      '-' exp %prec NEG
42    | %name EXP
43      exp '^' exp
44    | %name PAREN
45     '(' exp ')'
46  ;
47
48  %%
.. # Exactamente igual que en el ejemplo anterior
```

La forma que tiene el árbol construido mediante la directiva `%tree` puede ser modificada. Por defecto, todos aquellos terminales que aparecen en definidos en el programa `eyapp` mediante el uso de apóstrofes son eliminados del árbol. En la jerga "Eyapp" un *token sintáctico* es uno que será eliminado (podado) del árbol sintáctico. Por contra, un *token semántico* es uno que aparecerá en el árbol. Por defecto los tokens definidos simbólicamente como `NUM` o `VAR` son ”semanticos” y aparecerán como un nodo de tipo `TERMINAL` en el árbol de análisis sintáctico. Estos estatus por defecto pueden ser modificados mediante las directivas `%semantic token` y `%syntactic token`.

Los token sintácticos no forman parte del árbol construido. Asi pues, en el ejemplo que nos ocupa, los terminales `'='`, `'-'`, `'+'`, `'*'` y `'/'` serán, por defecto, eliminados del árbol sintáctico.

El Cliente

```
pl@nereida:~/LEyapp/examples$ cat -n usecalcsyntaxtree.pl
#!/usr/bin/perl -w
# usecalcsyntaxtree.pl prueba2.exp
use strict;
use CalcSyntaxTree;

sub slurp_file {
  my $fn = shift;
  my $f;

  local $/ = undef;
  if (defined($fn)) {
    open $f, $fn
  }
  else {
    $f = \*STDIN;
  }
  my $input = <$f>;
  return $input;
}

my $parser = CalcSyntaxTree->new();

my $input = slurp_file( shift() );
my $tree = $parser->Run($input);

$Parse::Eyapp::Node::INDENT = 2;
print $tree->str."\n";
```

El método `str` de `Parse::Eyapp::Node` devuelve una cadena describiendo el árbol de análisis sintáctico enraizado en el nodo que se ha pasado como argumento.

El método `str` cuando visita un nodo comprueba la existencia de un método `info` para la clase del nodo. Si es así el método será llamado. Obsérvese como en la cabecera del programa `Eyapp` proveemos un método `info` para los nodos `TERMINAL`.

```
%tree

%{
sub TERMINAL::info {
  my $self = shift;

  $self->attr;
}

%}
%%
```

Los nodos `TERMINAL` del árbol sintáctico tienen un atributo `attr` que guarda el valor pasado para ese terminal por el analizador léxico.

La variable de paquete `$Parse::Eyapp::Node::INDENT` controla el formato de presentación usado por `Parse::Eyapp::Node::str`: Si es 2 cada paréntesis cerrar que este a una distancia mayor de `$Parse::Eyapp::Node::LINESEP` líneas será comentado con el tipo del nodo.

Ejecución

```
pl@nereida:~/LEyapp/examples$ cat prueba2.exp
a=(2+b)*3
pl@nereida:~/LEyapp/examples$ usecalcsyntaxtree.pl prueba2.exp | cat -n
     1
     2  EXP(
     3    ASSIGN(
     4      TERMINAL[a],
     5      TIMES(
     6        PAREN(
     7          PLUS(
     8            NUM(
     9              TERMINAL[2]
    10            ),
    11            VAR(
    12              TERMINAL[b]
    13            )
    14          ) # PLUS
    15        ) # PAREN,
    16        NUM(
    17          TERMINAL[3]
    18        )
    19      ) # TIMES
    20    ) # ASSIGN
    21  ) # EXP
```

Estructura Interna de los Árboles Sintácticos

Para entender mejor la representación interna que `Parse::Eyapp` hace del árbol ejecutemos de nuevo el programa con la ayuda del depurador:

```
pl@nereida:~/LEyapp/examples$ perl -wd usecalcsyntaxtree.pl prueba2.exp
main::(usecalcsyntaxtree.pl:21):        my $parser = CalcSyntaxTree->new();
  DB<1> l 21-27   # Listamos las líneas de la 1 a la 27
21==>   my $parser = CalcSyntaxTree->new();
22
23:     my $input = slurp_file( shift() );
24:     my $tree = $parser->Run($input);
25
26:     $Parse::Eyapp::Node::INDENT = 2;
27:     print $tree->str."\n";
  DB<2> c 27  # Continuamos la ejecución hasta alcanzar la línea 27
main::(usecalcsyntaxtree.pl:27):        print $tree->str."\n";
  DB<3> x $tree
0  EXP=HASH(0x83dec7c)   # El nodo raíz pertenece a la clase EXP
   ’children’ => ARRAY(0x83df12c) # El atributo ’children’ es una referencia a un array
      0  ASSIGN=HASH(0x83decdc)
         ’children’ => ARRAY(0x83df114)
```

```
0  TERMINAL=HASH(0x83dec40) # Nodo construido para un terminal
   'attr' => 'a' # Atributo del terminal
   'children' => ARRAY(0x83dee38) # Para un terminal debera ser una lista
        empty array               # vacía
   'token' => 'VAR'
1  TIMES=HASH(0x83df084)
   'children' => ARRAY(0x83df078)
      0  PAREN=HASH(0x83ded9c)
         'children' => ARRAY(0x83defac)
            0  PLUS=HASH(0x83dee74)
               'children' => ARRAY(0x83deef8)
                  0  NUM=HASH(0x83dedc0)
                     'children' => ARRAY(0x832df14)
                        0  TERMINAL=HASH(0x83dedfc)
                           'attr' => 2
                           'children' => ARRAY(0x83dedd8)
                                empty array
                           'token' => 'NUM'
                  1  VAR=HASH(0x83decd0)
                     'children' => ARRAY(0x83dee2c)
                        0  TERMINAL=HASH(0x83deec8)
                           'attr' => 'b'
                           'children' => ARRAY(0x83dee98)
                                empty array
                           'token' => 'VAR'
      1  NUM=HASH(0x83dee44)
         'children' => ARRAY(0x83df054)
            0  TERMINAL=HASH(0x83df048)
               'attr' => 3
               'children' => ARRAY(0x83dece8)
                    empty array
               'token' => 'NUM'
```

Observe que un nodo es un objeto implantado mediante un hash. El objeto es bendecido en la clase/paquete que se especifico mediante la directiva `%name`. Todas estas clases heredan de la clase `Parse::Eyapp::Node` y por tanto disponen de los métodos proveídos por esta clase (en particular el método `str` usado en el ejemplo). Los nodos disponen de un atributo `children` que es una referencia a la lista de nodos hijo. Los nodos de la clase `TERMINAL` son construidos a partir de la pareja `(token, atributo)` proveída por el analizador léxico. Estos nodos disponen además del atributo `attr` que encapsula el atributo del terminal.

Terminales Semánticos y Terminales Sintácticos

Observe como en el árbol anterior no existen nodos `TERMINAL[+]` ni `TERMINAL[*]` ya que estos terminales fueron definidos mediante apóstrofes y son - por defecto - terminales sintácticos.

Ejercicio 3.8.1. *Modifique el programa eyapp anterior para que contenga la nueva línea 2:*

```
pl@nereida:~/LEyapp/examples$ head -9 CalcSyntaxTree3.eyp | cat -n
     1  # CalcSyntaxTree.eyp
     2  %semantic token '=' '-' '+' '*' '/' '^'
     3  %right  '='
     4  %left   '-' '+'
     5  %left   '*' '/'
     6  %left   NEG
```

```
     7  %right  '^'
     8
     9  %tree
```

Escriba un programa cliente y explique el árbol que resulta para la entrada `a=(2+b)*3`*:*

```
pl@nereida:~/LEyapp/examples$ cat prueba2.exp
a=(2+b)*3
pl@nereida:~/LEyapp/examples$ usecalcsyntaxtree3.pl prueba2.exp | cat -n
 1
 2  EXP(
 3    ASSIGN(
 4      TERMINAL[a],
 5      TERMINAL[=],
 6      TIMES(
 7        PAREN(
 8          PLUS(
 9            NUM(
10              TERMINAL[2]
11            ),
12            TERMINAL[+],
13            VAR(
14              TERMINAL[b]
15            )
16          ) # PLUS
17        ) # PAREN,
18        TERMINAL[*],
19        NUM(
20          TERMINAL[3]
21        )
22      ) # TIMES
23    ) # ASSIGN
24  ) # EXP
```

Ambiguedades y Arboles Sintácticos

Modifiquemos la precedencia de operadores utilizada en el ejemplo anterior:

```
pl@nereida:~/LEyapp/examples$ head -6 CalcSyntaxTree2.eyp | cat -n
     1  # CalcSyntaxTree.eyp
     2  %right  '='
     3  %left   '+'
     4  %left   '-' '*' '/'
     5  %left   NEG
     6  %right  '^'
```

Compilamos la gramática y la ejecutamos con entrada `a=2-b*3`:

```
pl@nereida:~/LEyapp/examples$ cat prueba3.exp
a=2-b*3
pl@nereida:~/LEyapp/examples$  usecalcsyntaxtree2.pl prueba3.exp

EXP(
  ASSIGN(
    TERMINAL[a],
    TIMES(
```

```
      MINUS(
        NUM(
          TERMINAL[2]
        ),
        VAR(
          TERMINAL[b]
        )
      ) # MINUS,
      NUM(
        TERMINAL[3]
      )
    ) # TIMES
  ) # ASSIGN
) # EXP
```

Ejercicio 3.8.2. *En el programa eyapp anterior modifique las prioridades establecidas para los terminales y muestre los árboles sintácticos formados. Pruebe los siguientes experimentos.*

- *Haga el menos binario* `-` *asociativo a derechas*
- *Haga que el menos binario tenga mayor prioridad que el menos unario*
- *Introduzca en la gramática los operadores de comparación (<, >, etc.). ¿Cual es la prioridad adecuada para estos operadores? ¿Que asociatividad es correcta para los mismos? Observe los árboles formados para frases como* `a = b < c * 2`. *Contraste como interpreta un lenguaje típico (Java, C, Perl) una expresión como esta.*

El Método `YYIssemantic`

Es posible consultar o cambiar dinámicamente el estatus de un terminal usando el método `YYIssemantic`:

```
pl@nereida:~/LEyapp/examples$ sed -ne '/sub Run/,$p' CalcSyntaxTreeDynamicSemantic.eyp | cat -
sub Run {
    my($self)=shift;

    $input = shift;
    $self->YYIssemantic('+', 1);
    return $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error );
}
```

Los árboles generados al ejecutar el programa contienen nodos `TERMINAL[+]`):

```
pl@nereida:~/LEyapp/examples$ usecalcsyntaxtreedynamicsemantic.pl
2+3+4

EXP(
  PLUS(
    PLUS(
      NUM(
        TERMINAL[2]
      ),
      TERMINAL[+],
      NUM(
        TERMINAL[3]
      )
    ) # PLUS,
```

```
    TERMINAL[+],
    NUM(
      TERMINAL[4]
    )
  ) # PLUS
) # EXP
```

3.9. La Maniobra de `bypass`

En el programa eyapp visto en la sección anterior generabamos una representación del árbol de análisis sintáctico utilizando la directiva `%tree`. Asi para la entrada:

```
pl@nereida:~/LEyapp/examples$ cat prueba2.exp
a=(2+b)*3
```

Obteníamos el árbol:

```
pl@nereida:~/LEyapp/examples$ usecalcsyntaxtree.pl prueba2.exp | cat -n
     1
     2  EXP(
     3    ASSIGN(
     4      TERMINAL[a],
     5      TIMES(
     6        PAREN(
     7          PLUS(
     8            NUM(
     9              TERMINAL[2]
    10            ),
    11            VAR(
    12              TERMINAL[b]
    13            )
    14          ) # PLUS
    15        ) # PAREN,
    16        NUM(
    17          TERMINAL[3]
    18        )
    19      ) # TIMES
    20    ) # ASSIGN
    21  ) # EXP
```

Es evidente que el árbol contiene información redundante: el nodo `PAREN` puede ser obviado sin que afecte a la interpretación del árbol.

La directiva `%tree` es un caso particular de la directiva `%default action`. De hecho usarla es equivalente a escribir:

```
%default action { goto &Parse::Eyapp::Driver::YYBuildAST }
```

La subrutina `Parse::Eyapp::Driver::YYBuildAST` se encarga de la construcción del nodo correspondiente a la regla de producción actual. De hecho el método retorna una referencia al objeto nodo recién construído.

Puesto que la directiva `%tree` es un caso particular de la directiva `%default action` la ubicación de acciones semánticas explícitas para una regla puede ser usada para alterar la forma del árbol de análisis. Un ejemplo de uso lo motiva el comentario anterior sobre los nodos `PAREN`. Sería deseable que tales nodos no formaran parte del árbol.

Observe la nueva versión de la gramática en la sección anterior (página 321). El programa `eyapp` trabaja bajo la directiva `%tree`. Se ha introducido una acción explícita en la línea 21.

```
pl@nereida:~/LEyapp/examples$ sed -ne '/^exp:/,/^;/p' CalcSyntaxTree4.eyp | cat -n
     1  exp:
     2      %name NUM
     3      NUM
     4    | %name VAR
     5      VAR
     6    | %name ASSIGN
     7      VAR '=' exp
     8    | %name PLUS
     9      exp '+' exp
    10    | %name MINUS
    11      exp '-' exp
    12    | %name TIMES
    13      exp '*' exp
    14    | %name DIV
    15      exp '/' exp
    16    | %name UMINUS
    17      '-' exp %prec NEG
    18    | %name EXP
    19      exp '^' exp
    20    | %name PAREN
    21     '(' exp ')' { $_[2] }
    22  ;
```

Como consecuencia de la acción de la línea 21 la referencia al nodo hijo asociado con `exp` es retornado y el nodo `PAREN` desparece del árbol:

```
pl@nereida:~/LEyapp/examples$ usecalcsyntaxtree4.pl prueba2.exp

EXP(
  ASSIGN(
    TERMINAL[a],
    TIMES(
      PLUS(
        NUM(
          TERMINAL[2]
        ),
        VAR(
          TERMINAL[b]
        )
      ) # PLUS,
      NUM(
        TERMINAL[3]
      )
    ) # TIMES
  ) # ASSIGN
) # EXP
```

A esta maniobra, consistente en obviar un nodo sustituyéndolo por el único hijo que realmente importa, la denominaremos `bypass` de un nodo.

3.10. Salvando la Información en los Terminales Sintácticos

La directiva `%tree`

La directiva `%tree` permite que `Parse::Eyapp` genere al árbol sintáctico. Haciendo uso de las directivas adecuadas podemos controlar la forma del árbol. La directiva `%tree` es una alternativa a la directiva `%metatree` y es incompatible con esta última. El objetivo es lograr una representación adecuada del árbol sintáctico y dejar para fases posteriores la decoración del mismo.

```
nereida:~/src/perl/YappWithDefaultAction/examples> cat -n Rule6.yp
 1  %{
 2  use Data::Dumper;
 3  %}
 4  %right  ’=’
 5  %left   ’-’ ’+’
 6  %left   ’*’ ’/’
 7  %left   NEG
 8  %tree
 9
10  %%
11  line: exp  { $_[1] }
12  ;
13
14  exp:      %name NUM
15              NUM
16          | %name VAR
17            VAR
18          | %name ASSIGN
19            VAR ’=’ exp
20          | %name PLUS
21            exp ’+’ exp
22          | %name MINUS
23            exp ’-’ exp
24          | %name TIMES
25            exp ’*’ exp
26          | %name DIV
27            exp ’/’ exp
28          | %name UMINUS
29            ’-’ exp %prec NEG
30          |   ’(’ exp ’)’  { $_[2] } /* Let us simplify a bit the tree */
31  ;
32
33  %%
34
35  sub _Error {
..    ..........
43  }
44
45  sub _Lexer {
..    ..........
62  }
63
64  sub Run {
65      my($self)=shift;
66      $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
67                      #yyprefix => "Rule6::",
68                      #yydebug =>0xFF
69                    );
```

```
70  }
```

Compilación Separada

Para compilar un programa separado `Parse::Eyapp` usamos el guión `eyapp`:

```
nereida:~/src/perl/YappWithDefaultAction/examples> eyapp Rule6
nereida:~/src/perl/YappWithDefaultAction/examples> ls -ltr | tail -1
-rw-rw---- 1 pl users   5475 2006-11-06 13:53 Rule6.pm
```

Terminales sintácticos y Terminales Semánticos

A diferencia de lo que ocurre cuando se usa la directiva `%metatree` la construcción del árbol solicitado mediante la directiva `%tree` implícitamente considera token sintácticos aquellos terminales que aparecen en definidos en el programa `eyapp` mediante el uso de apóstrofes. Los token sintácticos no forman parte del árbol construido. Asi pues -en el ejemplo que nos ocupa - los terminales `'='`, `'-'`, `'+'`, `'*'` y `'/'` serán -por defecto - eliminados del árbol sintáctico. Por ejemplo, para la entrada `a=b*32` el siguiente árbol es construido:

```
nereida:~/src/perl/YappWithDefaultAction/examples> useruleandshift.pl
a=b*32
$VAR1 = bless( { 'children' => [
    bless( { 'children' => [], 'attr' => 'a', 'token' => 'VAR' }, 'TERMINAL' ),
    bless( { 'children' => [
        bless( { 'children' => [
            bless( { 'children' => [], 'attr' => 'b', 'token' => 'VAR' }, 'TERMINAL' )
          ]
        }, 'VAR' ),
        bless( { 'children' => [
            bless( { 'children' => [], 'attr' => '32', 'token' => 'NUM' }, 'TERMINAL' )
          ]
        }, 'NUM' )
      ]
    }, 'TIMES' )
  ]
}, 'ASSIGN' );
```

Esta conducta puede cambiarse usando la directiva `%semantic token` la cual declara una lista de terminales como semánticos. En tal caso dichos terminales formarán parte del árbol construido. Si por el contrario lo que queremos es cambiar el estatus de un terminal - por ejemplo `NUM` o `ID` - a sintáctico usaremos la directiva `%syntactic token`.

Salvando la Información de los `%syntactic token`

Si bien la funcionalidad de un `%syntactic token` es determinar la forma del AST, en ocasiones la diferencia entre terminal sintáctico y semántico es difusa. Un terminal fundamentalmente sintáctico pueden acarrear alguna información útil para las restantes fases de procesado.

Por ejemplo, el atributo número de línea asociado con el terminal sintáctico `'+'` puede ser útil posteriormente: Si en la fase de comprobación de tipos se observa un error de tipos incompatibles con el operador, disponer de la línea asociada con `'+'` nos permitirá emitir mensajes de error mas precisos.

En `Parse::Eyapp` el programador puede proveer a la clase `TERMINAL` de un método

`TERMINAL::save_attributes`

el cuál será ejecutado durante la construcción del AST cada vez que un `syntaxtic token` es eliminado. El método recibe como argumento - además de la referencia al nodo terminal - una referencia al nodo padre del terminal. Sigue un ejemplo:

```
sub TERMINAL::save_attributes {
  # $_[0] is a syntactic terminal
  # $_[1] is the father.
  push @{$_[1]->{lines}}, $_[0]->[1]; # save the line!
}
```

La tabla 3.1 muestra como el nodo `PLUS` recoge la información del número de línea del terminal `+`.

El árbol fué generado usando el método `str`. La información sobre los atributos se hizo mediante la siguiente subrutina:

```
sub generic_info {
  my $info;

  $info = "";
  $info .= $_[0]->type_info if $_[0]->can(’type_info’);

  my $sep = $info?":":"";

  $info .= $sep.$_[0]->{line} if (defined $_[0]->{line});
  local $" = ’,’;
  $info .= "$sep@{$_[0]->{lines}}" if (defined $_[0]->{lines});
  return $info;
}

*PLUS::info = =\&generic_info;
```

Cuadro 3.1: El nodo `PLUS` recoge el número de línea del terminal`+`

Programa y Footnotes	AST generado por `$t->str`

Programa y Footnotes:

```
int f(int a, int b) {
 return a+b;
}
-------footnotes-----------
0)
Symbol Table:
$VAR1 = {
  ’f’ => {
    ’type’ => ’F(X_2(INT,INT),INT)’,
    ’line’ => 1
  }
};
---------------------------
1) etc.
```

AST generado por `$t->str`:

```
PROGRAM^{0}(
  FUNCTION[f]^{1}(
    RETURN[2,2](
      PLUS[INT:2](
        VAR[INT](
          TERMINAL[a:2]
        ),
        VAR[INT](
          TERMINAL[b:2]
        )
      ) # PLUS
    ) # RETURN
  ) # FUNCTION
) # PROGRAM
```

3.11. Práctica: Análisis Sintáctico

Realize durante la hora de prácticas un analizador sintáctico para el lenguaje especificado por el profesor.

3.12. Podando el Arbol

Acciones Explícitas Bajo la Directiva % tree

Como se ha mencionado anteriormente, la directiva `%tree` es es equivalente a escribir:

```
%default action { goto &Parse::Eyapp::Driver::YYBuildAST }
```

Donde el método `Parse::Eyapp::Driver::YYBuildAST` se encarga de retornar la referencia al objeto nodo recién construído.

La ubicación de acciones semánticas explícitas para una regla altera la forma del árbol de análisis. *El método* `Parse::Eyapp::Driver::YYBuildAST` *no inserta en la lista de hijos de un nodo el atributo asociado con dicho símbolo a menos que este sea una referencia. Esto conlleva que una forma de eliminar un subárbol es estableciendo una acción semántica explícita que no retorne un nodo.*

Las Declaraciones no se Incluyen en el Arbol de Análisis

El siguiente ejemplo ilustra como utilizar esta propiedad en el análisis de un lenguaje mínimo en el que los programas (`P`) son secuencias de declaraciones (`DS`) seguidas de sentencias (`SS`).

En un compilador típico, las declaraciones influyen en el análisis semántico (por ejemplo, enviando mensajes de error si el objeto ha sido repetidamente declarado o si es usado sin haber sido previamente declarado, etc.) y la información proveída por las mismas suele guardarse en una tabla de símbolos (declarada en la línea 5 en el siguiente código) para su utilización durante las fases posteriores.

```
pl@nereida:~/LEyapp/examples$ sed -ne ’1,58p’ RetUndef.eyp | cat -n
 1  /* File RetUndef.eyp */
 2
 3  %tree
 4  %{
 5    use Data::Dumper;
 6    my %s; # symbol table
 7  %}
 8
 9  %%
10  P:
11    %name PROG
12      $DS $SS
13        {
14          print Dumper($DS);
15          $SS->{symboltable} = \%s;
16          return $SS;
17        }
18  ;
19
20  SS:
21     %name ST
22       S
23   | %name STS
24       SS S
25  ;
26
27  DS:
28     %name DEC
29       D
30         {}
31   | %name DECS
```

```
32       DS  D
33         {}
34  ;
35
36  D : %name INTDEC
37      ’int’ $ID
38         {
39           die "Error: $ID declared twice\n" if exists($s{$ID});
40           $s{$ID} = ’int’;
41         }
42    | %name STRDEC
43      ’string’ $ID
44        {
45          die "Error: $ID declared twice\n" if exists($s{$ID});
46          $s{$ID} = ’string’;
47        }
48  ;
49
50  S: %name EXP
51      $ID
52        {
53          die "Error: $ID not declared\n" unless exists($s{$ID});
54          goto &Parse::Eyapp::Driver::YYBuildAST; # build the node
55        }
56  ;
57
58  %%
```

La Tabla de Símbolos

Las acciones explícitas en las líneas 37-41 y 44-47 no retornan referencias y eso significa que el subárbol de `D` no formará parte del árbol de análisis. En vez de ello las acciones tienen por efecto almacenar el tipo en la entrada de la tabla de símbolos asociada con la variable.

Dando Nombre a los Atributos

Cuando en la parte derecha de una regla de producción un símbolo va precedido de un dolar como ocurre en el ejemplo con `ID`:

```
36  D : %name INTDEC
37      ’int’ $ID
38         {
39           die "Error: $ID declared twice\n" if exists($s{$ID});
40           $s{$ID} = ’int’;
41         }
```

le estamos indicamos a eyapp que construya una copia en una variable léxica `$ID` del atributo asociado con ese símbolo. Así el fragmento anterior es equivalente a escribir:

```
  D : %name INTDEC
      ’int’ ID
         {
           my $ID = $_[2];
           die "Error: $ID declared twice\n" if exists($s{$ID});
           $s{$ID} = ’int’;
         }
```

Nótese que la tabla de símbolos es una variable léxica. Para evitar su pérdida cuando termine la fase de análisis sintáctico se guarda como un atributo del nodo programa:

```
P:
  %name PROG
    $DS $SS
      {
        print Dumper($DS);
        $SS->{symboltable} = \%s;
        return $SS;
      }
;
```

Si no se desea que el nombre del atributo coincida con el nombre de la variable se utiliza la notación *punto*. Por ejemplo:

```
exp : exp.left '+' exp.right { $left + $right }
;
```

es equivalente a:

```
exp : exp.left '+' exp.right
       {
         my $left  = $_[1];
         my $right = $_[2];
         $left + $right
       }
;
```

Insertando Tareas Adicionales a la Construcción de un Nodo

Si queremos ejecutar ciertas tareas antes de la construcción de un nodo tendremos que insertar una acción semántica para las mismas. Para evitar la pérdida del subárbol deberemos llamar explícitamente a `Parse::Eyapp::Driver::YYBuildAST`. Este es el caso cuando se visitan los nodos sentencia (`S`):

```
S: %name EXP
    $ID
      {
        die "Error: $ID not declared\n" unless exists($s{$ID});
        goto &Parse::Eyapp::Driver::YYBuildAST; # build the node
      }
;
```

El Programa Cliente y su Ejecución

Veamos el programa cliente:

```
pl@nereida:~/LEyapp/examples$ cat -n useretundef.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use Parse::Eyapp;
 4  use RetUndef;
 5  use Data::Dumper;
 6
 7  sub TERMINAL::info { $_[0]{attr} }
 8
 9  my $parser = RetUndef->new();
```

```
10  my $t = $parser->Run;
11  print $t->str,"\n";
12
13  $Data::Dumper::Indent = 1;
14  print Dumper($t);
```

Observe como no hay nodos de tipo `D` al ejecutar el programa con entrada `int a string b a b`:

```
pl@nereida:~/LEyapp/examples$ useretundef.pl
int a string b a b
$VAR1 = undef;
STS(ST(EXP(TERMINAL[a])),EXP(TERMINAL[b]))
```

El volcado de `Dumper` en la línea 14 de `RetUndef.eyp` nos muestra que el atributo asociado con `DS` es `undef`.

El volcado de `Dumper` en la línea 14 de `useretundef.pl` da el siguiente resultado:

```
$VAR1 = bless( {
  ’symboltable’ => {
    ’a’ => ’int’,
    ’b’ => ’string’
  },
  ’children’ => [
    bless( {
      ’children’ => [
        bless( {
          ’children’ => [
            bless( {
              ’children’ => [],
              ’attr’ => ’a’,
              ’token’ => ’ID’
            }, ’TERMINAL’ )
          ]
        }, ’EXP’ )
      ]
    }, ’ST’ ),
    bless( {
      ’children’ => [
        bless( {
          ’children’ => [],
          ’attr’ => ’b’,
          ’token’ => ’ID’
        }, ’TERMINAL’ )
      ]
    }, ’EXP’ )
  ]
}, ’STS’ );
```

3.13. Nombres para los Atributos

Desventajas de la Notación Posicional

Dentro de las acciones los atributos de la parte derecha de la regla de producción $A \rightarrow X_1 \ldots X_n$ se pasan como parámetros en `$_[1]`, `$_[2]`, etc. La notación posicional para los atributos puede ser inconveniente si el número de símbolos es grande. También puede resultar inconveniente durante la fase de desarrollo: Puede ocurrir que hemos escrito la regla $A \to X_1X_2X_3$ y la acción semántica (por ejemplo `{ $f = $_[1]*$_[3]+$_[2]; $_[2] }` y - en determinado momento - nos damos cuenta que debemos cambiar la regla añadiendo un nuevo símbolo $A \to X_1YX_2X_3$ o suprimiendo alguno que ya existía. O quizá queremos insertar una acción en algún lugar intermedio de la regla. O quizá reordenamos los símbolos en la parte derecha. En todos estos caso nos encontramos en la situación de que debemos reenumerar todos los argumentos dentro de la acción semántica. En el ejemplo anterior tendríamos que reescribir la acción como:

```
{ $f = $_[1]*$_[4]+$_[3]; $_[3] }
```

La notación Dolar

Para tratar con esta situación eyapp provee mecanismos para darle nombres a los atributos semánticos asociados con los símbolos. Uno de esos mecanismos es prefijar el símbolo con un dolar. Ello indica que el nombre del símbolo puede ser usado dentro de la acción como nombre del atributo asociado. Por ejemplo, el código en la línea 22 imprime el atributo asociado con la variable sintáctica `expr`, que en este caso es su valor numérico.

La Notación Punto

La otra notación usada para darle nombres a los atributos consiste en concatenar el símbolo en la parte derecha de la regla de un punto seguido del nombre del atributo (el código completo figura en el apéndice en la página **??**):

```
26  exp:
27      NUM
28    | $VAR                    { $s{$VAR} }
29    | VAR.x ’=’ exp.y         { $s{$x} = $y }
30    | exp.x ’+’ exp.y         { $x + $y }
31    | exp.x ’-’ exp.y         { $x - $y }
32    | exp.x ’*’ exp.y         { $x * $y }
33    | exp.x ’^’ exp.y         { $x ** $y }
34    | exp.x ’/’ exp.y
35      {
36        my $parser = shift;
37
38             $y and return($x / $y);
39         $parser->YYData->{ERRMSG}
40           =   "Illegal division by zero.\n";
41         $parser->YYError;
42         undef
43      }
44    |   ’-’ $exp %prec NEG
45      { -$exp }
46    |   ’(’ $exp ’)’
47      { $exp }
48  ;
```

La Cabecera y el Arranque

El atributo asociado con `start` (línea 12) es una referencia a un par. El segundo componente del par es una referencia a la tabla de símbolos. El primero es una referencia a la lista de valores resultantes de la evaluación de las diferentes expresiones.

```
pl@nereida:~/LEyapp/examples$ cat -n CalcSimple.eyp
 1  # CalcSimple.eyp
 2  %right  ’=’
```

```
 3  %left   '-' '+'
 4  %left   '*' '/'
 5  %left   NEG
 6  %right  '^'
 7  %{
 8  my %s;
 9  %}
10
11  %%
12  start: input { [ $_[1], \%s] }
13  ;
14
15  input:
16      /* empty */     { [] }
17    | input line  { push(@{$_[1]},$_[2]) if defined($_[2]); $_[1] }
18  ;
```

La línea 17 indica que el atributo asociado con la variable sintáctica `input` es una referencia a una pila y que el atributo asociado con la variable sintáctica `line` debe empujarse en la pila.

De hecho, el atributo asociado con `line` es el valor de la expresión evaluada en esa línea. Asi pues el atributo retornado por `input` es una referencia a una lista conteniendo los valores de las expresiones evaluadas. Observe que expresiones erróneas no aparecerán en la lista.

```
20  line:
21    '\n'         { undef }
22    | $exp '\n'  { print "$exp\n"; $exp }
23    | error '\n' { $_[0]->YYErrok; undef }
24  ;
```

La Cola

Veamos el código del analizador léxico:

```
50  %%
51
52  sub _Error {
..   ............................
62  }
63
64  my $input;
65
66  sub _Lexer {
67    my($parser)=shift;
68
69    # topicalize $input
70    for ($input) {
71      s/^[ \t]//;      # skip whites
72      return('',undef) unless $_;
73      return('NUM',$1) if s{^([0-9]+(?:\.[0-9]+)?)}{};
74      return('VAR',$1) if s/^([A-Za-z][A-Za-z0-9_]*)//;
75      return($1,$1)    if s/^(.)//s;
76    }
77
78    return('',undef);
79  }
```

```

sub Run {
    my($self)=shift;

    $input = shift;
    return $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
                           #yydebug => 0xF
                         );
}
```

Recuperación de Errores

Las entradas pueden contener errores. El lenguaje `eyapp` proporciona un *token* especial, `error`, que puede ser utilizado en el programa fuente para extender el analizador con producciones de error que lo dotan de cierta capacidad para recuperase de una entrada errónea y poder continuar analizando el resto de la entrada.

```
20 line:
21   '\n'         { undef }
22   | $exp '\n'  { print "$exp\n"; $exp }
23   | error '\n' { $_[0]->YYErrok; undef }
24 ;
```

Cuando se produce un error en el análisis, `eyapp` emite un mensaje de error (Algo como `"syntax error"`) y produce "mágicamente" el terminal especial denominado `error`. A partir de ahí permanecerá silencioso, consumiendo terminales hasta encontrar una regla que sea capaz de consumir el terminal `error`. La idea general es que, a traves de la regla de recuperación de errores de la línea 23 se indica que cuando se produzca un error el analizador debe descartar todos los *tokens* hasta llegar a un retorno de carro.

Además, mediante la llamada al método `$_[0]->YYErrok` el programador anuncia que, si se alcanza este punto, la recuperación puede considerarse "completa" y que `eyapp` puede emitir a partir de ese momento mensajes de error con la seguridad de que no son consecuencia de un comportamiento inestable provocado por el primer error.

La Acción por Defecto

Observemos la regla:

```
26 exp:
27     NUM
```

La acción por defecto es retornar `$_[1]`. Por tanto, en el caso de la regla de la línea 27 el valor retornado es el asociado a `NUM`.

Manejo de los símbolos

La calculadora usa un hash léxico `%s` como tabla de símbolos. Cuando dentro de una expresión encontramos una alusión a una variable retornamos el valor asociado `$s{$_[1]}` que ha sido guardado en la correspondiente entrada de la tabla de símbolos:

```
...
%{
my %s;
%}
...
%%
...
```

```
exp:
    NUM
  | $VAR                    { $s{$VAR} }
  | VAR.x ’=’ exp.y         { $s{$x} = $y }
  | exp.x ’+’ exp.y         { $x + $y }
...
```

El Atributo `YYData`

En la regla de la división comprobamos que el divisor es distinto de cero.

```
  | exp.x ’/’ exp.y
    {
      my $parser = shift;

           $y and return($x / $y);
       $parser->YYData->{ERRMSG}
         =   "Illegal division by zero.\n";
       $parser->YYError;
       undef
    }
```

El método `YYData` provee acceso a un hash que se maneja como una zona de datos para el programa cliente. En el ejemplo usamos una entrada `ERRMSG` para alojar el mensaje de error.

Este mensaje es aprovechado por la subrutina de tratamiento de errores:

```
sub _Error {
  my $private = $_[0]->YYData;

      exists $private->{ERRMSG}
  and do {
      print $private->{ERRMSG};
      delete $private->{ERRMSG};
      return;
  };
  print "Syntax error.\n";
}
```

La subrutina `_Error` es llamada por el analizador sintáctico generado por `eyapp` cada vez que ocurre un error sintáctico. Para que ello sea posible en la llamada al analizador se especifican quienes son la rutina de análisis léxico y quien es la rutina a llamar en caso de error:

```
$self->YYParse( yylex => \&_Lexer, yyerror => \&_Error );
```

El Programa Cliente

Veamos los contenidos del ejecutable `usecalcsimple.pl` el cuál utiliza el módulo generado por `eyapp`:

```
pl@nereida:~/LEyapp/examples$ cat -n usecalcsimple.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use CalcSimple;
 4  use Carp;
 5
 6  sub slurp_file {
 7    my $fn = shift;
```

```
 8     my $f;
 9
10     local $/ = undef;
11     if (defined($fn)) {
12       open $f, $fn
13     }
14     else {
15       $f = \*STDIN;
16     }
17     my $input = <$f>;
18     return $input;
19   }
20
21   my $parser = CalcSimple->new();
22
23   my $input = slurp_file( shift() );
24   my ($r, $s) = @{$parser->Run($input)};
25
26   print "========= Results  ==============\n";
27   print "$_\n" for @$r;
28   print "========= Symbol Table ==============\n";
29   print "$_ = $s->{$_}\n" for sort keys %$s;
```

Ejecución

Veamos una ejecución:

```
pl@nereida:~/LEyapp/examples$ cat prueba.exp
a=2*3
b=a+1
pl@nereida:~/LEyapp/examples$ usecalcsimple.pl prueba.exp
6
7
========= Results  ==============
6
7
========= Symbol Table ==============
a = 6
b = 7
```

La primera aparición de la cadena `"6\n7"` es debida a la acción `print` en el cuerpo de la gramática. La segunda es el resultado de la línea 27 en el programa cliente.

3.14. Bypass Automático

La directiva `%tree` admite la opción `bypass`. Esta opción ayuda en la fase de construcción del AST automatizando *operaciones de bypass*. Entendemos por *operación de bypass* a la operación de retornar el nodo hijo al nodo padre del actual (abuelo del hijo), obviando así la presencia del nodo actual en el AST final.

La opción de `bypass` modifica la forma en la que Eyapp construye el árbol: Si se usa la opción bypass, *Eyapp hará automáticamente una operación de bypass a todo nodo que resulte con un sólo hijo en tiempo de construcción el árbol*. Además *Eyapp cambiará la clase del nodo hijo a la clase del nodo siendo puenteado si a este se le dió un nombre explícitamente* a través de la directiva `%name`.

Hay dos razones principales por las que un nodo pueda resultar - en tiempo de construcción del árbol - con un sólo hijo:

- La mas obvia es que el nodo tuviera ya un sólo hijo por la forma de la gramática: el nodo en cuestión se corresponde con la parte izquierda de una *regla de producción unaria* de la forma $E \rightarrow NUM$ o $E \rightarrow VAR$.

  ```
  exp:
      %name NUM
      NUM
    | %name VAR
      VAR
  ```

  En este caso un árbol de la forma `PLUS(NUM(TERMINAL[4]), VAR(TERMINAL[a])` se verá transformado en `PLUS(NUM[4], VAR[a])`. El primer nodo `TERMINAL` será rebautizado (re-bendecido) en la clase `NUM`. y el segundo en la clase `VAR`.

  Observe que este proceso de renombre ocurre sólo si el nodo eliminado tiene un nombre explícito dado con la directiva `%name`. Así en la regla unaria:

  ```
  pl@nereida:~/LEyapp/examples$ sed -ne '/^line:/,/^;$/p' TreeBypass.eyp | cat -n
       1  line:
       2      exp '\n'
       3  ;
  ```

  ocurre un bypass automático pero el nodo hijo de `exp` conserva su nombre. Por ejemplo, un árbol como `exp_10(PLUS(NUM(TERMINAL),NUM(TERMINAL)))` se transforma en `PLUS(NUM,NUM)`. Mientras que los nodos `TERMINAL` fueron renombrados como nodos `NUM` el nodo `PLUS` no lo fué.

- La otra razón para que ocurra un bypass automático es que - aunque la parte derecha de la regla de producción tuviera mas de un símbolo - sus restantes hijos sean podados por ser terminales sintácticos. Esto ocurre, por ejemplo, con los paréntesis en la regla $E \rightarrow (E)$.

  Por ejemplo, un árbol como:

  ```
  PAREN(PLUS(NUM( TERMINAL[2]),NUM(TERMINAL[3])))
  ```

  se convierte en:

  ```
  PLUS(NUM[2],NUM[3])
  ```

Un Ejemplo con `bypass` Automático

```
pl@nereida:~/LEyapp/examples$ cat -n TreeBypass.eyp
 1  # TreeBypass.eyp
 2  %right  '='
 3  %left   '-' '+'
 4  %left   '*' '/'
 5  %left   NEG
 6  %right  '^'
 7
 8  %tree bypass
 9
10  %{
11  sub NUM::info {
12    my $self = shift;
13
14    $self->{attr};
```

```
15  }
16
17  *VAR::info = \&NUM::info;
18  %}
19  %%
```

Dado que los nodos `TERMINAL` serán rebautizados como nodos `VAR` y `NUM`, dotamos a dichos nodos de métodos `info` (líneas 11-17) que serán usados durante la impresión del árbol con el método `str`.

```
21  line:
22      exp ’\n’
23  ;
24
25  exp:
26      %name NUM
27      NUM
28    | %name VAR
29      VAR
30    | %name ASSIGN
31      var ’=’ exp
32    | %name PLUS
33      exp ’+’ exp
34    | %name MINUS
35      exp ’-’ exp
36    | %name TIMES
37      exp ’*’ exp
38    | %name DIV
39      exp ’/’ exp
40    | %name UMINUS
41      ’-’ exp %prec NEG
42    | %name EXPON
43      exp ’^’ exp
44    | ’(’ exp ’)’
45  ;
46
47  var:
48      %name VAR
49         VAR
50  ;
51  %%
```

Observe las líneas 30-31 y 47-49. La gramática original tenía sólo una regla para la asignación:

```
exp:
    %name NUM
    NUM
  | %name VAR
    VAR
  | %name ASSIGN
    VAR ’=’ exp
```

Si se hubiera dejado en esta forma un árbol como `ASSIGN(TERMINAL[a], NUM(TERMINAL[4]))` quedaría como `ASSIGN(TERMINAL[a], NUM[4])`. La introdución de la variable sintáctica `var` en la regla de asignación y de su regla unaria (líneas 47-50) produce un bypass y da lugar al árbol `ASSIGN(VAR[a], NUM[4])`.

Rutinas de Soporte

```
53  sub _Error {
54          exists $_[0]->YYData->{ERRMSG}
55      and do {
56          print $_[0]->YYData->{ERRMSG};
57          delete $_[0]->YYData->{ERRMSG};
58          return;
59      };
60      print "Syntax error.\n";
61  }
62
63  my $input;
64
65  sub _Lexer {
66      my($parser)=shift;
67
68      # topicalize $input
69      for ($input) {
70        s/ˆ[ \t]+//;      # skip whites
71        return(’’,undef) unless $_;
72        return(’NUM’,$1) if s{ˆ([0-9]+(?:\.[0-9]+)?)}{};
73        return(’VAR’,$1) if s/ˆ([A-Za-z][A-Za-z0-9_]*)//;
74        return($1,$1)    if s/ˆ(.)//s;
75      }
76      return(’’,undef);
77  }
78
79  sub Run {
80      my($self)=shift;
81
82      $input = shift;
83      print $input;
84      return $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
85        #yydebug => 0xF
86      );
87  }
```

El Programa Cliente

```
pl@nereida:~/LEyapp/examples$ cat -n usetreebypass.pl
 1  #!/usr/bin/perl -w
 2  # usetreebypass.pl prueba2.exp
 3  use strict;
 4  use TreeBypass;
 5
 6  sub slurp_file {
 7    my $fn = shift;
 8    my $f;
 9
10    local $/ = undef;
11    if (defined($fn)) {
12      open $f, $fn  or die "Can’t find file $fn!\n";
13    }
```

```
  else {
    $f = \*STDIN;
  }
  my $input = <$f>;
  return $input;
}

my $parser = TreeBypass->new();

my $input = slurp_file( shift() );
my $tree = $parser->Run($input);
die "There were errors\n" unless defined($tree);

$Parse::Eyapp::Node::INDENT = 2;
print $tree->str."\n";
```

Ejecuciones Veamos las salidas obtenidas sobre diversas entradas:

```
pl@nereida:~/LEyapp/examples$ eyapp TreeBypass.eyp
pl@nereida:~/LEyapp/examples$ usetreebypass.pl prueba2.exp
a=(2+b)*3

ASSIGN(
  VAR[a],
  TIMES(
    PLUS(
      NUM[2],
      VAR[b]
    ),
    NUM[3]
  ) # TIMES
) # ASSIGN
pl@nereida:~/LEyapp/examples$ usetreebypass.pl prueba3.exp
a=2-b*3

ASSIGN(
  VAR[a],
  MINUS(
    NUM[2],
    TIMES(
      VAR[b],
      NUM[3]
    )
  ) # MINUS
) # ASSIGN
pl@nereida:~/LEyapp/examples$ usetreebypass.pl prueba4.exp
4*3

TIMES(
  NUM[4],
  NUM[3]
)
pl@nereida:~/LEyapp/examples$ usetreebypass.pl prueba5.exp
2-)3*4
```

```
Syntax error.
There were errors
```

La Directiva `no bypass` La cláusula `bypass` suele producir un buen número de podas y reorganizaciones del árbol.

Es preciso tener especial cuidado en su uso. De hecho, el programa anterior contiene errores. Obsérvese la conducta del analizador para la entrada `-(2-3)`:

```
pl@nereida:~/LEyapp/examples$ usetreebypass.pl prueba7.exp
-(2-3)

UMINUS(
  NUM[2],
  NUM[3]
)
```

Que es una salida errónea: además de los bypasses en los terminales se ha producido un bypass adicional sobre el nodo `MINUS` en el árbol original

```
UMINUS(MINUS(NUM(TERMINAL[2],NUM(TERMINAL[3]))))
```

dando lugar al árbol:

```
UMINUS(NUM[2],NUM[3])
```

El bypass automático se produce en la regla del menos unario ya que el terminal `'-'` es por defecto - un terminal sintáctico:

```
25  exp:
..        ...........
40      | %name UMINUS
41        '-' exp %prec NEG
```

Mediante la aplicación de la directiva `%no bypass UMINUS` a la regla (línea 16 abajo) inhibimos la aplicación del bypass a la misma:

```
pl@nereida:~/LEyapp/examples$ sed -ne '/^exp:/,/^;$/p' TreeBypassNoBypass.eyp | cat -n
     1  exp:
     2      %name NUM
     3      NUM
     4    | %name VAR
     5      VAR
     6    | %name ASSIGN
     7      var '=' exp
     8    | %name PLUS
     9      exp '+' exp
    10    | %name MINUS
    11      exp '-' exp
    12    | %name TIMES
    13      exp '*' exp
    14    | %name DIV
    15      exp '/' exp
    16    | %no bypass UMINUS
    17      '-' exp %prec NEG
    18    | %name EXPON
    19      exp '^' exp
    20    | '(' exp ')'
    21  ;
```

```
pl@nereida:~/LEyapp/examples$ usetreebypassnobypass.pl prueba7.exp
-(2-3)

UMINUS(
  MINUS(
    NUM[2],
    NUM[3]
  )
) # UMINUS
```

El Método `YYBypassrule`

Aún mas potente que la directiva es usar el método `YYBypassrule` el cual permite modificar dinámicamente el estatus de bypass de una regla de producción. Vea esta nueva versión del anterior ejemplo:

```
pl@nereida:~/LEyapp/examples$ sed -ne '/%%/,/%%/p' TreeBypassDynamic.eyp | cat -n
 1  %%
 2
 3  line:
 4      exp '\n'
 5  ;
 6
 7  exp:
 8      %name NUM
 9      NUM
10    | %name VAR
11      VAR
12    | %name ASSIGN
13      var '=' exp
14    | %name PLUS
15      exp '+' exp
16    | %name MINUS
17      exp '-' exp
18    | %name TIMES
19      exp '*' exp
20    | %name DIV
21      exp '/' exp
22    | %name UMINUS
23      '-' exp %prec NEG
24        {
25          $_[0]->YYBypassrule(0);
26          goto &Parse::Eyapp::Driver::YYBuildAST;
27        }
28    | %name EXPON
29      exp '^' exp
30    | '(' exp ')'
31  ;
32
33  var:
34      %name VAR
35         VAR
36  ;
37  %%
```

El analizador produce un árbol sintáctico correcto cuando aparecen menos unarios:

```
pl@nereida:~/LEyapp/examples$ usetreebypassdynamic.pl
-(2--3*5)
-(2--3*5)

UMINUS(
  MINUS(
    NUM[2],
    TIMES(
      UMINUS(
        NUM[3]
      ),
      NUM[5]
    ) # TIMES
  ) # MINUS
) # UMINUS
```

3.15. La opción `alias` de `%tree`

La opción `alias` de la directiva `%tree` da lugar a la construcción de métodos de acceso a los hijos de los nodos para los cuales se haya explicitado un nombre a través de la *notación punto* o de la *notación dolar*.

Veamos un ejemplo:

```
nereida:~/src/perl/YappWithDefaultAction/examples> cat -n ./alias.pl
  1  #!/usr/bin/perl -w
  2  use strict;
  3  use Parse::Eyapp;
  4
  5  my $grammar = q{
  6  %right  '='
  7  %left   '-' '+'
  8  %left   '*' '/'
  9  %left   NEG
!10  %tree bypass alias
 11
 12  %%
 13  line: $exp  { $_[1] }
 14  ;
 15
 16  exp:
 17      %name NUM
 18            $NUM
 19          | %name VAR
 20            $VAR
 21          | %name ASSIGN
 22            $VAR '=' $exp
 23          | %name PLUS
 24            exp.left '+' exp.right
 25          | %name MINUS
 26            exp.left '-' exp.right
 27          | %name TIMES
 28            exp.left '*' exp.right
 29          | %name DIV
 30            exp.left '/' exp.right
```

```
        | %no bypass UMINUS
          '-' $exp %prec NEG
  |   '(' exp ')'  { $_[2] } /* Let us simplify a bit the tree */
;

%%

sub _Error {
        exists $_[0]->YYData->{ERRMSG}
    and do {
        print $_[0]->YYData->{ERRMSG};
        delete $_[0]->YYData->{ERRMSG};
        return;
    };
    print "Syntax error.\n";
}

sub _Lexer {
    my($parser)=shift;

        $parser->YYData->{INPUT}
    or  $parser->YYData->{INPUT} = <STDIN>
    or  return('',undef);

    $parser->YYData->{INPUT}=~s/^\s+//;

    for ($parser->YYData->{INPUT}) {
        s/^([0-9]+(?:\.[0-9]+)?)//
                and return('NUM',$1);
        s/^([A-Za-z][A-Za-z0-9_]*)//
                and return('VAR',$1);
        s/^(.)//s
                and return($1,$1);
    }
}

sub Run {
    my($self)=shift;
    $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
                    #yydebug =>0xFF
                  );
}
}; # end grammar


Parse::Eyapp->new_grammar(
  input=>$grammar,
  classname=>'Alias',
  firstline =>7,
);
my $parser = Alias->new();
$parser->YYData->{INPUT} = "a = -(2*3+5-1)\n";
my $t = $parser->Run;
```

```
$Parse::Eyapp::Node::INDENT=0;
print $t->VAR->str."\n";             # a
print "***************\n";
print $t->exp->exp->left->str."\n";  # 2*3+5
print "***************\n";
print $t->exp->exp->right->str."\n"; # 1
```

Este programa produce la salida:

```
nereida:˜/src/perl/YappWithDefaultAction/examples> ./alias.pl
TERMINAL
***************
PLUS(TIMES(NUM,NUM),NUM)
***************
NUM
```

El efecto de la opción `alias` en una regla como:

```
           | %name TIMES
             exp.left ’*’ exp.right
```

es crear dos métodos `left` y `right` en la clase `TIMES`. El método `left` permite acceder al hijo izquierdo del nodo en el AST y el método `right` al hijo derecho.

El efecto de la opción `alias` en una regla como:

```
           | %name ASSIGN
             $VAR ’=’ $exp
```

es crear dos métodos en la clase `ASSIGN` denominados `VAR` y `exp` los cuales permiten acceder a los correspondientes hijos de uno nodo `ASSIGN`.

Los métodos son construidos durante la compilación de la gramática. Por tanto, las modificaciones al AST que el programa introduzca en tiempo de ejecución (por ejemplo la supresión de hijos de ciertos nodos) invalidarán el uso de estos métodos.

3.16. Diseño de Analizadores con `Parse::Eyapp`

A la hora de construir un analizador sintáctico tenga en cuenta las siguientes normas de buena programación:

1. Comienze trabajando en el cuerpo de la gramática.

2. Olvídese al principio del analizador léxico. Su primer objetivo es tener una gramática limpia de conflictos y que reconozca el lenguaje dado.

3. Sustituya las repeticiones BNF por listas usando los operadores `eyapp` `+`, `*` y sus variantes con separadores. Si una variable describe una lista de `cosas` pongale un adjetivo adecuado como `cosaslist`. Ponga nombres significativos a las variables y terminales. No los llame `d1`, `d2`, etc.

4. Si tiene un elemento opcional en la BNF, por ejemplo, en la regla:

   functiondefinition → [basictype] functionheader functionbody

   use el operador `?`.

5. Cada par de reglas que introduzca vuelva a recompilar con `eyapp` la gramática para ver si se introducido ambiguedad. Cuando estoy editando la gramática suelo escribir a menudo la orden

   ```
   :!eyapp %
   ```

   para recompilar:

```

declaratorlist: declarator +
;
functiondefinition:
   basictype functionheader functionbody
 | functionheader functionbody
;

%%
~
~
~
~
~
~
~
~
~
~
:!eyapp %
```

Esto llama a `eyapp` con el fichero bajo edición. Si hay errores o conflictos (esto es, hemos introducido ambiguedad) los detectarémos enseguida. Procure detectar la aparición de un conflicto lo antes posible. Observe el sangrado del ejemplo. Es el que le recomiendo.

6. Cuando esté en el proceso de construcción de la gramática y aún le queden por rellenar variables sintácticas, declárelas como terminales mediante `%token`. De esta manera evitará las quejas de `eyapp`.

7. **Resolución de Ambiguedades y Conflictos**

   Las operaciones de asignación tienen la prioridad mas baja, seguidas de las lógicas, los test de igualdad, los de comparación, a continuación las aditivas, multiplicativas y por último las operaciones de tipo `unary` y `primary`. Exprese la asociatividad natural y la prioridad especificada usando los mecanismos que `eyapp` provee al efecto: `%left`, `%right`, `%nonassoc` y `%prec`.

8. La gramática de SimpleC es ambigua, ya que para una sentencia como

   $$\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$$

   existen dos árboles posibles: uno que asocia el "else" con el primer "if" y otra que lo asocia con el segundo. Los dos árboles corresponden a las dos posibles parentizaciones:

   $$\text{if } E_1 \text{ then (if } E_2 \text{ then } S_1 \text{ else } S_2)$$

   Esta es la regla de prioridad usada en la mayor parte de los lenguajes: un "else" casa con el "if" mas cercano. La otra posible parentización es:

   $$\text{if } E_1 \text{ then (if } E_2 \text{ then } S_1) \text{ else } S_2$$

   La conducta por defecto de `eyapp` *es parentizar a derechas.* El generador `eyapp` nos informará del conflicto pero si no se le indica como resolverlo parentizará a derechas. Resuelva este conflicto.

9. *¿Que clase de árbol debe producir el analizador?* La respuesta es que sea lo mas abstracto posible. Debe

   - Contener toda la información necesaria para el manejo eficiente de las fases subsiguientes: Análisis de ámbito, Comprobación de tipos, Optimización independiente de la máquina, etc.
   - Ser uniforme
   - Legible (human-friendly)
   - No contener nodos que no portan información.

   El siguiente ejemplo muestra una versión aceptable de árbol abstracto. Cuando se le proporciona el programa de entrada:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code> cat -n prueba5.c
     1  int f(int a)
     2  {
     3    if (a>0)
     4      a = f(a-1);
     5  }
```

   El siguiente árbol ha sido producido por un analizador usando la directiva `%tree` y añadiendo las correspondientes acciones de `bypass`. Puede considerarse un ejemplo aceptable de AST:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code> eyapp Simple2 ;\
                                  usesimple2.pl prueba5.c
PROGRAM(
  TYPEDFUNC(
    INT(TERMINAL[INT:1]),
    FUNCTION(
      TERMINAL[f:1],
      PARAMS(
        PARAM(
          INT(TERMINAL[INT:1]),
          TERMINAL[a:1],
          ARRAYSPEC
        )
      ),
      BLOCK(
        DECLARATIONS,
        STATEMENTS(
          IF(
            GT(
              VAR(TERMINAL[a:3]),
              INUM(TERMINAL[0:3])
            ),
            ASSIGN(
              VAR(TERMINAL[a:4]),
              FUNCTIONCALL(
                TERMINAL[f:4],
                ARGLIST(
                  MINUS(
                    VAR(TERMINAL[a:4]),
                    INUM(TERMINAL[1:4])
```

```
                  )
                ) # ARGLIST
              ) # FUNCTIONCALL
            ) # ASSIGN
          ) # IF
        ) # STATEMENTS
      ) # BLOCK
    ) # FUNCTION
  ) # TYPEDFUNC
) # PROGRAM
```

Es deseable darle una estructura uniforme al árbol. Por ejemplo, como consecuencia de que la gramática admite funciones con declaración implícita del tipo retornado cuando este es entero

```
definition:
    funcDef { $_[1]->type("INTFUNC"); $_[1] }
  | %name TYPEDFUNC
    basictype funcDef
  | declaration { $_[1] }
;
```

se producen dos tipos de árboles. Es conveniente convertir las definiciones de función con declaración implícita en el mismo árbol que se obtiene con declaración explícita.

3.17. Práctica: Construcción del Arbol para el Lenguaje Simple

Reescriba la gramática del lenguaje Simple introducido en la práctica 3.6 para producir un árbol abstracto fácil de manejar. La gramática original produce árboles excesivamente profundos y profusos. Reescriba la gramática usando precedencia de operadores para deshacer las ambiguedades. Optimice la forma de los árboles usando bypass donde sea conveniente.

Las declaraciones no formaran parte del árbol. Utilice una tabla de símbolos para guardar la información aportada en las declaraciones sobre los identificadores. Compruebe que ninguna variable es declarada dos veces y que no se usan variables sin declarar.

3.18. Práctica: Ampliación del Lenguaje Simple

Amplíe la versión del lenguaje Simple desarrollada en la práctica 3.17 para que incluya sentencias `if`, `if ... else ...`, bucles `while` y `do ... while`.

Añada el tipo array. Deberían aceptarse declaraciones como esta:

```
[10][20]int a,b;
[5]string c;
```

y sentencias como esta:

```
a[4][7] =  b[2][1]*2;
c[1] = "hola";
```

Diseñe una representación adecuada para almacenar los tipos array en la tabla de símbolos.

3.19. Agrupamiento y Operadores de Listas

Los operadores de listas `*`, `+` y `?` pueden usarse en las partes derechas de las reglas de producción para indicar repeticiones[1].

---

[1] La descripción que se hace en esta versión requiere una versión de `eyapp` igual o posterior a 1.087

El Operador Cierre Positivo

La gramática:

```
pl@nereida:~/LEyapp/examples$ head -12 List3.yp | cat -n
 1  # List3.yp
 2  %semantic token 'c'
 3  %{
 4  use Data::Dumper;
 5  %}
 6  %%
 7  S:      'c'+  'd'+
 8              {
 9                 print Dumper($_[1]);
10                 print Dumper($_[2]);
11              }
12  ;
```

Es equivalente a:

```
pl@nereida:~/LEyapp/examples$ eyapp -v List3.yp | head -9 List3.output
Rules:
------
0:      $start -> S $end
1:      PLUS-1 -> PLUS-1 'c'
2:      PLUS-1 -> 'c'
3:      PLUS-2 -> PLUS-2 'd'
4:      PLUS-2 -> 'd'
5:      S -> PLUS-1 PLUS-2
```

La acción semántica asociada con un operador de lista `+` retorna una lista con los atributos de los factores a los que afecta el `+`:

```
pl@nereida:~/LEyapp/examples$ use_list3.pl
ccdd
$VAR1 = [ 'c', 'c' ];
$VAR1 = [ 'd', 'd' ];
```

Si se activado una de las directivas de creación de árbol (`%tree` o `%metatree`) o bien se ha hecho uso explícito del método `YYBuildingTree` o se pasa la opción `yybuildingtree` de `YYParse` la semántica de la acción asociada cambia. En tal caso la acción semántica asociada con un operador de lista `+` es crear un nodo con la etiqueta

`_PLUS_LIST_#number`

cuyos hijos son los elementos de la lista. El número es el número de orden de la regla tal y como aparece en el fichero `.output`. Como ocurre cuando se trabaja bajo la directiva `%tree`, es necesario que los atributos de los símbolos sean terminales semánticos o referencias para que se incluyan en la lista.

Al ejecutar el programa anterior bajo la directiva `%tree` se obtiene la salida:

```
pl@nereida:~/LEyapp/examples$ head -3 List3.yp; eyapp List3.yp
# List3.yp
%semantic token 'c'
%tree

pl@nereida:~/LEyapp/examples$ use_list3.pl
ccdd
$VAR1 = bless( {
```

```
        'children' => [
          bless( { 'children' => [], 'attr' => 'c', 'token' => 'c' }, 'TERMINAL' ),
          bless( { 'children' => [], 'attr' => 'c', 'token' => 'c' }, 'TERMINAL' )
        ]
      }, '_PLUS_LIST_1' );
$VAR1 = bless( { 'children' => [] }, '_PLUS_LIST_2' );
```

El nodo asociado con la lista de `d`s aparece vacío por que el terminal `'d'` no fué declarado semántico.

Desaparición de Nodos en Listas

Cuando se trabaja con listas bajo la directiva `%tree` la acción por defecto es el "aplanamiento" de los nodos a los que se aplica el operador `+` en una sóla lista.

En el ejemplo anterior los nodos `'d'` no aparecen pues `'d'` es un token sintáctico. Sin embargo, puede que no sea suficiente declarar `'d'` como semántico.

Cuando se construye el árbol, el algoritmo de construcción de nodos asociados con las listas omite cualquier atributo que no sea una referencia. Por tanto, es necesario garantizar que el atributo asociado con el símbolo sea una referencia (o un token semántico) para asegurar su presencia en la lista de hijos del nodo.

```
pl@nereida:~/LEyapp/examples$ head -19 ListWithRefs1.eyp | cat -n
 1  # ListWithRefs.eyp
 2  %semantic token 'c' 'd'
 3  %{
 4  use Data::Dumper;
 5  %}
 6  %%
 7  S:      'c'+  D+
 8             {
 9                print Dumper($_[1]);
10                print $_[1]->str."\n";
11                print Dumper($_[2]);
12                print $_[2]->str."\n";
13             }
14  ;
15
16  D: 'd'
17  ;
18
19  %%
```

Para activar el modo de construcción de nodos usamos la opción `yybuildingtree` de `YYParse`:

```
pl@nereida:~/LEyapp/examples$ tail -7 ListWithRefs1.eyp | cat -n
     1  sub Run {
     2      my($self)=shift;
     3      $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
     4        yybuildingtree => 1,
     5        #, yydebug => 0x1F
     6      );
     7  }
```

Al ejecutar se producirá una salida similar a esta:

```
pl@nereida:~/LEyapp/examples$ eyapp ListWithRefs1.eyp; use_listwithrefs1.pl
ccdd
$VAR1 = bless( {
                 'children' => [
                                 bless( {
                                          'children' => [],
                                          'attr' => 'c',
                                          'token' => 'c'
                                        }, 'TERMINAL' ),
                                 bless( {
                                          'children' => [],
                                          'attr' => 'c',
                                          'token' => 'c'
                                        }, 'TERMINAL' )
                               ]
               }, '_PLUS_LIST_1' );
_PLUS_LIST_1(TERMINAL,TERMINAL)
$VAR1 = bless( {
                 'children' => []
               }, '_PLUS_LIST_2' );
_PLUS_LIST_2
```

Aunque `'d'` es declarado semántico la acción por defecto asociada con la regla `D: 'd'` en la línea 16 retornará `$_[1]` (esto es, el escalar `'d'`). Dado que no se trata de una referencia no es insertado en la lista de hijos del nodo `_PLUS_LIST`.

Recuperando los Nodos Desaparecidos

La solución es hacer que cada uno de los símbolos a los que afecta el operador de lista sea una referencia.

```
pl@nereida:~/LEyapp/examples$ head -22 ListWithRefs.eyp | cat -n
 1  # ListWithRefs.eyp
 2  %semantic token 'c'
 3  %{
 4  use Data::Dumper;
 5  %}
 6  %%
 7  S:      'c'+  D+
 8             {
 9                print Dumper($_[1]);
10                print $_[1]->str."\n";
11                print Dumper($_[2]);
12                print $_[2]->str."\n";
13             }
14  ;
15
16  D: 'd'
17       {
18         bless { attr => $_[1], children =>[]}, 'DES';
19       }
20  ;
21
22  %%
```

Ahora el atributo asociado con `D` es un nodo y aparece en la lista de hijos del nodo `_PLUS_LIST`:

```
pl@nereida:~/LEyapp/examples$ eyapp ListWithRefs.eyp; use_listwithrefs.pl
ccdd
$VAR1 = bless( {
                 'children' => [
                                 bless( {
                                          'children' => [],
                                          'attr' => 'c',
                                          'token' => 'c'
                                        }, 'TERMINAL' ),
                                 bless( {
                                          'children' => [],
                                          'attr' => 'c',
                                          'token' => 'c'
                                        }, 'TERMINAL' )
                               ]
               }, '_PLUS_LIST_1' );
_PLUS_LIST_1(TERMINAL,TERMINAL)
$VAR1 = bless( {
                 'children' => [
                                 bless( {
                                          'children' => [],
                                          'attr' => 'd'
                                        }, 'DES' ),
                                 bless( {
                                          'children' => [],
                                          'attr' => 'd'
                                        }, 'DES' )
                               ]
               }, '_PLUS_LIST_2' );
_PLUS_LIST_2(DES,DES)
```

Construcción de un Árbol con `Parse::Eyapp::Node->new`

La solución anterior consistente en escribir a mano el código de construcción del nodo puede ser suficiente cuando se trata de un sólo nodo. Escribir a mano el código para la construcción de un árbol con varios nodos puede ser tedioso. Peor aún: aunque el nodo construido en el ejemplo luce como los nodos `Parse::Eyapp` no es realmente un nodo `Parse::Eyapp`. Los nodos `Parse::Eyapp` siempre heredan de la clase `Parse::Eyapp::Node` y por tanto tienen acceso a los métodos definidos en esa clase. La siguiente ejecución con el depurador ilustra este punto:

```
pl@nereida:~/LEyapp/examples$ perl -wd use_listwithrefs.pl

Loading DB routines from perl5db.pl version 1.28
Editor support available.

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

main::(use_listwithrefs.pl:4):  $parser = new ListWithRefs();
  DB<1>  f ListWithRefs.eyp
1       2       #line 3 "ListWithRefs.eyp"
3
4:      use Data::Dumper;
5
6       #line 7 "ListWithRefs.eyp"
7       #line 8 "ListWithRefs.eyp"
```

```
8
9:                      print Dumper($_[1]);
10:                     print $_[1]->str."\n";
```

Mediante el comando `f ListWithRefs.eyp` le indicamos al depurador que los subsiguientes comandos harán referencia a dicho fichero. A continuación ejecutamos el programa hasta alcanzar la acción semántica asociada con la regla `S: 'c'+  D+` (línea 9)

```
  DB<2> c 9     # Continuar hasta la línea 9 de ListWithRefs.eyp
ccdd
ListWithRefs::CODE(0x84ebe5c)(ListWithRefs.eyp:9):
9:                      print Dumper($_[1]);
```

Estamos en condiciones ahora de observar los contenidos de los argumentos:

```
  DB<3> x $_[2]->str
0  '_PLUS_LIST_2(DES,DES)'
  DB<4> x $_[2]->child(0)
0  DES=HASH(0x85c4568)
   'attr' => 'd'
   'children' => ARRAY(0x85c458c)
        empty array
```

El método `str` funciona con el objeto `$_[2]` pues los nodos `_PLUS_LIST_2` heredan de la clase `Parse::Eyapp::Node`. sin embargo, cuando intentamos usarlo con un nodo `DES` obtenemos un error:

```
  DB<6> x $_[2]->child(0)->str
Can't locate object method "str" via package "DES" at \
  (eval 11)[/usr/share/perl/5.8/perl5db.pl:628] line 2, <STDIN> line 1.
  DB<7>
```

Una solución mas robusta que la anterior es usar el constructor `Parse::Eyapp::Node->new`. El método `Parse::Eyapp::Node->new` se usa para construir un bosque de árboles sintácticos. Recibe una secuencia de términos describiendo los árboles y - opcionalmente - una subrutina. La subrutina se utiliza para iniciar los atributos de los nodos recién creados. Después de la creación del árbol la subrutina es llamada por `Parse::Eyapp::Node->new` pasándole como argumentos la lista de referencias a los nodos en el órden en el que aparecen en la secuencia de términos de izquierda a derecha. `Parse::Eyapp::Node->new` retorna una lista de referencias a los nodos recién creados, en el órden en el que aparecen en la secuencia de términos de izquierda a derecha. En un contexto escalar devuelve la referencia al primero de esos árboles. Por ejemplo:

```
pl@nereida:~/LEyapp/examples$ perl -MParse::Eyapp -MData::Dumper -wde 0
main::(-e:1):   0
  DB<1> @t = Parse::Eyapp::Node->new('A(C,D) E(F)', sub { my $i = 0; $_->{n} = $i++ for @_ })
  DB<2> $Data::Dumper::Indent = 0
  DB<3> print Dumper($_)."\n" for @t
$VAR1 = bless( {'n' => 0,'children' => [bless( {'n' => 1,'children' => []}, 'C' ),
                                         bless( {'n' => 2,'children' => []}, 'D' )
                                        ]
               }, 'A' );
$VAR1 = bless( {'n' => 1,'children' => []}, 'C' );
$VAR1 = bless( {'n' => 2,'children' => []}, 'D' );
$VAR1 = bless( {'n' => 3,'children' => [bless( {'n' => 4,'children' => []}, 'F' )]}, 'E' );
$VAR1 = bless( {'n' => 4,'children' => []}, 'F' );
```

Véase el siguiente ejemplo en el cual los nodos asociados con las `'d'` se han convertido en un subárbol:

```
pl@nereida:~/LEyapp/examples$ head -28 ListWithRefs2.eyp| cat -n
# ListWithRefs2.eyp
%semantic token 'c'
%{
use Data::Dumper;
%}
%%
S:  'c'+  D+
      {
         print Dumper($_[1]);
         print $_[1]->str."\n";
         print Dumper($_[2]);
         print $_[2]->str."\n";
      }
;

D: 'd'.d
     {
       Parse::Eyapp::Node->new(
         'DES(TERMINAL)',
          sub {
            my ($DES, $TERMINAL) = @_;
            $TERMINAL->{attr} = $d;
          }
       );
     }
;

%%
```

Para saber mas sobre `Parse::Eyapp::Node->new` consulte la entrada `Parse::Eyapp::Node->new` de la documentación de `Parse::Eyapp`.

```
pl@nereida:~/LEyapp/examples$ eyapp ListWithRefs2.eyp; use_listwithrefs2.pl
ccdd
$VAR1 = bless( {
  'children' => [
    bless( { 'children' => [], 'attr' => 'c', 'token' => 'c' }, 'TERMINAL' ),
    bless( { 'children' => [], 'attr' => 'c', 'token' => 'c' }, 'TERMINAL' )
  ]
}, '_PLUS_LIST_1' );
_PLUS_LIST_1(TERMINAL,TERMINAL)
$VAR1 = bless( {
  'children' => [
    bless( {
      'children' => [
        bless( { 'children' => [], 'attr' => 'd' }, 'TERMINAL' )
      ]
    }, 'DES' ),
    bless( {
      'children' => [
        bless( { 'children' => [], 'attr' => 'd' }, 'TERMINAL' )
      ]
    }, 'DES' )
```

```
  ]
}, '_PLUS_LIST_2' );
_PLUS_LIST_2(DES(TERMINAL),DES(TERMINAL))
```

El Operador de Cierre *

Un operador de lista afecta al factor a su izquierda. Una lista en la parte derecha de una regla cuenta como un único símbolo.

Los operadores `*` y `+` pueden ser usados con el formato `X <* Separator>`. En ese caso lo que se describen son listas separadas por el separador `separator`.

```
pl@nereida:~/LEyapp/examples$ head -25 CsBetweenCommansAndD.eyp | cat -n
 1  # CsBetweenCommansAndD.eyp
 2
 3  %semantic token 'c' 'd'
 4
 5  %{
 6  sub TERMINAL::info {
 7    $_[0]->attr;
 8  }
 9  %}
10  %tree
11  %%
12  S:
13      ('c' <* ','> 'd')*
14        {
15           print "\nNode\n";
16           print $_[1]->str."\n";
17           print "\nChild 0\n";
18           print $_[1]->child(0)->str."\n";
19           print "\nChild 1\n";
20           print $_[1]->child(1)->str."\n";
21           $_[1]
22        }
23  ;
24
25  %%
```

La regla `S: ('c' <* ','> 'd')*` tiene dos elementos en su parte derecha: la lista de `c`s separadas por comas y la `d`. La regla es equivalente a:

```
pl@nereida:~/LEyapp/examples$ eyapp -v CsBetweenCommansAndD.eyp
pl@nereida:~/LEyapp/examples$ head -11 CsBetweenCommansAndD.output | cat -n
 1  Rules:
 2  ------
 3  0:      $start -> S $end
 4  1:      STAR-1 -> STAR-1 ',' 'c'
 5  2:      STAR-1 -> 'c'
 6  3:      STAR-2 -> STAR-1
 7  4:      STAR-2 -> /* empty */
 8  5:      PAREN-3 -> STAR-2 'd'
 9  6:      STAR-4 -> STAR-4 PAREN-3
10  7:      STAR-4 -> /* empty */
11  8:      S -> STAR-4
```

La acción semántica asociada con un operador de lista `*` es retornar una referencia a una lista con los atributos de los elementos a los que se aplica.

Si se trabaja - como en el ejemplo - bajo una directiva de creación de árbol retorna un nodo con la etiqueta `_STAR_LIST_#number` cuyos hijos son los elementos de la lista. El número es el número de orden de la regla tal y como aparece en el fichero `.output`. Es necesario que los elementos sean terminales o referencias para que se incluyan en la lista. Observe como el nodo `PAREN-3` ha sido eliminado del árbol. Los nodos paréntesis son -en general - obviados:

```
pl@nereida:~/LEyapp/examples$ use_csbetweencommansandd.pl
c,c,cd

Node
_STAR_LIST_4(_STAR_LIST_1(TERMINAL[c],TERMINAL[c],TERMINAL[c]),TERMINAL[d])

Child 0
_STAR_LIST_1(TERMINAL[c],TERMINAL[c],TERMINAL[c])

Child 1
TERMINAL[d]
```

Obsérvese también que la coma ha sido eliminada.

Poniendo Nombres a las Listas

Para poner nombre a una lista la directiva `%name` debe preceder al operador tal y como ilustra el siguiente ejemplo:

```
pl@nereida:~/LEyapp/examples$ sed -ne '1,27p' CsBetweenCommansAndDWithNames.eyp | cat -n
 1  # CsBetweenCommansAndDWithNames.eyp
 2
 3  %semantic token 'c' 'd'
 4
 5  %{
 6  sub TERMINAL::info {
 7    $_[0]->attr;
 8  }
 9  %}
10  %tree
11  %%
12  Start: S
13  ;
14  S:
15      ('c' <%name Cs * ','> 'd') %name Cs_and_d *
16        {
17           print "\nNode\n";
18           print $_[1]->str."\n";
19           print "\nChild 0\n";
20           print $_[1]->child(0)->str."\n";
21           print "\nChild 1\n";
22           print $_[1]->child(1)->str."\n";
23           $_[1]
24        }
25  ;
26
27  %%
```

La ejecución muestra que los nodos de lista han sido renombrados:

```
pl@nereida:~/LEyapp/examples$ use_csbetweencommansanddwithnames.pl
c,c,c,cd

Node
Cs_and_d(Cs(TERMINAL[c],TERMINAL[c],TERMINAL[c],TERMINAL[c]),TERMINAL[d])

Child 0
Cs(TERMINAL[c],TERMINAL[c],TERMINAL[c],TERMINAL[c])

Child 1
TERMINAL[d]
```

Opcionales

La utilización del operador `?` indica la presencia o ausencia del factor a su izquierda. La gramática:

```
pl@nereida:~/LEyapp/examples$ head -11 List5.yp | cat -n
     1	%semantic token 'c'
     2	%tree
     3	%%
     4	S: 'c' 'c'?
     5	     {
     6	       print $_[2]->str."\n";
     7	       print $_[2]->child(0)->attr."\n" if $_[2]->children;
     8	    }
     9	;
    10
    11	%%
```

produce la siguiente gramática:

```
l@nereida:~/LEyapp/examples$ eyapp -v List5
pl@nereida:~/LEyapp/examples$ head -7 List5.output
Rules:
------
0:      $start -> S $end
1:      OPTIONAL-1 -> 'c'
2:      OPTIONAL-1 -> /* empty */
3:      S -> 'c' OPTIONAL-1
```

Cuando no se trabaja bajo directivas de creación de árbol el atributo asociado es una lista (vacía si el opcional no aparece). Bajo la directiva `%tree` el efecto es crear el nodo:

```
pl@nereida:~/LEyapp/examples$ use_list5.pl
cc
_OPTIONAL_1(TERMINAL)
c
pl@nereida:~/LEyapp/examples$ use_list5.pl
c
_OPTIONAL_1
```

Agrupamientos

Es posible agrupar mediante paréntesis una subcadena de la parte derecha de una regla de producción. La introducción de un paréntesis implica la introducción de una variable adicional cuya única producción es la secuencia de símbolos entre paréntesis. Así la gramática:

```
pl@nereida:~/LEyapp/examples$ head -6 Parenthesis.eyp | cat -n
   1  %%
   2  S:
   3        ('a' S ) 'b'  { shift; [ @_ ] }
   4      | 'c'
   5  ;
   6  %%
```

genera esta otra gramática:

```
pl@nereida:~/LEyapp/examples$ eyapp -v Parenthesis.eyp; head -6 Parenthesis.output
Rules:
------
0:      $start -> S $end
1:      PAREN-1 -> 'a' S
2:      S -> PAREN-1 'b'
3:      S -> 'c'
```

Por defecto, la regla semántica asociada con un paréntesis consiste en formar una lista con los atributos de los símbolos entre paréntesis:

```
pl@nereida:~/LEyapp/examples$ cat -n use_parenthesis.pl
     1  #!/usr/bin/perl -w
     2  use Parenthesis;
     3  use Data::Dumper;
     4
     5  $Data::Dumper::Indent = 1;
     6  $parser = Parenthesis->new();
     7  print Dumper($parser->Run);
pl@nereida:~/LEyapp/examples$ use_parenthesis.pl
acb
$VAR1 = [
  [ 'a', 'c' ], 'b'
];
pl@nereida:~/LEyapp/examples$ use_parenthesis.pl
aacbb
$VAR1 = [
  [
    'a',
    [ [ 'a', 'c' ], 'b' ]
  ],
  'b'
];
```

Cuando se trabaja bajo una directiva de creación de árbol o se establece el atributo con el método `YYBuildingtree` la acción semántica es construir un nodo con un hijo por cada atributo de cada símbolo entre paréntesis. Si el atributo no es una referencia no es insertado como hijo del nodo.

Veamos un ejemplo:

```
pl@nereida:~/LEyapp/examples$ head -23 List2.yp | cat -n
 1  %{
 2  use Data::Dumper;
 3  %}
 4  %semantic token 'a' 'b' 'c'
 5  %tree
 6  %%
 7  S:
 8        (%name AS 'a' S )'b'
 9          {
10            print "S -> ('a' S )'b'\n";
11            print "Atributo del Primer Símbolo:\n".Dumper($_[1]);
12            print "Atributo del Segundo símbolo: $_[2]\n";
13            $_[0]->YYBuildAST(@_[1..$#_]);
14          }
15      | 'c'
16          {
17            print "S -> 'c'\n";
18            my $r = Parse::Eyapp::Node->new(qw(C(TERMINAL)), sub { $_[1]->attr('c') }) ;
19            print Dumper($r);
20            $r;
21          }
22  ;
23  %%
```

El ejemplo muestra (línea 8) como renombrar un nodo `_PAREN` asociado con un paréntesis. Se escribe la directiva `%name CLASSNAME` despúes del paréntesis de apertura.

La llamada de la línea 13 al método `YYBuildAST` con argumentos los atributos de los símbolos de la parte derecha se encarga de retornar el nodo describiendo la regla de producción actual. Observe que la línea 13 puede ser reescrita como:

```
goto &Parse::Eyapp::Driver::YYBuildAST;
```

En la línea 18 se construye explícitamente un nodo para la regla usando `Parse::Eyapp::Node->new`. El manejador pasado como segundo argumento se encarga de establecer un valor para el atributo `attr` del nodo `TERMINAL` que acaba de ser creado.

Veamos una ejecución:

```
pl@nereida:~/LEyapp/examples$ use_list2.pl
aacbb
S -> 'c'
$VAR1 = bless( {
  'children' => [
    bless( {
      'children' => [],
      'attr' => 'c'
    }, 'TERMINAL' )
  ]
}, 'C' );
```

La primera reducción ocurre por la regla no recursiva. La ejecución muestra el árbol construido mediante la llamada a `Parse::Eyapp::Node->new` de la línea 18.

La ejecución continúa con una reducción o anti-derivación por la regla `S -> ('a' S )'b'`. La acción de las líneas 9-14 hace que se muestre el atributo asociado con `('a' S)` o lo que es lo mismo con la variable sintáctica adicional `PAREN-1` asi como el atributo de `'b'`:

```
S -> (’a’ S )’b’
Atributo del Primer Símbolo:
$VAR1 = bless( {
    ’children’ => [
      bless( { ’children’ => [], ’attr’ => ’a’, ’token’ => ’a’ }, ’TERMINAL’ ),
      bless( { ’children’ => [ bless( { ’children’ => [], ’attr’ => ’c’ }, ’TERMINAL’ )
     ]
   }, ’C’ )
  ]
}, ’AS’ );
Atributo del Segundo símbolo: b
```

La última reducción visible es por la regla `S -> (’a’ S )’b’`:

```
S -> (’a’ S )’b’
Atributo del Primer Símbolo:
$VAR1 = bless( {
  ’children’ => [
    bless( { ’children’ => [], ’attr’ => ’a’, ’token’ => ’a’ }, ’TERMINAL’ ),
    bless( {
      ’children’ => [
        bless( {
          ’children’ => [
            bless( { ’children’ => [], ’attr’ => ’a’, ’token’ => ’a’ }, ’TERMINAL’ ),
            bless( {
              ’children’ => [
                bless( { ’children’ => [], ’attr’ => ’c’ }, ’TERMINAL’ )
              ]
            }, ’C’ )
          ]
        }, ’AS’ ),
        bless( { ’children’ => [], ’attr’ => ’b’, ’token’ => ’b’ }, ’TERMINAL’ )
      ]
    }, ’S_2’ )
  ]
}, ’AS’ );
Atributo del Segundo símbolo: b
```

Acciones Entre Paréntesis

Aunque no es una práctica recomendable es posible introducir acciones en los paréntesis como en este ejemplo:

```
pl@nereida:~/LEyapp/examples$ head -16 ListAndAction.eyp | cat -n
 1  # ListAndAction.eyp
 2  %{
 3  my $num = 0;
 4  %}
 5
 6  %%
 7  S:      ’c’
 8              {
 9                print "S -> c\n"
10              }
11      |    (’a’ {$num++; print "Seen <$num> ’a’s\n"; $_[1] }) S ’b’
12              {
```

```
       print "S -> (a ) S b\n"
      }
;
%%
```

Al ejecutar el ejemplo con la entrada **aaacbbb** se obtiene la siguiente salida:

```
pl@nereida:~/LEyapp/examples$ use_listandaction.pl
aaacbbb
Seen <1> 'a's
Seen <2> 'a's
Seen <3> 'a's
S -> c
S -> (a ) S b
S -> (a ) S b
S -> (a ) S b
```

3.20. El método `str` en Mas Detalle

El método `str` de los Nodos

Para dar soporte al análisis de los árboles y su representación, `Parse::Eyapp` provee el método `str`. El siguiente ejemplo con el depurador muestra el uso del método `str`. El método `Parse::Eyapp::Node::str` retorna una cadena que describe como término el árbol enraizado en el nodo que se ha pasado como argumento.

El método `info`

El método `str` cuando visita un nodo comprueba la existencia de un método `info` para la clase del nodo. Si es así el método será llamado. Obsérvese como en las líneas 5 y 6 proveemos de métodos `info` a los nodos `TERMINAL` y `FUNCTION`. La consecuencia es que en la llamada de la línea 7 el término que describe al árbol es decorado con los nombres de funciones y los atributos y números de línea de los terminales.

Modo de uso de `str`

El método `str` ha sido concebido como una herramienta de ayuda a la depuración y verificación de los programas árbol. La metodología consiste en incorporar las pequeñas funciones `info` al programa en el que trabajamos. Por ejemplo:

```
sub Run {
 my($self)=shift;
 ................
}

sub TERMINAL::info {
  my @a = join ':', @{$_[0]->{attr}};
  return "[@a]"
}

sub FUNCTION::info {
  return "[".$_[0]->{function_name}[0]."]"
}

sub BLOCK::info {
  return "[".$_[0]->{line}."]"
}
```

Variables que Gobiernan la Conducta de `str`

Las variables de paquete que gobiernan la conducta de `str` y sus valores por defecto aparecen en la siguiente lista:

- `our @PREFIXES = qw(Parse::Eyapp::Node::)`

  La variable de paquete `Parse::Eyapp::Node::PREFIXES` contiene la lista de prefijos de los nombres de tipo que serán eliminados cuando `str` muestra el término. Por defecto contiene la cadena `'Parse::Eyapp::Node::'`.

- `our $INDENT = 0` La variable `$Parse::Eyapp::Node::INDENT` controla el formato de presentación usado por `Parse::Eyapp::Node::str`:

  1. Si es `0` la representación es compacta.
  2. Si es 1 se usará un sangrado razonable.
  3. Si es 2 cada paréntesis cerrar que este a una distancia mayor de `$Parse::Eyapp::Node::LINESEP` líneas será comentado con el tipo del nodo. Por defecto la variable `$Parse::Eyapp::Node::LINESEP` está a 4. Véase la página 481 para un ejemplo con `$INDENT` a 2 y `$Parse::Eyapp::Node::LINESEP` a 4.

- `our $STRSEP = ','`

  Separador de nodos en un término. Por defecto es la coma.

- `our $DELIMITER = '['`

  Delimitador de presentación de la información proveída por el método `info`. Por defecto los delimitadores son corchetes. Puede elegirse uno cualquiera de la lista:

  `'[', '{', '(', '<'`

  si se define como la cadena vacía o `undef` no se usarán delimitadores.

- `our $FOOTNOTE_HEADER = "\n---------------------------\n"`

  Delimitador para las notas a pie de página.

- `our $FOOTNOTE_SEP = ")\n"` Separador de la etiqueta/número de la nota al pie

- `our $FOOTNOTE_LEFT = '^{'`, `our $FOOTNOTE_RIGHT = '}'`

  Definen el texto que rodea al número de referencia en el nodo.

Notas a Pié de Página o Footnotes

El siguiente ejemplo ilustra el manejo de *notas a pié de árbol* usando `str`. Se han definido los siguientes métodos:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code> sed -ne '677,$p' Simple6.eyp
$Parse::Eyapp::Node::INDENT = 1;
sub TERMINAL::info {
  my @a = join ':', @{$_[0]->{attr}};
  return "@a"
}

sub PROGRAM::footnote {
  return "Types:\n"
         .Dumper($_[0]->{types}).
         "Symbol Table:\n"
         .Dumper($_[0]->{symboltable})
```

```
}

sub FUNCTION::info {
  return $_[0]->{function_name}[0]
}

sub FUNCTION::footnote {
  return Dumper($_[0]->{symboltable})
}

sub BLOCK::info {
  return $_[0]->{line}
}

sub VAR::info {
  return $_[0]->{definition}{type} if defined $_[0]->{definition}{type};
  return "No declarado!";
}

*FUNCTIONCALL::info = *VARARRAY::info = \&VAR::info;
```

El resultado para el programa de entrada:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code> cat salida
int a,b;

int f(char c) {
 a[2] = 4;
 b[1][3] = a + b;
 c = c[5] * 2;
 return g(c);
}
```

produce una salida por `stderr`:

```
Identifier g not declared
```

y la siguiente descripción de la estructura de datos:

```
PROGRAM^{0}(
  FUNCTION{f}^{1}(
    ASSIGN(
      VARARRAY{INT}(
        TERMINAL{a:4},
        INDEXSPEC(
          INUM(
            TERMINAL{2:4}
          )
        )
      ),
      INUM(
        TERMINAL{4:4}
      )
    ),
    ASSIGN(
      VARARRAY{INT}(
```

```
        TERMINAL{b:5},
        INDEXSPEC(
          INUM(
            TERMINAL{1:5}
          ),
          INUM(
            TERMINAL{3:5}
          )
        )
      ),
      PLUS(
        VAR{INT}(
          TERMINAL{a:5}
        ),
        VAR{INT}(
          TERMINAL{b:5}
        )
      )
    ),
    ASSIGN(
      VAR{CHAR}(
        TERMINAL{c:6}
      ),
      TIMES(
        VARARRAY{CHAR}(
          TERMINAL{c:6},
          INDEXSPEC(
            INUM(
              TERMINAL{5:6}
            )
          )
        ),
        INUM(
          TERMINAL{2:6}
        )
      )
    ),
    RETURN(
      FUNCTIONCALL{No declarado!}(
        TERMINAL{g:7},
        ARGLIST(
          VAR{CHAR}(
            TERMINAL{c:7}
          )
        )
      )
    )
  )
)
---------------------------
0)
Types:
$VAR1 = {
```

```
  'F(X_1(CHAR),INT)' => bless( {
    'children' => [
      bless( {
        'children' => [
          bless( {
            'children' => []
          }, 'CHAR' )
        ]
      }, 'X_1' ),
      bless( {
        'children' => []
      }, 'INT' )
    ]
  }, 'F' ),
  'CHAR' => bless( {
    'children' => []
  }, 'CHAR' ),
  'INT' => bless( {
    'children' => []
  }, 'INT' )
};
Symbol Table:
$VAR1 = {
  'a' => {
    'type' => 'INT',
    'line' => 1
  },
  'b' => {
    'type' => 'INT',
    'line' => 1
  },
  'f' => {
    'type' => 'F(X_1(CHAR),INT)',
    'line' => 3
  }
};

---------------------------
1)
$VAR1 = {
  'c' => {
    'type' => 'CHAR',
    'param' => 1,
    'line' => 3
  }
};
```

Usando `str` sobre una Lista de Áboles

Para usar **`str`** sobre una lista de árboles la llamada deberá hacerse como método de clase, i.e. con el nombre de la clase como prefijo y no con el objeto:

`Parse::Eyapp::Node->str(@forest)`

cuando quiera convertir a su representación término mas de un árbol. El código:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code> sed -ne '652,658p' Simple4.eyp \
                                          | cat -n
     1       local $Parse::Eyapp::Node::INDENT = 2;
     2       local $Parse::Eyapp::Node::DELIMITER = "";
     3       print $t->str."\n";
     4       {
     5         local $" = "\n";
     6         print Parse::Eyapp::Node->str(@blocks)."\n";
     7       }
```

produce la salida:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code> eyapp Simple4 ;\
                                          treereg SimpleTrans.trg ;\
                                          usesimple4.pl
****************
f() {
  int a,b[1][2],c[1][2][3];
  char d[10];
  b[0][1] = a;
}

PROGRAM(
  FUNCTION[f](
    ASSIGN(
      VARARRAY(
        TERMINAL[b:4],
        INDEXSPEC(
          INUM(
            TERMINAL[0:4]
          ),
          INUM(
            TERMINAL[1:4]
          )
        ) # INDEXSPEC
      ) # VARARRAY,
      VAR(
        TERMINAL[a:4]
      )
    ) # ASSIGN
  ) # FUNCTION
) # PROGRAM

Match[PROGRAM:0:blocks](
  Match[FUNCTION:1:blocks:[f]]
)
```

Obsérvense los comentarios `# TIPODENODO` que acompañan a los paréntesis cerrar cuando estos están lejos (esto es, a una distancia de mas de `$Parse::Eyapp::Node::LINESEP` líneas) del correspondiente paréntesis abrir. Tales comentarios son consecuencia de haber establecido el valor de `$Parse::Eyapp::Node::INDENT` a 2.

3.21. El Método `descendant`

El método `descendant` es un método de los objetos `Parse::Eyapp::Node` y retorna una referencia al nodo descrito por la cadena de coordenadas que se le pasa como argumento. En este sentido el método `child` es una especialización de `descendant`.

```
  DB<7> x $t->child(0)->child(0)->child(1)->child(0)->child(2)->child(1)->str
0  '
BLOCK[8:4:test]^{0}(
  CONTINUE[10,10]
)
  DB<8> x $t->descendant('.0.0.1.0.2.1')->str
0  '
BLOCK[8:4:test]^{0}(
  CONTINUE[10,10]
)
```

3.22. Conceptos Básicos del Análisis LR

Los analizadores generados por `eyapp` entran en la categoría de analizadores *LR*. Estos analizadores construyen una derivación a derechas inversa (o *antiderivación*). De ahí la R en LR (del inglés *rightmost derivation*). El árbol sintáctico es construido de las hojas hacia la raíz, siendo el último paso en la antiderivación la construcción de la primera derivación desde el símbolo de arranque.

Empezaremos entonces considerando las frases que pueden aparecer en una derivación a derechas. Tales frases consituyen el lenguaje FSD:

Definición 3.22.1. *Dada una gramática* $G = (\Sigma, V, P, S)$ *no ambigua, se denota por* FSD *(lenguaje de las formas Sentenciales a Derechas) al lenguaje de las sentencias que aparecen en una derivación a derechas desde el símbolo de arranque.*

$$FSD = \left\{ \alpha \in (\Sigma \cup V)* : \exists S \overset{*}{\underset{RM}{\Longrightarrow}} \alpha \right\}$$

*Donde la notacion RM indica una derivación a derechas (*rightmost*). Los elementos de* FSD *se llaman "formas sentenciales derechas".*

Dada una gramática no ambigua $G = (\Sigma, V, P, S)$ y una frase $x \in L(G)$ el proceso de antiderivación consiste en encontrar la última derivación a derechas que dió lugar a x. Esto es, si $x \in L(G)$ es porque existe una derivación a derechas de la forma

$$S \overset{*}{\Longrightarrow} yAz \Longrightarrow ywz = x.$$

El problema es averiguar que regla $A \rightarrow w$ se aplicó y en que lugar de la cadena x se aplicó. En general, si queremos antiderivar una forma sentencial derecha $\beta\alpha w$ debemos averiguar por que regla $A \rightarrow \alpha$ seguir y en que lugar de la forma (después de β en el ejemplo) aplicarla.

$$S \overset{*}{\Longrightarrow} \beta A w \Longrightarrow \beta\alpha w.$$

La pareja formada por la regla y la posición se denomina mango o manecilla de la forma. Esta denominación viene de la visualización gráfica de la regla de producción como una mano que nos permite escalar hacia arriba en el árbol. Los "dedos" serían los símbolos en la parte derecha de la regla de producción.

Definición 3.22.2. *Dada una gramática* $G = (\Sigma, V, P, S)$ *no ambigua, y dada una forma sentencial derecha* $\alpha = \beta\gamma x$*, con* $x \in \Sigma^*$*, el* mango *o* handle *de* α *es la última producción/posición que dió lugar a* α*:*

$$S \underset{RM}{\overset{*}{\Longrightarrow}} \beta B x \Longrightarrow \beta\gamma x = \alpha$$

Escribiremos: $handle(\alpha) = (B \rightarrow \gamma, \beta\gamma)$. La función *handle* tiene dos componentes: $handle_1(\alpha) = B \rightarrow \gamma$ y $handle_2(\alpha) = \beta\gamma$

Si dispusieramos de un procedimiento que fuera capaz de identificar el mango, esto es, de detectar la regla y el lugar en el que se posiciona, tendríamos un mecanismo para construir un analizador. Lo curioso es que, a menudo es posible encontrar un autómata finito que reconoce el lenguaje de los prefijos $\beta\gamma$ que terminan en el mango. Con mas precisión, del lenguaje:

Definición 3.22.3. *El conjunto de* prefijos viables *de una gramática G se define como el conjunto:*

$$PV = \left\{ \delta \in (\Sigma \cup V)* : \exists S \underset{RM}{\overset{*}{\Longrightarrow}} \alpha \ y \ \delta \ es \ un \ prefijo \ de \ handle_2(\alpha) \right\}$$

Esto es, es el lenguaje de los prefijos viables es el conjunto de frases que son prefijos de $handle_2(\alpha)) = \beta\gamma$, siendo α una forma sentencial derecha ($\alpha \in FSD$). Los elementos de PV se denominan prefijos viables.

Obsérvese que si se dispone de un autómata que reconoce PV entonces se dispone de un mecanismo para investigar el lugar y el aspecto que pueda tener el mango. Si damos como entrada la sentencia $\alpha = \beta\gamma x$ a dicho autómata, el autómata aceptará la cadena $\beta\gamma$ pero rechazará cualquier extensión del prefijo. Ahora sabemos que el mango será alguna regla de producción de G cuya parte derecha sea un sufijo de $\beta\gamma$.

Definición 3.22.4. *El siguiente autómata finito no determinista puede ser utilizado para reconocer el lenguaje de los prefijos viables PV:*

- $Alfabeto = V \cup \Sigma$
- *Los estados del autómata se denominan LR(0) items. Son parejas formadas por una regla de producción de la gramática y una posición en la parte derecha de la regla de producción. Por ejemplo,* $(E \rightarrow E + E, 2)$ *sería un LR(0) item para la gramática de las expresiones.*

  Conjunto de Estados:

  $Q = \{(A \rightarrow \alpha, n) : A \rightarrow \alpha \in P, \ n \leq |\alpha|\}$

  La notación $\mid \alpha \mid$ *denota la longitud de la cadena* $\mid \alpha \mid$*. En vez de la notación* $(A \rightarrow \alpha, n)$ *escribiremos:* $A \rightarrow \beta_{\uparrow}\gamma = \alpha$*, donde la flecha ocupa el lugar indicado por el número* $n = \mid \beta \mid$ *:*
- *Función de transición:*

  $\delta(A \rightarrow \alpha_{\uparrow} X \beta, X) = A \rightarrow \alpha X_{\uparrow} \beta \ \forall X \in V \cup \Sigma$

  $\delta(A \rightarrow \alpha_{\uparrow} B \beta, \epsilon) = B \rightarrow \gamma \forall B \in V$
- *Estado de arranque: Se añade la "superregla"* $S' \rightarrow S$ *a la gramática* $G = (\Sigma, V, P, S)$*. El LR(0) item* $S' \rightarrow_{\uparrow} S$ *es el estado de arranque.*
- *Todos los estados definidos (salvo el de muerte) son de aceptación.*

Denotaremos por $LR(0)$ a este autómata. Sus estados se denominan $LR(0) - items$. La idea es que este autómata nos ayuda a reconocer los prefijos viables PV.

Una vez que se tiene un autómata que reconoce los prefijos viables es posible construir un analizador sintáctico que construye una antiderivación a derechas. La estrategia consiste en "alimentar" el autómata con la forma sentencial derecha. El lugar en el que el autómata se detiene, rechazando indica el lugar exacto en el que termina el *handle* de dicha forma.

Ejemplo 3.22.1. *Consideremos la gramática:*

$$\begin{aligned} S &\rightarrow a\ S\ b \\ S &\rightarrow \epsilon \end{aligned}$$

El lenguaje generado por esta gramática es $L(G) = \{a^n b^n \ :\ n \geq 0\}$ *Es bien sabido que el lenguaje* $L(G)$ *no es regular. La figura 3.22.1 muestra el* autómata finito no determinista con ϵ-transiciones *(*NFA*) que reconoce los prefijos viables de esta gramática, construido de acuerdo con el algoritmo 3.22.4.*

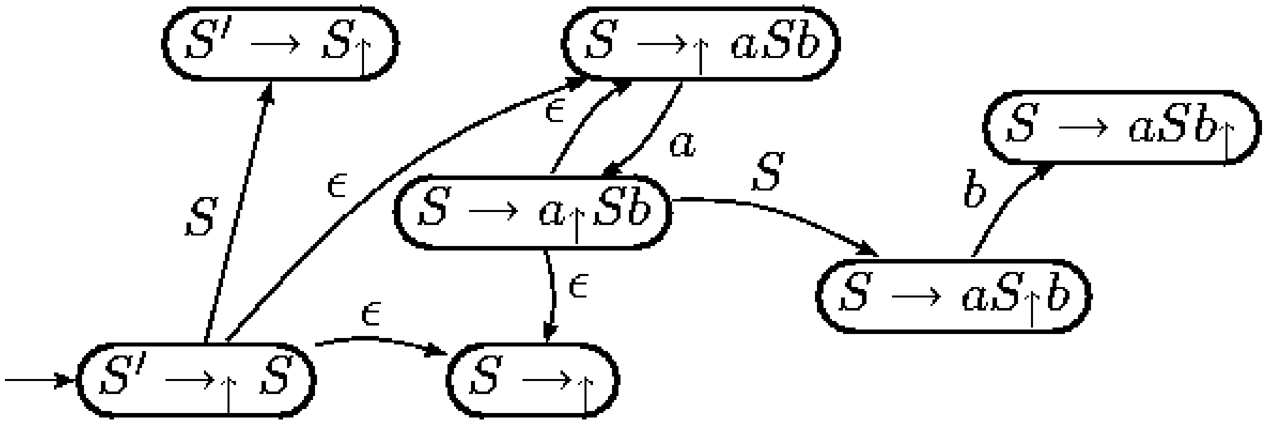


Figura 3.1: NFA que reconoce los prefijos viables

Ejercicio 3.22.1. *Simule el comportamiento del autómata sobre la entrada aabb. ¿Donde rechaza? ¿En que estados está el autómata en el momento del rechazo?. ¿Qué etiquetas tienen? Haga también las trazas del autómata para las entradas aaSbb y aSb. ¿Que antiderivación ha construido el autómata con sus sucesivos rechazos? ¿Que terminales se puede esperar que hayan en la entrada cuando se produce el rechazo del autómata?*

3.23. Construcción de las Tablas para el Análisis SLR

3.23.1. Los conjuntos de Primeros y Siguientes

Repasemos las nociones de conjuntos de *Primeros* y *siguientes*:

Definición 3.23.1. *Dada una gramática* $G = (\Sigma, V, P, S)$ *y un símbolo* $\alpha \in (V \cup \Sigma)^*$ *se define el conjunto* $FIRST(\alpha)$ *como:*

$$FIRST(\alpha) = \left\{ b \in \Sigma : \alpha \overset{*}{\Longrightarrow} b\beta \right\} \cup N(\alpha)$$

donde:

$$N(\alpha) = \left\{ \begin{array}{ll} \{\epsilon\} & si\ \alpha \overset{*}{\Longrightarrow} \epsilon \\ \emptyset & en\ otro\ caso \end{array} \right.$$

Definición 3.23.2. *Dada una gramática* $G = (\Sigma, V, P, S)$ *y una variable* $A \in V$ *se define el conjunto* $FOLLOW(A)$ *como:*

$$FOLLOW(A) = \left\{ b \in \Sigma : \exists\ S \overset{*}{\Longrightarrow} \alpha A b \beta \right\} \cup E(A)$$

donde

$$E(A) = \left\{ \begin{array}{ll} \{\$\} & si\ S \overset{*}{\Longrightarrow} \alpha A \\ \emptyset & en\ otro\ caso \end{array} \right.$$

Algoritmo 3.23.1. *Construcción de los conjuntos* $FIRST(X)$

1. *Si* $X \in \Sigma$ *entonces* $FIRST(X) = X$
2. *Si* $X \to \epsilon$ *entonces* $FIRST(X) = FIRST(X) \cup \{\epsilon\}$
3. *Si*$X \in V$ *y* $X \to Y_1 Y_2 \cdots Y_k \in P$ *entonces*

```
$i = 1;$
do
  $FIRST(X) = FIRST(X) \cup FIRST(Y_i) - \{\epsilon\};$
  $i++;$
mientras $(\epsilon \in FIRST(Y_i)$ and $(i \leq k))$
si $(\epsilon \in FIRST(Y_k)$ and $i > k)$ $FIRST(X) = FIRST(X) \cup \{\epsilon\}$
```

Este algoritmo puede ser extendido para calcular $FIRST(\alpha)$ para $\alpha = X_1 X_2 \cdots X_n \in (V \cup \Sigma)^*$.

Algoritmo 3.23.2. *Construcción del conjunto* $FIRST(\alpha)$

```
$i = 1;$
$FIRST(\alpha) = \emptyset;$
do
  $FIRST(\alpha) = FIRST(\alpha) \cup FIRST(X_i) - \{\epsilon\};$
  $i++;$
mientras $(\epsilon \in FIRST(X_i)$ and $(i \leq n))$
si $(\epsilon \in FIRST(X_n)$ and $i > n)$ $FIRST(\alpha) = FIRST(X) \cup \{\epsilon\}$
```

Algoritmo 3.23.3. *Construcción de los conjuntos* $FOLLOW(A)$ *para las variables sintácticas* $A \in V$*:*
Repetir los siguientes pasos hasta que ninguno de los conjuntos $FOLLOW$ *cambie:*

1. $FOLLOW(S) = \{\$\}$ *($ representa el final de la entrada)*
2. *Si* $A \to \alpha B \beta$ *entonces*

$$FOLLOW(B) = FOLLOW(B) \cup (FIRST(\beta) - \{\epsilon\})$$

3. *Si* $A \to \alpha B$ *o bien* $A \to \alpha B \beta$ *y* $\epsilon \in FIRST(\beta)$ *entonces*

$$FOLLOW(B) = FOLLOW(B) \cup FOLLOW(A)$$

3.23.2. Construcción de las Tablas

Para la construcción de las tablas de un analizador SLR se construye el *autómata finito determinista* (*DFA*) (Q, Σ, δ, q_0) equivalente al NFA presentado en la sección 3.22 usando el *algoritmo de construcción del subconjunto*.

Como recordará, en la construcción del subconjunto, partiendo del estado de arranque q_0 del NFA con ϵ-transiciones se calcula su *clausura* $\overline{\{q_0\}}$ y las clausuras de los conjuntos de estados $\overline{\delta(\overline{\{q_0\}}, a)}$ a los que transita. Se repite el proceso con los conjuntos resultantes hasta que no se introducen nuevos conjuntos-estado.

Clausura de un Conjunto de Estados

La clausura $\overline{A}$ de un subconjunto de estados del autómata A esta formada por todos los estados que pueden ser alcanzados mediante transiciones etiquetadas con la palabra vacía (denominadas ϵ transiciones) desde los estados de A. Se incluyen en $\overline{A}$, naturalmente los estados de A.

$$\overline{A} = \{q \in Q \;/\; \exists q' \in Q \;:\; \hat{\delta}(q, \epsilon) = q\}$$

Aquí $\hat{\delta}$ denota la *función de transición del autómata* extendida a cadenas de Σ^*.

$$\hat{\delta}(q, x) = \begin{cases} \delta(\hat{\delta}(q, y), a) & \text{si } x = ya \\ q & \text{si } x = \epsilon \end{cases} \tag{3.1}$$

En la práctica, y a partir de ahora así lo haremos, se prescinde de diferenciar entre δ y $\hat{\delta}$ usándose indistintamente la notación δ para ambas funciones.

La clausura puede ser computada usando una estructura de pila o aplicando la expresión recursiva dada en la ecuación 3.1. Una forma de computarla viene dada por el siguiente seudocódigo:

```
function closure(I : set of LR(0)-items)
begin
  J = I;
  repeat
    changes = FALSE;
    for A->alpha . B beta in J do
      for B->gamma in G do
        next if B->.gamma in J
        insert B->.gamma in J
        changes = TRUE;
      end for
    end for
  until nochanges;
  return J
end
```

Ejemplo de Construcción del DFA

Para el NFA mostrado en el ejemplo 3.22.1 el DFA construído mediante esta técnica es el que se muestra en la figura 3.23.2. Se ha utilizado el símbolo `#` como marcador. Se ha omitido el número 3 para que los estados coincidan en numeración con los generados por `eyapp` (véase el cuadro 3.23.2).

Las Tablas de Saltos y de Acciones

Un analizador sintáctico LR utiliza una tabla para su análisis. Esa tabla se construye a partir de la tabla de transiciones del DFA. De hecho, la tabla se divide en dos tablas, una llamada *tabla de saltos* o *tabla de gotos* y la otra *tabla de acciones*.

La tabla *goto* de un analizador *SLR* no es más que la tabla de transiciones del autómata DFA obtenido aplicando la construcción del subconjunto al NFA definido en 3.22.4. De hecho es la tabla de transiciones restringida a V (recuerde que el alfabeto del autómata es $V \cup \Sigma$). Esto es,

$$\delta_{|V \times Q} : V \times Q \to Q.$$
$$\text{donde se define } goto(i, A) = \delta(A, I_i)$$

La parte de la función de transiciones del DFA que corresponde a los terminales que no producen rechazo, esto es, $\delta_{|\Sigma \times Q} : \Sigma \times Q \to Q$ se adjunta a una tabla que se denomina *tabla de acciones*. La tabla de acciones es una tabla de doble entrada en los estados y en los símbolos de Σ. Las acciones de transición ante terminales se denominan *acciones de desplazamiento* o (*acciones shift*):

$$\delta_{|\Sigma \times Q} : \Sigma \times Q \to Q$$
$$\text{donde se define } action(i, a) = \delta(a, I_i)$$

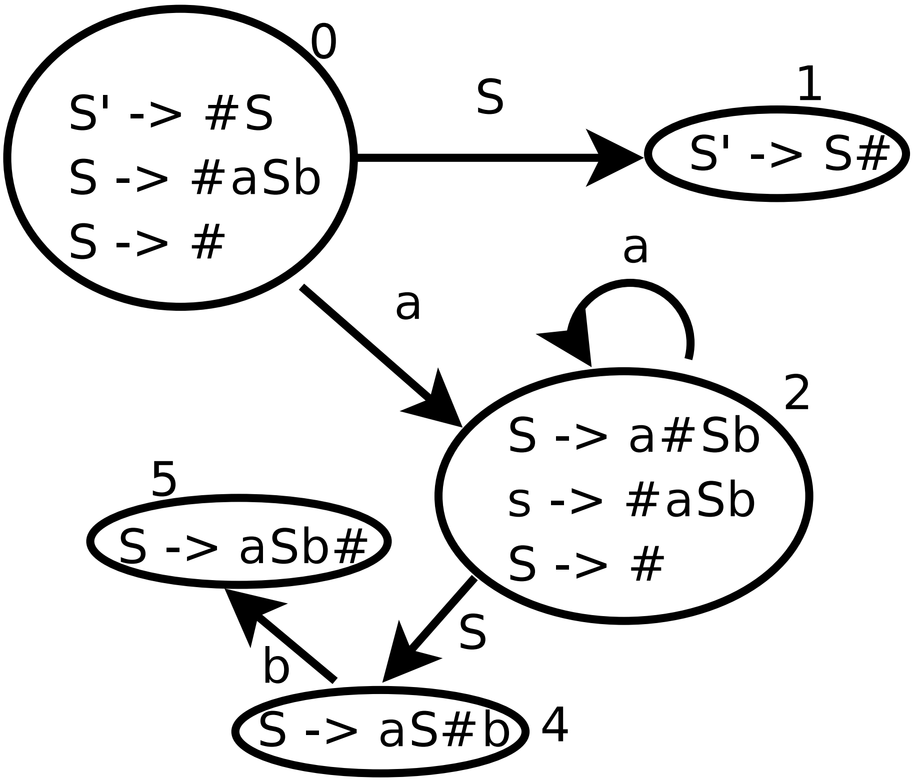


Figura 3.2: DFA equivalente al NFA de la figura 3.22.1

Ante que Terminales se debe Reducir

Cuando un estado s contiene un LR(0)-item de la forma $A \rightarrow \alpha_{\uparrow}$, esto es, el estado corresponde a un posible rechazo, ello indica que hemos llegado a un final del prefijo viable, que hemos visto α y que, por tanto, es probable que $A \rightarrow \alpha$ sea el *handle* de la forma sentencial derecha actual. Por tanto, añadiremos en entradas de la forma (s, a) de la tabla de acciones una acción que indique que hemos encontrado el mango en la posición actual y que la regla asociada es $A \rightarrow \alpha$. A una acción de este tipo se la denomina *acción de reducción*.

La cuestión es, ¿para que valores de $a \in \Sigma$ debemos disponer que la acción para (s, a) es de reducción? Podríamos decidir que ante cualquier terminal $a \in \Sigma$ que produzca un rechazo del autómata, pero podemos ser un poco mas selectivos. No cualquier terminal puede estar en la entrada en el momento en el que se produce la antiderivación o reducción. Observemos que si $A \rightarrow \alpha$ es el *handle* de γ es porque:

$$\exists S \overset{*}{\underset{RM}{\Longrightarrow}} \beta A b x \overset{*}{\underset{RM}{\Longrightarrow}} \beta \alpha b x = \gamma$$

Por tanto, cuando estamos reduciendo por $A \rightarrow \alpha$ los únicos terminales legales que cabe esperar en una reducción por $A \rightarrow \alpha$ son los terminales $b \in FOLLOW(A)$.

Algoritmo de Construcción de Las Tablas SLR

Dada una gramática $G = (\Sigma, V, P, S)$, podemos construir las tablas de acciones (*action table*) y transiciones (*gotos table*) mediante el siguiente algoritmo:

Algoritmo 3.23.4. *Construcción de Tablas* SLR

1. *Utilizando el Algoritmo de Construcción del Subconjunto, se construye el Autómata Finito Determinista (DFA)* $(Q, V \cup \Sigma, \delta, I_0, F)$ *equivalente al Autómata Finito No Determinista (NFA) definido en 3.22.4. Sea* $C = \{I_1, I_2, \cdots I_n\}$ *el conjunto de estados del DFA. Cada estado* I_i *es un conjunto de LR(0)-items o estados del NFA. Asociemos un índice i con cada conjunto* I_i*.*

2. *La tabla de* gotos *no es más que la función de transición del autómata restringida a las variables de la gramática:*

$$goto(i, A) = \delta(I_i, A) \text{ para todo } A \in V$$

3. *Las acciones para el estado I_i se determinan como sigue:*

    a) *Si $A \to \alpha_\uparrow a\beta \in I_i$, $\delta(I_i, a) = I_j$, $a \in \Sigma$ entonces:*

    $$action[i][a] = shift\ j$$

    b) *Si $S' \to S_\uparrow \in I_i$ entonces*

    $$action[i][\$] = accept$$

    c) *Para cualquier otro caso de la forma $A \to \alpha_\uparrow \in I_i$ distinto del anterior hacer*

    $$\forall a \in \ FOLLOW(A) : \ action[i][a] = reduce\ A \to \alpha$$

4. *Las entradas de la tabla de acción que queden indefinidas después de aplicado el proceso anterior corresponden a acciones de "error".*

Conflictos en Un Analizador SLR

Definición 3.23.3. *Si alguna de las entradas de la tabla resulta multievaluada, decimos que existe un conflicto y que la gramática no es* SLR*.*

1. *En tal caso, si una de las acciones es de 'reducción" y la otra es de 'desplazamiento", decimos que hay un* conflicto shift-reduce *o* conflicto de desplazamiento-reducción*.*
2. *Si las dos reglas indican una acción de reducción, decimos que tenemos un* conflicto reduce-reduce *o de* reducción-reducción*.*

Ejemplo de Cálculo de las Tablas SLR

Ejemplo 3.23.1. *Al aplicar el algoritmo 3.23.4 a la gramática 3.22.1*

1	$S \to a\ S\ b$
2	$S \to \epsilon$

partiendo del autómata finito determinista que se construyó en la figura 3.23.2 y calculando los conjuntos de primeros y siguientes

	FIRST	*FOLLOW*
S	a, ϵ	b, \$

obtenemos la siguiente tabla de acciones SLR:

	a	b	\$
0	s2	r2	r2
1			aceptar
2	s2	r2	r2
4		s5	
5		r1	r1

Las entradas denotadas con s n (s por shift) indican un desplazamiento al estado n, las denotadas con r n (r por reduce o reducción) indican una operación de reducción o antiderivación por la regla n. Las entradas vacías corresponden a acciones de error.

Las Tablas Construidas por `eyapp`

El método de análisis *LALR* usado por `eyapp` es una extensión del método SLR esbozado aqui. Supone un compromiso entre potencia (conjunto de gramáticas englobadas) y eficiencia (cantidad de memoria utilizada, tiempo de proceso). Veamos como `eyapp` aplica la construcción del subconjunto a la gramática del ejemplo 3.22.1. Para ello construimos el siguiente programa `eyapp`:

```
$ cat -n aSb.yp
     1  %%
     2  S:  # empty
     3      |   'a' S 'b'
     4  ;
     5  %%
     ......
```

y compilamos, haciendo uso de la opción `-v` para que `eyapp` produzca las tablas en el fichero `aSb.output`:

```
$ ls -l aSb.*
-rw-r--r--  1 lhp lhp  738 2004-12-19 09:52 aSb.output
-rw-r--r--  1 lhp lhp 1841 2004-12-19 09:52 aSb.pm
-rw-r--r--  1 lhp lhp  677 2004-12-19 09:46 aSb.yp
```

El contenido del fichero `aSb.output` se muestra en la tabla 3.23.2. Los números de referencia a las producciones en las acciones de reducción vienen dados por:

```
0: $start -> S $end
1: S -> /* empty */
2: S -> 'a' S 'b'
```

Observe que el final de la entrada se denota por `$end` y el marcador en un LR-item por un punto. Fíjese en el estado 2: En ese estado están también los items

`S -> . 'a' S 'b'` y `S -> .`

sin embargo no se explicitan por que se entiende que su pertenencia es consecuencia directa de aplicar la operación de clausura. Los LR items cuyo marcador no está al principio se denominan *items núcleo*.

Estado 0	Estado 1	Estado 2

`$start -> . S $end`
`'a'shift 2`
`$default`
`reduce 1 (S)`
`S go to state 1`
	`$start -> S . $end`
	`$end shift 3`
		`S -> 'a' . S 'b'`
		`'a'shift 2`
		`$default`
		`reduce 1 (S)`
		`S go to state 4`

Estado 3	Estado 4	Estado 5

`$start -> S $end .`
`$default accept`
	`S -> 'a' S . 'b'`
	`'b'shift 5`
		`S -> 'a' S 'b' .`
		`$default`
		`reduce 2 (S)`

Cuadro 3.2: Tablas generadas por `eyapp`. El estado 3 resulta de transitar con $

Puede encontrar el listado completo de las tablas en `aSb.output` en el apéndice que se encuentra en la página **??**.

Ejercicio 3.23.1. *Compare la tabla 3.23.2 resultante de aplicar* `eyapp` *con la que obtuvo en el ejemplo 3.23.1.*

3.24. Algoritmo de Análisis LR

Asi pues la tabla de transiciones del autómata nos genera dos tablas: la tabla de acciones y la de saltos. El algoritmo de análisis sintáctico *LR* en el que se basa *eyapp* utiliza una pila y dos tablas para analizar la entrada. Como se ha visto, la tabla de acciones contiene cuatro tipo de acciones:

1. Desplazar (*shift*)
2. Reducir (*reduce*)
3. Aceptar
4. Error

El algoritmo utiliza una pila en la que se guardan los estados del autómata. De este modo se evita tener que "comenzar" el procesado de la forma sentencial derecha resultante después de una reducción (antiderivación).

Algoritmo 3.24.1. *Análizador LR*

```
my $parse = shift;
my @stack;
my $s0 = $parse->startstate;
push(@stack, $s0);
my $b = $parse->yylex();
FOREVER: {
  my $s = top(0);
  my $a = $b;
  switch ($parse->action[$s->state][$a]) {
    case "shift t" :
      my $t;
      $t->{state} = t;
      $t->{attr}  = $a->{attr};
      push($t);
      $b = $parse->yylex();
      break;
    case "reduce A ->alpha" :
      my $r;
      $r->{attr} = $parse->Semantic{A ->alpha}->(top(|alpha|-1)->attr, ... , top(0)->attr);
      pop(|alpha|);  # Saquemos length(alpha) elementos de la pila
      $r->{state} = $parse->goto[top(0)][A];
      push($r);
      break;
    case "accept" : return (1);
    default : $parse->yyerror("syntax error");
  }
  redo FOREVER;
}
```

Como es habitual, $|x|$ denota la longitud de la cadena x. La función `top(k)` devuelve el elemento que ocupa la posición `k` desde el *top* de la pila (esto es, está a profundidad `k`). La función `pop(k)` extrae `k` elementos de la pila. La notación `state->attr` hace referencia al atributo asociado con cada estado. Denotamos por `$Semantic{A->alpha}` el código de la acción asociada con la regla $A \rightarrow \alpha$.

Todos los analizadores LR comparten, salvo pequeñas exepciones, el mismo algoritmo de análisis. Lo que más los diferencia es la forma en la que construyen las tablas. En `eyapp` la construcción de las tablas de *acciones* y *gotos* se realiza mediante el algoritmo *LALR*.

```
pl@nereida:~/LEyapp/examples$ use_aSb.pl
----------------------------------------
In state 0:
Stack:[0]
aabb
Need token. Got >a<
Shift and go to state 2.
----------------------------------------
In state 2:
Stack:[0,2]
Need token. Got >a<
Shift and go to state 2.
----------------------------------------
In state 2:
Stack:[0,2,2]
Need token. Got >b<
Reduce using rule 1 (S --> /* empty */): S -> epsilon
Back to state 2, then go to state 4.
----------------------------------------
In state 4:
Stack:[0,2,2,4]
Shift and go to state 5.
----------------------------------------
In state 5:
Stack:[0,2,2,4,5]
Don't need token.
Reduce using rule 2 (S --> a S b): S -> a S b
Back to state 2, then go to state 4.
----------------------------------------
In state 4:
Stack:[0,2,4]
Need token. Got >b<
Shift and go to state 5.
----------------------------------------
In state 5:
Stack:[0,2,4,5]
Don't need token.
Reduce using rule 2 (S --> a S b): S -> a S b
Back to state 0, then go to state 1.
----------------------------------------
In state 1:
Stack:[0,1]
Need token. Got ><
Shift and go to state 3.
----------------------------------------
In state 3:
Stack:[0,1,3]
Don't need token.
Accept.
```

```
pl@nereida:~/LEyapp/examples$ cat -n aSb.output
 1  Rules:
 2  ------
 3  0:      $start -> S $end
 4  1:      S -> /* empty */
 5  2:      S -> 'a' S 'b'
 6
 7  States:
 8  -------
 9  State 0:
10
11          $start -> . S $end      (Rule 0)
12
13          'a'     shift, and go to state 2
14
15          $default        reduce using rule 1 (S)
16
17          S       go to state 1
18
19  State 1:
20
21          $start -> S . $end      (Rule 0)
22
23          $end    shift, and go to state 3
24
25  State 2:
26
27          S -> 'a' . S 'b'        (Rule 2)
28
29          'a'     shift, and go to state 2
30
31          $default        reduce using rule 1 (S)
32
33          S       go to state 4
34
35  State 3:
36
37          $start -> S $end .      (Rule 0)
38
39          $default        accept
40
41  State 4:
42
43          S -> 'a' S . 'b'        (Rule 2)
44
45          'b'     shift, and go to state 5
46
47  State 5:
48
49          S -> 'a' S 'b' .        (Rule 2)
50
51          $default        reduce using rule 2 (S)
52
53
54  Summary:
55  --------
56  Number of rules         : 3
```

3.25. Precedencia y Asociatividad

Recordemos que si al construir la tabla LALR, alguna de las entradas de la tabla resulta multievaluada, decimos que existe un conflicto. Si una de las acciones es de ‘reducción” y la otra es de ‘desplazamiento”, se dice que hay un *conflicto shift-reduce* o *conflicto de desplazamiento-reducción*. Si las dos reglas indican una acción de reducción, decimos que tenemos un *conflicto reduce-reduce* o de *reducción-reducción*. En caso de que no existan indicaciones específicas *eyapp* resuelve los conflictos que aparecen en la construcción de la tabla utilizando las siguientes reglas:

1. Un conflicto *reduce-reduce* se resuelve eligiendo la producción que se listó primero en la especificación de la gramática.
2. Un conflicto *shift-reduce* se resuelve siempre en favor del *shift*

Las declaraciones de precedencia y asociatividad mediante las palabras reservadas `%left`, `%right`, `%nonassoc` se utilizan para modificar estos criterios por defecto. La declaración de `token`s mediante la palabra reservada `%token` no modifica la precedencia. Si lo hacen las declaraciones realizadas usando las palabras `left`, `right` y `nonassoc`.

1. Los *tokens* declarados en la misma línea tienen igual precedencia e igual asociatividad. La precedencia es mayor cuanto mas abajo su posición en el texto. Así, en el ejemplo de la calculadora en la sección **??**, el *token* `*` tiene mayor precedencia que `+` pero la misma que `/`.
2. La precedencia de una regla $A \rightarrow \alpha$ se define como la del terminal mas a la derecha que aparece en α. En el ejemplo, la producción

   ```
   expr : expr ’+’ expr
   ```

   tiene la precedencia del *token* `+`.
3. Para decidir en un conflicto *shift-reduce* se comparan la precedencia de la regla con la del terminal que va a ser desplazado. Si la de la regla es mayor se reduce si la del *token* es mayor, se desplaza.
4. Si en un conflicto *shift-reduce* ambos la regla y el terminal que va a ser desplazado tiene la misma precedencia *eyapp* considera la asociatividad, si es asociativa a izquierdas, reduce y si es asociativa a derechas desplaza. Si no es asociativa, genera un mensaje de error.
   Obsérvese que, en esta situación, la asociatividad de la regla y la del *token* han de ser por fuerza, las mismas. Ello es así, porque en *eyapp* los *tokens* con la misma precedencia se declaran en la misma línea y sólo se permite una declaración por línea.
5. *Por tanto es imposible declarar dos* tokens *con diferente asociatividad y la misma precedencia.*
6. Es posible modificar la precedencia “natural” de una regla, calificándola con un *token* específico. para ello se escribe a la derecha de la regla `prec token`, donde `token` es un *token* con la precedencia que deseamos. Vea el uso del *token* `dummy` en el siguiente ejercicio.

Para ilustrar las reglas anteriores usaremos el siguiente programa `eyapp`:

```
$ cat -n Precedencia.yp
%token NUMBER
%left ’@’
%right ’&’  dummy
%%
list
    :
    | list ’\n’
```

```
    | list e
    ;

e : NUMBER
  | e ’&’ e
  | e ’@’ e %prec dummy
  ;

%%
```

El código del programa cliente es el siguiente:

```
$ cat -n useprecedencia.pl
cat -n useprecedencia.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3  use Precedencia;
     4
     5  sub Error {
     6    exists $_[0]->YYData->{ERRMSG}
     7    and do {
     8      print $_[0]->YYData->{ERRMSG};
     9      delete $_[0]->YYData->{ERRMSG};
    10      return;
    11    };
    12    print "Syntax error.\n";
    13  }
    14
    15  sub Lexer {
    16    my($parser)=shift;
    17
    18    defined($parser->YYData->{INPUT})
    19    or  $parser->YYData->{INPUT} = <STDIN>
    20    or  return(’’,undef);
    21
    22    $parser->YYData->{INPUT}=~s/^[ \t]//;
    23
    24    for ($parser->YYData->{INPUT}) {
    25        s/^([0-9]+(?:\.[0-9]+)?)//
    26                and return(’NUMBER’,$1);
    27        s/^(.)//s
    28                and return($1,$1);
    29    }
    30  }
    31
    32  my $debug_level = (@ARGV)? oct(shift @ARGV): 0x1F;
    33  my $parser = Precedencia->new();
    34  $parser->YYParse( yylex => \&Lexer, yyerror => \&Error, yydebug => $debug_level );
```

Observe la llamada al analizador en la línea 34. Hemos añadido el parámetro con nombre *yydebug* con argumento `yydebug => $debug_level` (véase la sección 3.3 para ver los posibles valores de depuración).

Compilamos a continuación el módulo usando la opción `-v` para producir información sobre los conflictos y las tablas de salto y de acciones:

```
eyapp -v -m Precedencia Precedencia.yp
$ ls -ltr |tail -2
-rw-r--r--  1 lhp lhp   1628 2004-12-07 13:21 Precedencia.pm
-rw-r--r--  1 lhp lhp   1785 2004-12-07 13:21 Precedencia.output
```

La opción `-v` genera el fichero `Precedencia.output` el cual contiene información detallada sobre el autómata:

```
$ cat -n Precedencia.output
     1  Conflicts:
     2  ----------
     3  Conflict in state 8 between rule 6 and token ’@’ resolved as reduce.
     4  Conflict in state 8 between rule 6 and token ’&’ resolved as shift.
     5  Conflict in state 9 between rule 5 and token ’@’ resolved as reduce.
     6  Conflict in state 9 between rule 5 and token ’&’ resolved as shift.
     7
     8  Rules:
     9  ------
    10  0:      $start -> list $end
    11  1:      list -> /* empty */
    12  2:      list -> list ’\n’
    13  3:      list -> list e
    14  4:      e -> NUMBER
    15  5:      e -> e ’&’ e
    16  6:      e -> e ’@’ e
    17  ...
```

¿Porqué se produce un conflicto en el estado 8 entre la regla 6 (`e -> e ’@’ e`) y el terminal `’@’`?. Editando el fichero `Precedencia.output` podemos ver los contenidos del estado 8:

```
85  State 8:
86
87          e -> e . ’&’ e  (Rule 5)
88          e -> e . ’@’ e  (Rule 6)
89          e -> e ’@’ e .  (Rule 6)
90
91          ’&’     shift, and go to state 7
92
93          $default        reduce using rule 6 (e)
```

El item de la línea 88 indica que debemos desplazar, el de la línea 89 que debemos reducir por la regla 6. ¿Porqué `eyapp` resuelve el conflicto optando por reducir? ¿Que prioridad tiene la regla 6? ¿Que asociatividad tiene la regla 6? La declaración en la línea 13 modifica la precedencia y asociatividad de la regla:

```
    13    | e ’@’ e %prec dummy
```

de manera que la regla pasa a tener la precedencia y asociatividad de `dummy`. Recuerde que habíamos declarado `dummy` como asociativo a derechas:

```
     2  %left ’@’
     3  %right ’&’  dummy
```

¿Que ocurre? Que `dummy` tiene mayor prioridad que `’@’` y por tanto la regla tiene mayor prioridad que el terminal: por tanto se reduce.

¿Que ocurre cuando el terminal en conflicto es `’&’`? En ese caso la regla y el terminal tienen la misma prioridad. Se hace uso de la asociatividad a derechas que indica que el conflicto debe resolverse desplazando.

Ejercicio 3.25.1. *Explique la forma en que* eyapp *resuelve los conflictos que aparecen en el estado 9. Esta es la información sobre el estado 9:*

```
State 9:

e -> e . ’&’ e (Rule 5)
e -> e ’&’ e . (Rule 5)
e -> e . ’@’ e (Rule 6)
’&’shift, and go to state 7
$default reduce using rule 5 (e)
```

Veamos un ejemplo de ejecución:

```
$ ./useprecedencia.pl
----------------------------------------
In state 0:
Stack:[0]
Don’t need token.
Reduce using rule 1 (list,0): Back to state 0, then go to state 1.
```

Lo primero que ocurre es una reducción por la regla en la que `list` produce vacío. Si miramos el estado 0 del autómata vemos que contiene:

```
State 0:

  $start -> . list $end (Rule 0)

  $default  reduce using rule 1 (list)

  list  go to state 1
```

A continuación se transita desde 0 con `list` y se consume el primer terminal:

```
----------------------------------------
In state 1:
Stack:[0,1]
2@3@4
Need token. Got >NUMBER<
Shift and go to state 5.
----------------------------------------
In state 5:
Stack:[0,1,5]
Don’t need token.
Reduce using rule 4 (e,1): Back to state 1, then go to state 2.
----------------------------------------
```

En el estado 5 se reduce por la regla `e -> NUMBER`. Esto hace que se retire el estado 5 de la pila y se transite desde el estado 1 viendo el símbolo `e`:

```
In state 2:
Stack:[0,1,2]
Need token. Got >@<
Shift and go to state 6.
----------------------------------------
In state 6:
Stack:[0,1,2,6]
```

```
Need token. Got >NUMBER<
Shift and go to state 5.
----------------------------------------
In state 5:
Stack:[0,1,2,6,5]
Don't need token.
Reduce using rule 4 (e,1): Back to state 6, then go to state 8.
----------------------------------------
In state 8:
Stack:[0,1,2,6,8]
Need token. Got >@<
Reduce using rule 6 (e,3): Back to state 1, then go to state 2.
----------------------------------------
...
Accept.
```

Obsérvese la resolución del conflicto en el estado 8

La presencia de conflictos, aunque no siempre, en muchos casos es debida a la introducción de ambiguedad en la gramática. Si el conflicto es de desplazamiento-reducción se puede resolver explicitando alguna regla que rompa la ambiguedad. Los conflictos de reducción-reducción suelen producirse por un diseño erróneo de la gramática. En tales casos, suele ser mas adecuado modificar la gramática.

3.26. Acciones en Medio de una Regla

A veces necesitamos insertar una acción en medio de una regla. Una acción en medio de una regla puede hacer referencia a los atributos de los símbolos que la preceden (usando `$n`), pero no a los que le siguen.

Cuando se inserta una acción $\{action_1\}$ para su ejecución en medio de una regla $A \rightarrow \alpha\beta$:

$$A \rightarrow \alpha \{action_1\} \beta \{action_2\}$$

`eyapp` crea una variable sintáctica temporal T e introduce una nueva regla:

1. $A \rightarrow \alpha T \beta \{action_2\}$
2. $T \rightarrow \epsilon \{action_1\}$

Las acciones en mitad de una regla cuentan como un símbolo mas en la parte derecha de la regla. Asi pues, en una acción posterior en la regla, se deberán referenciar los atributos de los símbolos, teniendo en cuenta este hecho.

Las acciones en mitad de la regla pueden tener un atributo. Las acciones posteriores en la regla se referirán a él como `$_[n]`, siendo `n` su número de orden en la parte derecha.

Observe que la existencia de acciones intermedias implica que la gramática inicial es modificada. La introducción de las nuevas reglas puede dar lugar a ambiguedades y/o conflictos. Es responsabilidad del programador eliminarlos. Por ejemplo, dada la gramática:

```
cs : '{' decs ss '}' | '{' ss '}' ;
```

Si la modificamos como sigue:

```
cs : { decl(); } '{' decs ss '}'
   | '{' ss '}'
   ;
```

habremos introducido un conflicto.

Ejercicio 3.26.1. *Explique las razones por las cuales la introducción de la acción que prefija la primera regla da lugar a conflictos*

Ejercicio 3.26.2. *El conflicto no se arregla haciendo que la acción que precede sea la misma:*

```
cs : { decl(); } '{' decs ss '}'
   | { decl(); } '{' ss '}'
   ;
```

Explique donde está el fallo de esta propuesta

Ejercicio 3.26.3. *¿Se arregla el conflicto anterior usando esta otra alternativa?*

```
cs :  tp '{' decs ss '}'
   |  tp { decl(); } '{' ss '}'
   ;

tp : /* empty */ { decl(); }
   ;
```

3.27. Manejo en `eyapp` de Atributos Heredados

Supongamos que `eyapp` esta inmerso en la construcción de la antiderivación a derechas y que la forma sentencial derecha en ese momento es:

$$X_m \ldots X_1 X_0 Y_1 \ldots Y_n a_1 \ldots a_0$$

y que el mango es $B \rightarrow Y_1 \ldots Y_n$ y en la entrada quedan por procesar $a_1 \ldots a_0$.

Es posible acceder en `eyapp` a los valores de los atributos de los estados en la pila del analizador que se encuentran “por debajo” o si se quiere “a la izquierda” de los estados asociados con la regla por la que se reduce. Para ello se usa una llamada al método `YYSemval`. La llamada es de la forma `$_[0]->YYSemval( index )`, donde `index` es un entero. Cuando se usan los valores `1` ...`n` devuelve lo mismo que `$_[1]`, ...`$_[n]`. Esto es `$_[1]` es el atributo asociado con Y_1 y `$_[n]` es el atributo asociado con Y_n. Cuando se usa con el valor 0 devolverá el valor del atributo asociado con el símbolo que esta a la izquierda del mango actual, esto es el atributo asociado con X_0, si se llama con -1 el que está dos unidades a la izquierda de la variable actual, esto es, el asociado con X_1 etc. Así `$_[-m]` denota el atributo de X_m.

Esta forma de acceder a los atributos es especialmente útil cuando se trabaja con *atributos heredados*. Esto es, cuando un atributo de un nodo del árbol sintáctico se computa en términos de valores de atributos de su padre y/o sus hermanos. Ejemplos de atributos heredados son la clase y tipo en la declaración de variables. Supongamos que tenemos el siguiente *esquema de traducción* para calcular la clase (C) y tipo (T) en las declaraciones (D) de listas (L) de identificadores:

1	D →	C T `{ $L{c} = $C{c}; $L{t} = $T{t} }` L
2	C →	global `{ $C{c} = "global" }`
3	C →	local `{ $C{c} = "local" }`
4	T →	integer `{ $T{t} = "integer" }`
5	T →	float `{ $T{t} = "float" }`
6	L →	`{ $L_1{t} = $L{t}; $L_1{c} = $L{c}; }` L_1 ’,’
7		id `{ set_class($id{v}, $L{c}); set_type($id{v}, $L{t}); }`
8	L →	id `{ set_class($id{v}, $L{c}); set_type($id{v}, $L{t}); }`

Los atributos `c` y `t` denotan respectivamente la clase y el tipo.

Ejercicio 3.27.1. *Evalúe el esquema de traducción para la entrada* `global float x,y`. *Represente el árbol de análisis, las acciones incrustadas y determine el orden de ejecución.*

Olvide por un momento la notación usada en las acciones y suponga que se tratara de acciones `eyapp`. *¿En que orden construye* `eyapp` *el árbol y en que orden ejecutará las acciones?*

Observe que la simple razón por la cual la acción intermedia `{ $L1{t} = $L{t}; $L1{c} = $L{c}; }` no puede ser ejecutada en un programa `eyapp` es porque el nodo del árbol sintáctico asociado con `L1` no existe en el momento en el que la acción es ejecutada. No es posible guardar algo en `L1{c}` ya que no hay nodo asociado con `L1`. La situación es distinta si se esta trabajando con un esquema de traducción ya que estos primero construyen el árbol y luego lo recorren.

A la hora de transformar este esquema de traducción en un programa `eyapp` es importante darse cuenta que en cualquier derivación a derechas desde D, cuando se reduce por una de las reglas

$$\mathrm{L} \rightarrow \mathrm{id} \mid \mathrm{L}_1 \text{ ',' } \mathrm{id}$$

el símbolo a la izquierda de L es T y el que esta a la izquierda de T es C. Considere, por ejemplo la derivación a derechas:

$$\begin{gathered}\mathrm{D} \Longrightarrow \mathrm{C\ T\ L} \Longrightarrow \mathrm{C\ T\ L,\ id} \Longrightarrow \mathrm{C\ T\ L,\ id,\ id} \Longrightarrow \mathrm{C\ T\ id,\ id,\ id} \Longrightarrow \\ \Longrightarrow \mathrm{C\ float\ id,\ id,\ id} \Longrightarrow \mathrm{local\ float\ id,\ id,\ id}\end{gathered}$$

Observe que el orden de recorrido de `eyapp` es:

$$\begin{gathered}\mathrm{local\ float\ id,\ id,\ id} \Longleftarrow \mathrm{C\ float\ id,\ id} \Longleftarrow \mathrm{C\ T\ id,\ id,\ id} \Longleftarrow \\ \Longleftarrow \mathrm{C\ T\ L,\ id,\ id} \Longleftarrow \mathrm{C\ T\ L,\ id} \Longleftarrow \mathrm{C\ T\ L} \Longleftarrow \mathrm{D}\end{gathered}$$

en la antiderivación, cuando el mango es una de las dos reglas para listas de identificadores, L → id y L → L, id es decir durante las tres ultimas antiderivaciones:

$$\mathrm{C\ T\ L,\ id,\ id} \Longleftarrow \mathrm{C\ T\ L,\ id} \Longleftarrow \mathrm{C\ T\ L} \Longleftarrow \mathrm{D}$$

las variables a la izquierda del mango son T y C. Esto ocurre siempre. Estas observaciones nos conducen al siguiente programa `eyapp`:

```
$ cat -n Inherited.yp
%token FLOAT INTEGER
%token GLOBAL
%token LOCAL
%token NAME

%%
declarationlist
  : # vacio
  | declaration ';' declarationlist
  ;

declaration
  : class type namelist { ; }
  ;

class
  : GLOBAL
  | LOCAL
  ;

type
  : FLOAT
  | INTEGER
  ;

```

```
namelist
  : NAME
     { printf("%s de clase %s, tipo %s\n",
             $_[1], $_[0]->YYSemval(-1),$_[0]->YYSemval(0)); }
  | namelist ',' NAME
      { printf("%s de clase %s, tipo %s\n",
               $_[3], $_[0]->YYSemval(-1),$_[0]->YYSemval(0)); }
  ;
%%
```

A continuación escribimos el programa que usa el módulo generado por `eyapp`:

```
$ cat -n useinherited.pl
#!/usr/bin/perl -w
use strict;
use Inherited;

sub Error {
  exists $_[0]->YYData->{ERRMSG}
  and do {
    print $_[0]->YYData->{ERRMSG};
    delete $_[0]->YYData->{ERRMSG};
    return;
  };
  print "Error sintáctico\n";
}

{ # hagamos una clausura con la entrada
  my $input;
  local $/ = undef;
  print "Entrada (En Unix, presione CTRL-D para terminar):\n";
  $input = <stdin>;

  sub scanner {

    { # Con el redo del final hacemos un bucle "infinito"
      if ($input =~ m|\G\s*INTEGER\b|igc) {
        return ('INTEGER', 'INTEGER');
      }
      elsif ($input =~ m|\G\s*FLOAT\b|igc) {
        return ('FLOAT', 'FLOAT');
      }
      elsif ($input =~ m|\G\s*LOCAL\b|igc) {
        return ('LOCAL', 'LOCAL');
      }
      elsif ($input =~ m|\G\s*GLOBAL\b|igc) {
        return ('GLOBAL', 'GLOBAL');
      }
      elsif ($input =~ m|\G\s*([a-z_]\w*)\b|igc) {
        return ('NAME', $1);
      }
      elsif ($input =~ m/\G\s*([,;])/gc) {
        return ($1, $1);
      }
```

```
42        elsif ($input =~ m/\G\s*(.)/gc) {
43          die "Caracter invalido: $1\n";
44        }
45        else {
46          return (’’, undef); # end of file
47        }
48        redo;
49      }
50    }
51  }
52
53  my $debug_level = (@ARGV)? oct(shift @ARGV): 0x1F;
54  my $parser = Inherited->new();
55  $parser->YYParse( yylex => \&scanner, yyerror => \&Error, yydebug => $debug_level );
```

En las líneas de la 15 a la 51 esta nuestro analizador léxico. La entrada se lee en una variable local cuyo valor permanece entre llamadas: hemos creado una clausura con la variable `$input` (véase la sección [2] para mas detalles sobre el uso de clausuras en Perl). Aunque la variable `$input` queda inaccesible desde fuera de la clausura, persiste entre llamadas como consecuencia de que la subrutina `scanner` la utiliza.

A continuación sigue un ejemplo de ejecución:

```
$ ./useinherited.pl 0
Entrada (En Unix, presione CTRL-D para terminar):
global integer x, y, z;
local float a,b;
x de clase GLOBAL, tipo INTEGER
y de clase GLOBAL, tipo INTEGER
z de clase GLOBAL, tipo INTEGER
a de clase LOCAL, tipo FLOAT
b de clase LOCAL, tipo FLOAT
```

3.28. Acciones en Medio de una Regla y Atributos Heredados

La estrategia utilizada en la sección 3.27 funciona si podemos predecir la posición del atributo en la pila del analizador. En el ejemplo anterior los atributos clase y tipo estaban siempre, cualquiera que fuera la derivación a derechas, en las posiciones 0 y -1. Esto no siempre es asi. Consideremos la siguiente *definición dirigida por la sintáxis*:

S → a A C	`$C{i} = $A{s}`
S → b A B C	`$C{i} = $A{s}`
C → c	`$C{s} = $C{i}`
A → a	`$A{s} = "a"`
B → b	`$B{s} = "b"`

Ejercicio 3.28.1. *Determine un orden correcto de evaluación de la anterior definición dirigida por la sintáxis para la entrada* `b a b c`*.*

C hereda el atributo sintetizado de A. El problema es que, en la pila del analizador el atributo `$A{s}` puede estar en la posición 0 o -1 dependiendo de si la regla por la que se derivó fué S → a A C o bien S → b A B C. La solución a este tipo de problemas consiste en insertar acciones intermedias de copia del atributo de manera que se garantize que el atributo de interés está siempre a una distancia fija. Esto es, se inserta una variable sintáctica intermedia auxiliar M la cual deriva a vacío y que tiene como acción asociada una regla de copia:

S → a A C	`$C{i} = $A{s}`
S → b A B M C	`$M{i} = $A{s}; $C{i} = $M{s}`
C → c	`$C{s} = $C{i}`
A → a	`$A{s} = "a"`
B → b	`$B{s} = "b"`
M → ϵ	`$M{s} = $M{i}`

El nuevo esquema de traducción puede ser implantado mediante un programa `eyapp`:

```
$ cat -n Inherited2.yp
     1  %%
     2  S : 'a' A C
     3    | 'b' A B  { $_[2]; } C
     4    ;
     5
     6  C : 'c' { print "Valor: ",$_[0]->YYSemval(0),"\n"; $_[0]->YYSemval(0) }
     7    ;
     8
     9  A : 'a' { 'a' }
    10    ;
    11
    12  B : 'b' { 'b' }
    13    ;
    14
    15  %%
```

La ejecución muestra como se ha propagado el valor del atributo:

```
$ ./useinherited2.pl '0x04'
Entrada (En Unix, presione CTRL-D para terminar):
b a b c
Shift 2.  Shift 6.
Reduce using rule 5 (A,1): Back to state 2, then state 5.
Shift 8.
Reduce 6 (B,1): Back to state 5, then state 9.
Reduce 2 (@1-3,0): Back to state 9, then state 12.
```

En este momento se esta ejecutando la acción intermedia. Lo podemos comprobar revisando el fichero `Inherited2.output` que fué generado usando la opción `-v` al llamar a `eyapp`. La regla 2 por la que se reduce es la asociada con la acción intermedia:

```
$ cat -n Inherited2.output
     1  Rules:
     2  ------
     3  0:      $start -> S $end
     4  1:      S -> 'a' A C
     5  2:      @1-3 -> /* empty */
     6  3:      S -> 'b' A B @1-3 C
     7  4:      C -> 'c'
     8  5:      A -> 'a'
     9  6:      B -> 'b'
        ...
```

Obsérvese la notación usada por `eyapp` para la *acción en medio de la regla*: `@1-3`. Continuamos con la antiderivación:

```
Shift 10.
Reduce 4 (C,1):
Valor: a
Back to state 12, then 13.
Reduce using rule 3 (S,5): Back to state 0, then state 1.
Shift 4.
Accept.
```

El método puede ser generalizado a casos en los que el atributo de interés este a diferentes distancias en diferentes reglas sin mas que introducir las correspondientes acciones intermedias de copia.

Capítulo 4

Análisis Sintáctico con yacc

4.1. Introducción a yacc

Veamos un ejemplo sencillo de analizador sintáctico escrito en `yacc`. La gramática se especifica entre las dos líneas de `%%`. Por defecto, el símbolo de arranque es el primero que aparece, en este caso `list`. En `bison` es posible hacer que otro variable lo sea utilizando la declaración `%start`:

Ejemplo: La Calculadora en yacc

Programa 4.1.1. *Calculadora elemental. Análizador sintáctico.*

```
nereida:~/src/precedencia/hoc1> cat -n hoc1.y
%{
/* File: /home/pl/src/precedencia/hoc1/hoc1.y */
#define YYSTYPE double
#include <stdio.h>
%}
%token NUMBER
%left ’+’ ’-’
%left ’*’ ’/’
%%
list
    :
    | list ’\n’
    | list expr   { printf("%.8g\n",$2);}
    ;

expr
    : NUMBER { $$ = $1;}
    | expr ’+’ expr {$$ = $1 + $3;}
    | expr ’-’ expr {$$ = $1 - $3;}
    | expr ’*’ expr {$$ = $1 * $3;}
    | expr ’/’ expr {$$ = $1 / $3;}
    ;

%%

extern FILE * yyin;

main(int argc, char **argv) {
  if (argc > 1) yyin = fopen(argv[1],"r");
  yydebug = 1;
```

```
31    yyparse();
32  }
33
34  yyerror(char *s) {
35    printf("%s\n",s);
36  }
```

La macro `YYSTYPE` (línea 3) contiene el tipo del valor semántico. Si no se declara se asume `int`.

El fichero `yyin` (líneas 26 y 29) es definido en el fichero conteniendo el analizador léxico `lex.yy.c`. Refiere al fichero de entrada conteniendo el texto a analizar.

Al poner la variable `yydebug` a 1 activamos el modo depuración. Para que la depuración se haga efectiva es necesario definir además la macro `YYDEBUG`.

El analizador sintáctico proveido por `yacc` se llama `yyparse` (línea 31). Por defecto su declaración es `int yyparse ()`

El Analizador Léxico

Programa 4.1.2. *Calculadora elemental. Analizador léxico:*

```
nereida:~/src/precedencia/hoc1> cat -n hoc1.l
 1  %{
 2  #include "y.tab.h"
 3  extern YYSTYPE yylval;
 4  %}
 5  number [0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?
 6  %%
 7  {number} { yylval = atof(yytext); return NUMBER; }
 8  .|\n     { return yytext[0];}
 9  %%
10  int yywrap() { return 1; }
```

Compilación

Al compilar el program `yacc` con la opción `-d` se produce además del fichero `y.tab.c` conteniendo el analizador sintáctico un fichero adicional de cabecera `y.tab.h` conteniendo las definiciones de los terminales:

```
nereida:~/src/precedencia/hoc1> yacc -d -v hoc1.y
nereida:~/src/precedencia/hoc1> ls -lt | head -4
total 200
-rw-rw----  1 pl users    2857 2007-01-18 10:26 y.output
-rw-rw----  1 pl users   35936 2007-01-18 10:26 y.tab.c
-rw-rw----  1 pl users    1638 2007-01-18 10:26 y.tab.h
nereida:~/src/precedencia/hoc1> sed -ne '27,48p' y.tab.h | cat -n
 1  #ifndef YYTOKENTYPE
 2  # define YYTOKENTYPE
 3     /* Put the tokens into the symbol table, so that GDB and other debuggers
 4        know about them.  */
 5     enum yytokentype {
 6       NUMBER = 258
 7     };
 8  #endif
 9  /* Tokens.  */
10  #define NUMBER 258
..  ..................................................
15  #if ! defined (YYSTYPE) && ! defined (YYSTYPE_IS_DECLARED)
```

```
16  typedef int YYSTYPE;
17  # define yystype YYSTYPE /* obsolescent; will be withdrawn */
18  # define YYSTYPE_IS_DECLARED 1
19  # define YYSTYPE_IS_TRIVIAL 1
20  #endif
21
22  extern YYSTYPE yylval;
```

La variable `yylval` (líneas 3 y 7 del listado 4.1.2) es declarada por el analizador sintáctico y usada por el analizador léxico. El analizador léxico deja en la misma el valor semántico asociado con el token actual.

Makefile

Para compilar todo el proyecto usaremos el siguiente fichero Makefile:

Programa 4.1.3. *Calculadora elemental. Makefile:*

```
> cat Makefile
hoc1: y.tab.c lex.yy.c
        gcc -DYYDEBUG=1 -g -o hoc1 y.tab.c lex.yy.c
y.tab.c y.tab.h: hoc1.y
        yacc -d -v hoc1.y
lex.yy.c: hoc1.l y.tab.h
        flex -l hoc1.l
clean:
        - rm -f y.tab.c lex.yy.c *.o core hoc1
```

Ejecución

Ejecución 4.1.1. *Para saber que esta haciendo el analizador, insertamos una asignación:* `yydebug = 1;` *justo antes de la llamada a* `yyparse()` *y ejecutamos el programa resultante:*

```
$ hoc1
yydebug: state 0, reducing by rule 1 (list :)
yydebug: after reduction, shifting from state 0 to state 1
2.5+3.5+1
```

Introducimos la expresión `2.5+3.5+1`*. Antes que incluso ocurra la entrada, el algoritmo LR reduce por la regla List* $\rightarrow \epsilon$*.*

```
yydebug: state 1, reading 257 (NUMBER)
yydebug: state 1, shifting to state 2
yydebug: state 2, reducing by rule 4 (expr : NUMBER)
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, reading 43 ('+')
yydebug: state 4, shifting to state 5
yydebug: state 5, reading 257 (NUMBER)
yydebug: state 5, shifting to state 2
yydebug: state 2, reducing by rule 4 (expr : NUMBER)
yydebug: after reduction, shifting from state 5 to state 6
yydebug: state 6, reducing by rule 5 (expr : expr '+' expr)
```

Observe como la declaración de la asociatividad a izquierdas `%left '+'` *se traduce en la reducción por la regla 5.*

```
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, reading 43 ('+')
yydebug: state 4, shifting to state 5
yydebug: state 5, reading 257 (NUMBER)
yydebug: state 5, shifting to state 2
yydebug: state 2, reducing by rule 4 (expr : NUMBER)
yydebug: after reduction, shifting from state 5 to state 6
yydebug: state 6, reducing by rule 5 (expr : expr '+' expr)
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, reading 10 ('\n')
yydebug: state 4, reducing by rule 3 (list : list expr)
7
```

La reducción por la regla list → list expr produce la ejecución del `printf("%.8g\n",$2);` *asociado con la regla y la salida del valor* 7 *que constituye el atributo de expr.*

```
yydebug: after reduction, shifting from state 0 to state 1
yydebug: state 1, shifting to state 3
yydebug: state 3, reducing by rule 2 (list : list '\n')
yydebug: after reduction, shifting from state 0 to state 1
yydebug: state 1, reading 0 (end-of-file)
$
```

En Unix la combinación de teclas CTRL-D nos permite generar el final de fichero.

4.2. Precedencia y Asociatividad

En caso de que no existan indicaciones específicas *yacc* resuelve los conflictos que aparecen en la construcción de la tabla utilizando las siguientes reglas:

1. Un conflicto *reduce-reduce* se resuelve eligiendo la producción que se listó primero en la especificación de la gramática.
2. Un conflicto *shift-reduce* se resuelve siempre en favor del *shift*

La precedencia se utiliza para modificar estos criterios. Para ello se define:

1. La precedencia de los *tokens* es determinada según el orden de declaración. La declaración de *tokens* mediante la palabra reservada `token` no modifica la precedencia. Si lo hacen las declaraciones realizadas usando las palabras `left`, `right` y `nonassoc`. Los *tokens* declarados en la misma línea tienen igual precedencia. La precedencia es mayor cuanto mas abajo en el texto. Así, en el ejemplo que sigue, el *token* `*` tiene mayor precedencia que `+` pero la misma que `/`.
2. La precedencia de una regla $A \rightarrow \alpha$ se define como la del terminal mas a la derecha que aparece en α. En el ejemplo, la producción

   ```
   expr : expr '+' expr
   ```

   tiene la precedencia del *token* `+`.
3. Para decidir en un conflicto *shift-reduce* se comparan la precedencia de la regla con la del terminal que va a ser desplazado. Si la de la regla es mayor se reduce si la del *token* es mayor, se desplaza.
4. Si en un conflicto *shift-reduce* ambos la regla y el terminal que va a ser desplazado tiene la misma precedencia *yacc* considera la asociatividad, si es asociativa a izquierdas, reduce y si es asociativa a derechas desplaza. Si no es asociativa, genera un mensaje de error.

   Obsérvese que, en esta situación, la asociatividad de la regla y la del *token* han de ser por fuerza, las mismas. Ello es así, porque en *yacc* los *tokens* con la misma precedencia se declaran en la misma línea.

   Por tanto es imposible declarar dos tokens *con diferente asociatividad y la misma precedencia.*

5. Es posible modificar la precedencia “natural” de una regla, calificándola con un *token* específico. para ello se escribe a la derecha de la regla `prec token`, donde `token` es un *token* con la precedencia que deseamos. Vea el uso del *token* `dummy` en el siguiente ejercicio.

Programa 4.2.1. *Este programa muestra el manejo de las reglas de precedencia.*

```
%{
#define YYSTYPE double
#include <stdio.h>
%}
%token NUMBER
%left ’@’
%right ’&’  dummy
%%
list
    :
    | list ’\n’
    | list e
    ;

e : NUMBER
  | e ’&’ e
  | e ’@’ e %prec dummy
    ;

%%
extern FILE * yyin;

main(int argc, char **argv) {
  if (argc > 1) yyin = fopen(argv[1],"r");
  yydebug = 1;
  yyparse();
}

yyerror(char *s) {
  printf("%s\n",s);
}
```

Ejercicio 4.2.1. *Dado el programa* `yacc` *4.2.1 Responda a las siguientes cuestiones:*

1. *Construya las tablas SLR de acciones y* gotos.
2. *Determine el árbol construido para las frases:* `4@3@5`*,* `4&3&5`*,* `4@3&5`*,* `4&3@5`*.*
3. *¿Cuál es la asociatividad final de la regla* `e : e ’@’ e`*?*

Listado 4.2.1. *Fichero* y.output*:*

```
   0  $accept : list $end
```

```
1  list :
2       | list '\n'
3       | list e

4  e : NUMBER
5    | e '&' e
6    | e '@' e
```

```
state 0
$accept : . list $end  (0)
list : .  (1)

.  reduce 1

list  goto 1


state 1
$accept : list . $end  (0)
list : list . '\n'  (2)
list : list . e  (3)

$end  accept
NUMBER  shift 2
'\n'  shift 3
.  error

e  goto 4


state 2
e : NUMBER .  (4)

.  reduce 4


state 3
list : list '\n' .  (2)

.  reduce 2


state 4
list : list e .  (3)
e : e . '&' e  (5)
e : e . '@' e  (6)

'@'  shift 5
'&'  shift 6
$end  reduce 3
NUMBER  reduce 3
'\n'  reduce 3
```

```
state 5
e : e '@' . e  (6)

NUMBER  shift 2
.  error

e  goto 7


state 6
e : e '&' . e  (5)

NUMBER  shift 2
.  error

e  goto 8


state 7
e : e . '&' e  (5)
e : e . '@' e  (6)
e : e '@' e .  (6)

'&'  shift 6
$end  reduce 6
NUMBER  reduce 6
'@'  reduce 6
'\n'  reduce 6

state 8
e : e . '&' e  (5)
e : e '&' e .  (5)
e : e . '@' e  (6)

'&'  shift 6
$end  reduce 5
NUMBER  reduce 5
'@'  reduce 5
'\n'  reduce 5
```

```
7 terminals, 3 nonterminals
7 grammar rules, 9 states
```

Ejemplo 4.2.1. *Contrasta tu respuesta con la traza seguida por el programa anterior ante la entrada* `1@2&3`*, al establecer la variable* `yydebug = 1` *y definir la macro* `YYDEBUG`*:*

Ejecución 4.2.1. `$ hocprec`

```
yydebug: state 0, reducing by rule 1 (list :)
yydebug: after reduction, shifting from state 0 to state 1
1@2&3
yydebug: state 1, reading 257 (NUMBER)
yydebug: state 1, shifting to state 2
yydebug: state 2, reducing by rule 4 (e : NUMBER)
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, reading 64 ('@')
yydebug: state 4, shifting to state 5
yydebug: state 5, reading 257 (NUMBER)
yydebug: state 5, shifting to state 2
yydebug: state 2, reducing by rule 4 (e : NUMBER)
yydebug: after reduction, shifting from state 5 to state 7
yydebug: state 7, reading 38 ('&')
yydebug: state 7, shifting to state 6
```

¿Por que se desplaza? ¿No va eso en contra de la declaración `%left '@'`*?. ¿O quizá es porque la precedencia de* `@` *es menor que la de* `&`*? La respuesta es que la precedencia asignada por la declaración*

`e : e '@' e %prec dummy`

cambio la asociatividad de la regla. Ahora la regla se "enfrenta" a un token*,* `&` *con su misma precedencia. Al pasar a ser asociativa a derechas (debido a que* `dummy` *lo es), se debe desplazar y no reducir.*

Ejemplo 4.2.2. *Otra ejecución, esta vez con entrada* `1&2@3`*. Compara tus predicciones con los resultados.*

Ejecución 4.2.2. `$ hocprec`

```
yydebug: state 0, reducing by rule 1 (list :)
yydebug: after reduction, shifting from state 0 to state 1
1&2@3
yydebug: state 1, reading 257 (NUMBER)
yydebug: state 1, shifting to state 2
yydebug: state 2, reducing by rule 4 (e : NUMBER)
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, reading 38 ('&')
yydebug: state 4, shifting to state 6
yydebug: state 6, reading 257 (NUMBER)
yydebug: state 6, shifting to state 2
yydebug: state 2, reducing by rule 4 (e : NUMBER)
yydebug: after reduction, shifting from state 6 to state 8
yydebug: state 8, reading 64 ('@')
yydebug: state 8, reducing by rule 5 (e : e '&' e)
```

En este caso se comparan la producción :

`e` → `e & e`

con el token `@`*. La regla tiene mayor precedencia que el* token*,dado que la precedencia de la regla es la de* `&`*.*

4.3. Uso de union y type

En general, los atributos asociados con las diferentes variables sintácticas y terminales tendrán tipos de datos distintos. Para ello, `yacc` provee la declaración `%union`.

La declaración `%union` especifica la colección de posibles tipos de datos de `yylval` y de los atributos `$1`, `$2`, ...

He aqui un ejemplo:

```
%union {
  double val;
  int index;
}
```

Esto dice que los dos tipos de alternativas son `double` y `int` . Se les han dado los nombres `val` y `index`.

```
%token <val>   NUMBER
%token <index> VAR
%type  <val>   expr
%right '='
%left '+' '-'
%left '*' '/'
```

Estos nombres `<val>` e `<index>` se utilizan en las declaraciones de `%token` y `%type` para definir el tipo del correspondiente atributo asociado.

Dentro de las acciones, se puede especificar el tipo de un símbolo insertando `<tipo>` despues del `$` que referencia al atributo. En el ejemplo anterior podríamos escribir `$<val>1` para indicar que manipulamos el atributo del primer símbolo de la parte derecha de la regla como si fuera un `double`.

La información que provee la declaración `%union` es utilizada por `yacc` para realizar la sustitución de las referencias textuales/formales (`$$`, `$1`, ...`$`$|\alpha|$) a los atributos de los símbolos que conforman la regla (que pueden verse como parámetros formales de las acciones) por las referencias a las zonas de memoria en las que se guardan (que están asociadas con los correspondientes estados de la pila) cuando tiene lugar la reducción en el algoritmo de análisis LR:

```
case "reduce A $\rightarrow \alpha$" :
  execute("reduce A $\rightarrow \alpha$", top($|\alpha|$-1), $\cdots$ , top(0));
  pop($|\alpha|$);
  push(goto[top(0)][A];
  break;
```

Asi, `yacc` es capaz de insertar el código de ahormado de tipos correcto. Ello puede hacerse porque se conocen los tipos asociados con los símbolos en la parte derecha de una regla, ya que han sido proveídos en las declaraciones `%union`, `%token` y `%type`.

4.4. Acciones en medio de una regla

A veces necesitamos insertar una acción en medio de una regla. Una acción en medio de una regla puede hacer referencia a los atributos de los símbolos que la preceden (usando `$n`), pero no a los que le siguen.

Cuando se inserta una acción $\{action_1\}$ para su ejecución en medio de una regla $A \rightarrow \alpha\beta$:

$$A \rightarrow \alpha \, \{action_1\} \, \beta \, \{action_2\}$$

`yacc` crea una variable sintáctica temporal T e introduce una nueva regla:

1. $A \rightarrow \alpha T \beta \, \{action_2\}$

2. $T \rightarrow \epsilon \{action_1\}$

Las acciones en mitad de una regla cuentan como un símbolo mas en la parte derecha de la regla. Asi pues, en una acción posterior en la regla, se deberán referenciar los atributos de los símbolos, teniendo en cuenta este hecho.

Las acciones en mitad de la regla pueden tener un atributo. La acción en cuestión puede hacer referencia a ese atributo mediante `$$`, y las acciones posteriores en la regla se referirán a él como `$n`, siendo `n` su número de orden en la parte derecha. Dado que no existe un símbolo explícito que identifique a la acción, no hay manera de que el programador declare su tipo. Sin embargo, es posible utilizar la construcción `$<valtipo>#` para especificar la forma en la que queremos manipular su atributo.

Na hay forma de darle, en una acción a media regla, un valor al atributo asociado con la variable en la izquierda de la regla de producción (ya que `$$` se refiere al atributo de la variable temporal utilizada para introducir la acción a media regla).

Programa 4.4.1. *El siguiente programa ilustra el uso de* `%union` *y de las acciones en medio de una regla.*

```
%{
#include <string.h>
char buffer[256];
#define YYDEBUG 1
%}
%union {
  char tA;
  char *tx;
}
%token <tA> A
%type <tx> x
%%
s : x { *$1 = '\0'; printf("%s\n",buffer); } '\n' s
  |
  ;

x : A { $$ = buffer + sprintf(buffer,"%c",$1); }
  | A { $<tx>$ = strdup("**"); } x
    { $$ = $3 + sprintf($3,"%s%c",$<tx>2,$1); free($2); }
  ;

%%

main() {
  yydebug=1;
  yyparse();
}

yyerror(char *s) {
  printf("%s\n",s);
}
```

Programa 4.4.2. *El analizador léxico utilizado es el siguiente:*

```
%{
#include "y.tab.h"
%}
%%
```

```
[\t ]+
[a-zA-Z0-9]    { yylval.tA = yytext[0]; return A; }
(.|\n)         { return yytext[0]; }
%%
yywrap() { return 1; }
```

Ejemplo 4.4.1. *Considere el programa* `yacc` *4.4.1. ¿Cuál es la salida para la entrada ABC?*

La gramática inicial se ve aumentada con dos nuevas variables sintácticas temporales y dos reglas $t_1 \rightarrow \epsilon$ *y* $t_2 \rightarrow \epsilon$*. Además las reglas correspondientes pasan a ser:* $s \rightarrow xt_1s$ *y* $x \rightarrow At_2x$*. El análisis de la entrada ABC nos produce el siguiente árbol anotado:*

Ejecución 4.4.1. *Observe la salida de la ejecución del programa 4.4.1. La variable "temporal" creada por* `yacc` *para la acción en medio de la regla*

```
s →  x { *$1 = ’\0’; printf("%s\n",buffer); } ’\n’ s
```

se denota por `$$1`*. La asociada con la acción en medio de la regla*

```
x  →  A { $<tx>$ = strdup("**"); } x
```

se denota `$$2`*.*

```
$ yacc -d -v media4.y ; flex -l medial.l ; gcc -g y.tab.c lex.yy.c
$ a.out
ABC
yydebug: state 0, reading 257 (A)
yydebug: state 0, shifting to state 1
yydebug: state 1, reading 257 (A)
yydebug: state 1, reducing by rule 5 ($$2 :)
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, shifting to state 1
yydebug: state 1, reading 257 (A)
yydebug: state 1, reducing by rule 5 ($$2 :)
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, shifting to state 1
yydebug: state 1, reading 10 (’\n’)
yydebug: state 1, reducing by rule 4 (x : A)
yydebug: after reduction, shifting from state 4 to state 6
yydebug: state 6, reducing by rule 6 (x : A $$2 x)
yydebug: after reduction, shifting from state 4 to state 6
yydebug: state 6, reducing by rule 6 (x : A $$2 x)
yydebug: after reduction, shifting from state 0 to state 3
yydebug: state 3, reducing by rule 1 ($$1 :)
C**B**A
yydebug: after reduction, shifting from state 3 to state 5
yydebug: state 5, shifting to state 7
yydebug: state 7, reading 0 (end-of-file)
yydebug: state 7, reducing by rule 3 (s :)
yydebug: after reduction, shifting from state 7 to state 8
yydebug: state 8, reducing by rule 2 (s : x $$1 ’\n’ s)
yydebug: after reduction, shifting from state 0 to state 2
```

Ejemplo 4.4.2. *¿Que ocurre si en el programa 4.4.1 adelantamos la acción intermedia en la regla*

```
x →  A { $<tx>$ = strdup("**"); } x
```

y la reescribimos

`x → { $<tx>$ = strdup("**"); } A x`?

Ejecución 4.4.2. *En tal caso obtendremos:*

```
$ yacc -d -v media3.y
yacc: 1 rule never reduced
yacc: 3 shift/reduce conflicts.
```

¿Cuáles son esos 3 conflictos?

Listado 4.4.1. *El fichero* `y.output` *comienza enumerando las reglas de la gramática extendida:*

```
   0  $accept : s $end

   1  $$1 :

   2  s : x $$1 '\n' s
   3    |

   4  x : A

   5  $$2 :

   6  x : $$2 A x
^L
```

A continuación nos informa de un conflicto en el estado 0. Ante el token `A` *no se sabe si se debe desplazar al estado 1 o reducir por la regla 5:* `$$2 : .`

```
0: shift/reduce conflict (shift 1, reduce 5) on A
state 0
        $accept : . s $end  (0)
        s : .  (3)
        $$2 : .  (5)

        A  shift 1
        $end  reduce 3

        s  goto 2
        x  goto 3
        $$2  goto 4
```

Observe que, efectivamente, `$$2 : .` *esta en la clausura del estado de arranque del NFA (*`$accept : . s $end`*) Esto es asi, ya que al estar el marcador junto a* `x`*, estará el item* `s : . x $$1 '\n' s` *y de aqui que también este* `x : . $$2 A x`*.*

Además el token `A` *está en el conjunto FOLLOW(*`$$2`*) (Basta con mirar la regla 6 para confirmarlo). Por razones análogas también está en la clausura del estado de arranque el item* `x : . A` *que es el que motiva el desplazamiento al estado 1.*

La dificultad para `yacc` *se resolvería si dispusiera de información acerca de cual es el* token *que viene después de la* `A` *que causa el conflicto.*

Ejercicio 4.4.1. *¿Que acción debe tomarse en el conflicto del ejemplo 4.4.2 si el* token *que viene después de* `A` *es* `\n`*? ¿Y si el* token *es* `A`*? ¿Se debe reducir o desplazar?*

4.5. Recuperación de Errores

Las entradas de un traductor pueden contener errores. El lenguaje `yacc` proporciona un *token* especial, `error`, que puede ser utilizado en el programa fuente para extender el traductor con producciones de error que lo doten de cierta capacidad para recuperase de una entrada errónea y poder continuar analizando el resto de la entrada.

Ejecución 4.5.1. *Consideremos lo que ocurre al ejecutar el programa* `yacc` *4.1.1 con una entrada errónea:*

```
$ hoc1
yydebug: state 0, reducing by rule 1 (list :)
yydebug: after reduction, shifting from state 0 to state 1
3--2
yydebug: state 1, reading 257 (NUMBER)
yydebug: state 1, shifting to state 2
yydebug: state 2, reducing by rule 4 (expr : NUMBER)
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, reading 45 (illegal-symbol)
syntax error
yydebug: error recovery discarding state 4
yydebug: error recovery discarding state 1
yydebug: error recovery discarding state 0
```

Después de detectar el mensaje `yacc` emite el mensaje `syntax error` y comienza a sacar estados de la pìla hasta que esta se vacía.

Programa 4.5.1. *La conducta anterior puede modificarse si se introducen "reglas de recuperación de errores" como en la siguiente modificación del programa 4.1.1:*

```
%{
#define YYSTYPE double
#define YYDEBUG 1
#include <stdio.h>
%}
%token NUMBER
%left '-'
%%
list
    :
    | list '\n'
    | list error '\n'  { yyerrok; }
    | list expr   { printf("%.8g\n",$2);}
    ;

expr
    : NUMBER { $$ = $1;}
    | expr '-' expr {$$ = $1 - $3;}
    ;

%%
```

La regla `list` → `list error '\n'` es una producción de error. La idea general de uso es que, a traves de la misma, el programador le indica a `yacc` que, cuando se produce un error dentro de una expresión, descarte todos los *tokens* hasta llegar al retorno del carro y prosiga con el análisis. Además, mediante la llamada a la macro `yyerrok` el programador anuncia que, si se alcanza este punto, la recuperación puede considerarse “completa” y que `yacc` puede emitir a partir de ese moemnto mensajes de error con la seguridad de que no son consecuencia de un comportamiento inestable provocado por el primer error.

Algoritmo 4.5.1. *El esquema general del algoritmo de recuperación de errores usado por la versión actual de* `yacc` *es el siguiente:*

1. *Cuando se encuentra ante una acción de error, el analizador genera un* token `error`*.*
2. *A continuación pasa a retirar estados de la pila hasta que descubre un estado capaz de transitar ante el* token `error`*.*
3. *En este punto transita al estado correspondiente a desplazar el* token `error`*.*
4. *Entonces lee* tokens *y los descarta hasta encontrar uno que sea aceptable.*
5. *Sólo se envían nuevos mensajes de error una vez asimilados (desplazados) tres símbolos terminales. De este modos se intenta evitar la aparición masiva de mensajes de error.*

Algoritmo 4.5.2. *El cuerpo principal del analizador LR permanece sin demasiados cambios:*

```
goodtoken = 3; b = yylex();
for( ; ; ;) {
  s = top(); a = b;
  switch (action[s][a])) {
    case "shift t" : push(t); b = yylex(); goodtoken++; break;
    case "reduce A -> alpha" :
            pop(strlen(alpha));
            push(goto[top()][A];
          break;
    case "accept" : return (1);
    default : if (errorrecovery("syntax error")) return (ERROR);
  }
}
```

Algoritmo 4.5.3. *El siguiente seudocódigo es una reescritura mas detallada del algoritmo 4.5.1. Asumimos que las funciones* `pop()` *y* `popstate()` *comprueban que hay suficientes elementos en la pila para retirar. En caso contrario se emitirá un mensaje de error y se terminará el análisis.*

```
errorrecovery(char * s) {
 if (goodtoken > 2) {
   yyerror(s); goodtoken = 0;
 }
 while (action[s][error] != shift)
   popstate(s);
 push(goto[s][error]);
 s = top();
 while (action[s][a] == reduce A -> alpha) {
   pop(strlen(|alpha|));
   push(goto[top()][A];
   s = top();
 }
 switch (action[s][a])) {
   case "shift t" :
     push(t);
     b = yylex();
```

```
      goodtoken++;
      RETURN RECOVERING;
    case "accept" : return (ERROR);
    default :
      do b = yylex();
      while ((b != EOF)&&(action[s][b] == error);
      if (b == EOF)
        return (ERROR);
      else
        RETURN RECOVERING
}
```

Parecen existir diferencias en la forma en la que *bison* y *yacc* se recuperan de los errores.

Ejecución 4.5.2. *Ejecutemos el programa 4.5.1 con* `yacc`*.*

```
$ yacc -d hoc1.y; flex -l hoc1.l; gcc y.tab.c lex.yy.c; a.out
yydebug: state 0, reducing by rule 1 (list :)
yydebug: after reduction, shifting from state 0 to state 1
2--3-1
yydebug: state 1, reading 257 (NUMBER)
yydebug: state 1, shifting to state 3
yydebug: state 3, reducing by rule 5 (expr : NUMBER)
yydebug: after reduction, shifting from state 1 to state 5
yydebug: state 5, reading 45 ('-')
yydebug: state 5, shifting to state 7
yydebug: state 7, reading 45 ('-')
syntax error
```

Puesto que es el primer error, se cumple que `(goodtoken > 2)`*, emitiéndose el mensaje de error. Ahora comienzan a ejecutarse las líneas:*

```
while (!(action[s][error] != shift)) popstate(s);
```

que descartan los estados, hasta encontrar el estado que contiene el item
`list` → `list` ↑ `error '\n'`

```
yydebug: error recovery discarding state 7
yydebug: error recovery discarding state 5
yydebug: state 1, error recovery shifting to state 2
```

Una vez en ese estado, transitamos con el token `error`*,*

```
yydebug: state 2, error recovery discards token 45 ('-')
yydebug: state 2, reading 257 (NUMBER)
yydebug: state 2, error recovery discards token 257 (NUMBER)
yydebug: state 2, reading 45 ('-')
yydebug: state 2, error recovery discards token 45 ('-')
yydebug: state 2, reading 257 (NUMBER)
yydebug: state 2, error recovery discards token 257 (NUMBER)
```

Se ha procedido a descartar tokens *hasta encontrar el retorno de carro, ejecutando las líneas:*

```
b = yylex(); while ((b != EOF)&&(action[s][b] == error);
```

```
yydebug: state 2, reading 10 ('\n')
yydebug: state 2, shifting to state 6
yydebug: state 6, reducing by rule 3 (list : list error '\n')
yydebug: after reduction, shifting from state 0 to state 1
```

Al reducir por la regla de error, se ejecuta `yyerrok` *y* `yacc` *reestablece el valor de* `goodtoken`*. Si se producen nuevos errores serán señalados.*

4.6. Recuperación de Errores en Listas

Aunque no existe un método exacto para decidir como ubicar las reglas de recuperación de errores, en general, los símbolos de error deben ubicarse intentado satisfacer las siguientes reglas:

- Tan cerca como sea posible del símbolo de arranque.
- Tan cerca como sea posible de los símbolos terminales.
- Sin introducir nuevos conflictos.

Esquema 4.6.1. *En el caso particular de las listas, se recomienda seguir el siguiente esquema:*

Construct	*EBNF*	*yacc input*
optional sequence	$x : \{y\}$	`x : /* null */` $xyyyerrok;\|$ $xerror\|$
sequence	$x : y\{y\}$	`x : y` $xyyyerrok;\|$ $error\|$ $xerror\|$
list	$x : y\{Ty\}$	`x : y` $xTyyyerrok;\|$ $error\|$ $xerror\|$ $xerroryyyerrok;\|$ $xTerror\|$

Programa 4.6.1. *Para comprobar el funcionamiento y la validez de la metodología esquematizada en el esquema 4.6.1, consideremos los contenidos del fichero* `error.y`. *En el se muestra el tercer caso x:y{Ty} con x =* `list`*, T =* `,` *e y =* `NUMBER`*:*

```
%{
#include <stdio.h>
void put(double x);
void err(int code);
%}

%union {
  double val;
}
%token <val>NUMBER
%%
command
  :
  | command list ’\n’ { yyerrok; }
  ;

list
    : NUMBER            { put($1); }
    | list ’,’ NUMBER   { put($3); yyerrok; }
    | error             { err(1); }
    | list error        { err(2); }
    | list error NUMBER { err(3); put($3); yyerrok; }
    | list ’,’ error    { err(4); }
    ;
```

```
%%
void put(double x) {
  printf("%2.1lf\n",x);
}

void err(int code) {
  printf("err %d\n",code);
}

main() {
  yydebug = 1;
  yyparse();
}

yyerror(char *s) {
  printf("%s\n",s);
}
```

Listado 4.6.1. *La compilación con* `yacc` *da lugar a una tabla ligeramente diferente de la producida por* `bison`*. El fichero* `y.output` *contiene la tabla:*

```
   0  $accept : command $end

   1  command :
   2          | command list '\n'

   3  list : NUMBER
   4       | list ',' NUMBER
   5       | error
   6       | list error
   7       | list error NUMBER
   8       | list ',' error

state 0
$accept : . command $end  (0)
command : .  (1)

.  reduce 1

command  goto 1


state 1
$accept : command . $end  (0)
command : command . list '\n'  (2)

$end  accept
error  shift 2
NUMBER  shift 3
.  error

list  goto 4
```

```
state 2
list : error .  (5)

.  reduce 5


state 3
list : NUMBER .  (3)

.  reduce 3


state 4
command : command list . '\n'  (2)
list : list . ',' NUMBER  (4)
list : list . error  (6)
list : list . error NUMBER  (7)
list : list . ',' error  (8)

error  shift 5
'\n'  shift 6
','  shift 7
.  error


state 5
list : list error .  (6)
list : list error . NUMBER  (7)

NUMBER  shift 8
error  reduce 6
'\n'  reduce 6
','  reduce 6


state 6
command : command list '\n' .  (2)

.  reduce 2


state 7
list : list ',' . NUMBER  (4)
list : list ',' . error  (8)

error  shift 9
NUMBER  shift 10
.  error


state 8
list : list error NUMBER .  (7)
```

```
.  reduce 7


state 9
list : list ',' error .  (8)

.  reduce 8


state 10
list : list ',' NUMBER .  (4)

.  reduce 4


5 terminals, 3 nonterminals
9 grammar rules, 11 states
```

Ejecución 4.6.1. *La ejecución del programa generado por* `yacc` *es como sigue:*

```
> error
yydebug: state 0, reducing by rule 1 (command :)
yydebug: after reduction, shifting from state 0 to state 1
10 20
yydebug: state 1, reading 257 (NUMBER)
yydebug: state 1, shifting to state 3
yydebug: state 3, reducing by rule 3 (list : NUMBER)
10.0
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, reading 257 (NUMBER)
syntax error
yydebug: state 4, error recovery shifting to state 5
yydebug: state 5, shifting to state 8
yydebug: state 8, reducing by rule 7 (list : list error NUMBER)
err 3
20.0
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, reading 10 ('\n')
yydebug: state 4, shifting to state 6
yydebug: state 6, reducing by rule 2 (command : command list '\n')
yydebug: after reduction, shifting from state 0 to state 1
10;20 30
yydebug: state 1, reading 257 (NUMBER)
yydebug: state 1, shifting to state 3
yydebug: state 3, reducing by rule 3 (list : NUMBER)
10.0
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, reading 59 (illegal-symbol)
syntax error
yydebug: state 4, error recovery shifting to state 5
yydebug: state 5, error recovery discards token 59 (illegal-symbol)
yydebug: state 5, reading 257 (NUMBER)
yydebug: state 5, shifting to state 8
yydebug: state 8, reducing by rule 7 (list : list error NUMBER)
```

```
err 3
20.0
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, reading 257 (NUMBER)
syntax error
yydebug: state 4, error recovery shifting to state 5
yydebug: state 5, shifting to state 8
yydebug: state 8, reducing by rule 7 (list : list error NUMBER)
err 3
30.0
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, reading 10 ('\n')
yydebug: state 4, shifting to state 6
yydebug: state 6, reducing by rule 2 (command : command list '\n')
yydebug: after reduction, shifting from state 0 to state 1
3,
yydebug: state 1, reading 257 (NUMBER)
yydebug: state 1, shifting to state 3
yydebug: state 3, reducing by rule 3 (list : NUMBER)
3.0
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, reading 44 (',')
yydebug: state 4, shifting to state 7
yydebug: state 7, reading 10 ('\n')
syntax error
yydebug: state 7, error recovery shifting to state 9
yydebug: state 9, reducing by rule 8 (list : list ',' error)
err 4
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, shifting to state 6
yydebug: state 6, reducing by rule 2 (command : command list '\n')
yydebug: after reduction, shifting from state 0 to state 1
#
yydebug: state 1, reading 35 (illegal-symbol)
syntax error
yydebug: state 1, error recovery shifting to state 2
yydebug: state 2, reducing by rule 5 (list : error)
err 1
yydebug: after reduction, shifting from state 1 to state 4
yydebug: state 4, error recovery discards token 35 (illegal-symbol)
yydebug: state 4, reading 10 ('\n')
yydebug: state 4, shifting to state 6
yydebug: state 6, reducing by rule 2 (command : command list '\n')
yydebug: after reduction, shifting from state 0 to state 1
yydebug: state 1, reading 0 (end-of-file)
```

Capítulo 5

Análisis de Ámbito

5.1. Análisis de Ámbito: Conceptos

El Problema del Análisis de Ámbito

En los lenguajes de programación *name binding* (binding = encuadernado, encarpetado, ligadura, unificación) es la asociación de valores con identificadores. Decimos de un identificador ligado a un valor que es una *referencia* a dicho valor. El concepto de binding es un concepto proveído por los lenguajes de programación: a nivel de máquina no existe el concepto de binding, de relación (nombre, valor). El concepto de Binding esta intimamente relacionado con el concepto de ámbito *scope*), ya que el *análisis de ámbito* es la determinación de las relaciones de binding.

El problema del análisis de ámbito sería sencillo sino fuera porque los lenguajes de programación suelen permitir el uso del mismo *nombre*[1] para denotar distintos elementos de un programa. Es por ello que es necesario determinar que definición o declaración se aplica a una determinada ocurrencia de un elemento.

Definición 5.1.1. *En un lenguaje de programación, una* declaración *es un constructo sintáctico que define y provee información sobre un nombre. La declaración provee información sobre las propiedades asociadas con el uso del nombre: 'este nombre es una función que recibe enteros y retorna enteros', 'este nombre puede ser usado para referirse a listas de enteros y es visible sólo en el ámbito actual', etc.*

Definición 5.1.2. *Las* reglas de ámbito *de un lenguaje determinan que declaración del nombre es la que se* aplica *cuando el nombre es usado.*

Binding Estático y Binding Dinámico

En la definición anterior no se especifica en que momento se resuelve la correspondencia (nombre, definición).

Se habla de *static binding* cuando las reglas y la resolución de la correspondencia (nombre, definición) puede ser resuelta en tiempo de compilación, a partir del análisis del texto del programa fuente (también se habla de *early binding*).

Por el contrario cuando se habla de *dynamic binding* cuando la determinación de que definición se aplica a un nombre es establecida en tiempo de ejecución (también se denomina *late binding* o *virtual binding*).

Un ejemplo de static binding es una llamada a a una función en C: la función referenciada por un identificador no puede cambiarse en tiempo de ejecución. Un ejemplo de binding dinámico puede ocurrir cuando se trabaja con métodos polimorfos en un lenguaje de programación orientada a objetos, ya que la definición completa del tipo del objeto no se conoce hasta el momento de la ejecución.

El siguiente ejemplo de Dynamic_bindingtomado de la wikipedia, ilustra el binding dinámico.

Supongamos que todas las formas de vida son mortales. En OOP podemos decir que la clase `Persona` y la clase `Planta` deben implementar los métodos de `Mortal`, el cual contiene el método `muere`.

---

[1] Usamos el término nombre y no identificador ya que este último tiene una connotación mas precisa

Las personas y las plantas mueren de forma diferente, por ejemplo las plantas no tienen un corazón que se detenga. Dynamic binding es la práctica de determinar que definición/declaración se aplica a un método en tiempo de ejecución:

```
void mata(Mortal m) {
  m.muere();
}
```

No esta claro cual es la clase actual de `m`, una persona o una planta. Ambos `Planta.muere` y `Persona.muere` pueden ser invocados. Cuando se usa dynamic binding, el objeto `m` es examinado en tiempo de ejecución para determinar que método es invocado. Esto supone una 'renuncia' por parte del lenguaje y su compilador a obtener una definición completa del objeto.

Definición 5.1.3. *Cuando se usa* static binding*, la parte del texto del programa al cual se aplica la declaración de un nombre se denomina* ámbito de la declaración

Ejercicio 5.1.1. *En el siguiente código existen dos definiciones para el nombre* `one`*, una en la línea 11 y otra en la línea 20.*

```
pl@europa:~/src/perl/perltesting$ cat -n ds.pl
     1  package Father;
     2  use warnings;
     3  use strict;
     4
     5  sub new {
     6    my $class = shift;
     7
     8    bless { @_ }, $class;
     9  }
    10
    11  sub one {
    12    "Executing Father::one\n";
    13  }
    14
    15  package Child;
    16  use warnings;
    17  use strict;
    18  our @ISA = 'Father';
    19
    20  sub one {
    21    "Executing Child::one\n";
    22  }
    23
    24  package main;
    25
    26  for (1..10) {
    27    my $class = int(rand(2)) ? 'Child' : 'Father';
    28    my $c = $class->new;
    29    print $c->one;
    30  }
```

¿Que definiciones se aplican a los 10 usos del nombre `one` *en la línea 28? ¿Estos usos constituyen un ejemplo de binding estático o dinámico? ¿Cuál es el ámbito de las declaraciones de* `one`*?*

Incluso en los casos en los que la resolución del binding se deja para el momento de la ejecución el compilador debe tener información suficiente para poder generar código. En el caso anterior, el compilador de Perl infiere de la presencia de la flecha en `$c->one` que `one` es el nombre de una subrutina.

Ejercicio 5.1.2. *En el siguiente ejemplo se usa una referencia simbólica para acceder a una función:*

```
pl@europa:~/src/perl/testing$ cat -n symbolic.pl
 1  use warnings;
 2  use strict;
 3
 4  sub one {
 5    "1\n";
 6  }
 7
 8  sub two {
 9    "2\n";
10  }
11
12  my $x = <>;
13  chomp($x);
14
15  no strict 'refs';
16  print &$x();
```

Al ejecutarlo con entrada `one` *obtenenmos:*

```
pl@europa:~/src/perl/testing$ perl symbolic.pl
one
1
```

¿El uso de la línea 16 es un ejemplo de binding estático o dinámico? ¿Cuál es el binding de las declaraciones de `x`*?*

Ejercicio 5.1.3. *En el siguiente ejemplo la clase* `Two` *hereda de* `One`*.*

```
pl@europa:~/src/perl/testing$ cat -n latebinding.pl
     1  package One;
     2  use warnings;
     3  use strict;
     4
     5  our $x = 1;
     6  sub tutu {
     7    "Inside tutu: x = $x\n";
     8  }
     9
    10  package Two;
    11  use warnings;
    12  use strict;
    13  our @ISA = 'One';
    14
    15  our $x = 2;
    16
    17  print Two->tutu();
```

¿Qué definición de `$x` *se aplica al uso en la línea 7? ¿Cuál será la salida del programa?*

Definición 5.1.4. *La tarea de asignar las ocurrencias de las declaraciones de nombres a las ocurrencias de uso de los nombres de acuerdo a las reglas de ámbito del lenguaje se denomina* identificación de los nombres

Definición 5.1.5. *Una ocurrencia de un nombre se dice* local *si está en el ámbito de una declaración que no se aplica desde el comienzo de la declaración hasta el final del texto del programa. Tal declaración es una* declaración local*.*

Definición 5.1.6. *Si, por el contrario, una ocurrencia de un nombre está en el ámbito de una declaración que se aplica desde el comienzo de la declaración hasta el final del texto del programa se dice que la declaración es una* declaración global*.*

Definición 5.1.7. *Aunque la definición anterior establece el* atributo ámbito *como un atributo de la declaración es usual y conveniente hablar del*

> ”ámbito del nombre `x`*”*

como una abreviación de

> ”el ámbito de la declaración del nombre `x` que se aplica a esta ocurrencia de `x`*”*

Intervención del Programador en Tiempo de Compilación

En algunos lenguajes - especialmente en los lenguajes dinámicos- la diferenciación entre tiempo de compilación y tiempo de ejecución puede ser difusa. En el siguiente fragmento de código Perl se usa el módulo `Contextual::Return` para crear una variable cuya definición cambia con la forma de uso.

```
lhp@nereida:~/Lperl/src/testing$ cat -n context1.pl
 1  #!/usr/local/bin/perl -w
 2  use strict;
 3  use Contextual::Return;
 4
 5  my $x = BOOL { 0 } NUM { 3.14 } STR { "pi" };
 6
 7  unless ($x) { warn "¡El famoso número $x (".(0+$x).") pasa a ser falso!\n" } # executed!

lhp@nereida:~/Lperl/src/testing$ context1.pl
¡El famoso número pi (3.14) pasa a ser falso!
```

Obsérvese que el binding de `$x` es estático y que a los tres usos de `$x` en la línea 7 se les asigna la definición en la línea 5. La declaración de `$x` ocurre en lo que Perl denomina ’tiempo de compilación’, sin embargo, el hecho de que un módulo cargado en tiempo de compilación puede ejecutar sentencias permite a `Contextual::Return` expandir el lenguaje de las declaraciones Perl.

Ejercicio 5.1.4. *Considere el siguiente código Perl:*

```
pl@europa:~/src/perl/testing$ cat -n contextual.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3  use Contextual::Return;
     4
     5  sub sensible {
     6    return STR { "one" }
     7           NUM { 1 }
     8           LIST   { 1,2,3 }
     9           HASHREF   { {name => 'foo', value => 99} }
    10    ;
    11  }
```

```
12
13  print "Result = ".sensible()."\n";
14  print "Result = ".(0+sensible())."\n";
15  print "Result = ",sensible(),"\n";
16  print "Result = (name = ",sensible()->{name},", value = ", sensible()->{value},")\n";
```

Cuando se ejecuta, este programa produce la siguiente salida:

```
pl@europa:~/src/perl/testing$ ./contextual.pl
Result = one
Result = 1
Result = 123
Result = (name = foo, value = 99)
```

Las relaciones de definición-uso de la función `sensible` *¿Son un caso de binding estático o de binding dinámico? ¿Cuál es el ámbito de la declaración de* `sensible` *en las líneas 5-11?*

Visibilidad

Como ya sabemos, es falso que en el ámbito de una declaración que define a `x` dicha declaración se aplique a todas las ocurrencias de `x` en su ámbito. En un ámbito estático, una declaración local a la anterior puede *ocultar la visibilidad* de la declaración anterior de `x`.

Definición 5.1.8. *Las* reglas de visibilidad *de un lenguaje especifican como se relacionan los nombres con las declaraciones que se les aplican.*

El concepto de nombre depende del lenguaje. Algunos lenguajes permiten que en un cierto ámbito haya mas de una definición asociada con un identificador. Un mecanismo que puede ser usado para determinar univocamente que definición se aplica a un determinado uso de un nombre es que el uso del nombre vaya acompañado de un sigil. (podría reinterpretarse que en realidad el nombre de una variable en Perl incluye el sigil).

Así, en Perl tenemos que es legal tener diferentes variables con nombre `x`: `$x`, `@x`, `%x`, `&x`, `*x`, etc. ya que van prefijadas por diferentes *sigils* `$`, `@`, etc. El sigil que prefija `x` determina que definición se aplica al uso de `x` (La palabra sigil hace referencia a 'sellos mágicos' - combinaciones de símbolos y figuras geométricas - que son usados en algunas invocaciones con el propósito de producir un sortilegio).

En algunos casos existen mecanismos para la extensión de los nombres. En Perl es posible acceder a una variable de paquete ocultada por una léxica usando su nombre completo.

La asignación de una declaración a ciertos usos de un identificador puede requerir de otras fases de análisis semántico, como el análisis de tipos. El uso de los nombres de campo de un registro en Pascal y en C constituye un ejemplo:

```
type
  a = ^b;
  b = record
        a: Integer;
        b: Char;
        c: a
      end;
var
  pointertob: a;
  c         : Integer;

...
new(pointertob);

pointertob^.c := nil;
c             := 4;
...
```

El uso de `c` en la línea 15 es posible porque el tipo de la expresión `pointertob^` es un registro. La definición que se aplica al uso de `c` en la línea 16 es la de la línea 10.

También es posible hacer visible un nombre escondido - sin necesidad de extender el identificador - mediante alguna directiva que lo haga visible: Un ejemplo es la declaración `with` the Pascal:

```
new(pointertob);

with pointertob^ do
begin
  a := 10;
  b := 'A';
  c := nil
end;
...
```

Declaraciones y Definiciones

En algunos lenguajes se distingue entre declaraciones que sólo proporcionan información sobre el elemento y lo hacen *visible* - pero no asignan memoria o producen código para la implementación del mismo - y otras que si producen dicho código. A las primeras se las suele llamar declaraciones y a las segundas definiciones. En tales lenguajes se considera un error que dos declaraciones de un mismo elemento difieran.

Por ejemplo, en C una variable o función sólo es definida una vez, pero puede ser declarada varias veces. El calificativo `extern` es usado en C para indicar que una declaración provee visibilidad pero no conlleva definición (creación):

```
extern char stack[10];
extern int stkptr;
```

Estas declaraciones le dicen al compilador C que las definiciones de los nombres `stack` y `stackptr` se encuentran en otro fichero. Si la palabra `extern` fuera omitida el compilador asignaría memoria para las mismas.

Otro ejemplo en el que una directiva hace visible una definición escondida es el uso de la declaración `our` de Perl cuando un paquete está repartido entre varios ficheros que usan repetitivamente `strict`:

Ejercicio 5.1.5. *Considere el siguiente programa:*

```
pl@europa:~/src/perl/perltesting$ cat -n useA.pl
     1  #!/usr/bin/perl
     2  package A;
     3  use warnings;
     4  use strict;
     5
     6  use A;
     7
     8  #our $x;
     9  print "$x\n";
```

La variable `$x` *esta declarada en el fichero* `A.pm`*:*

```
pl@europa:~/src/perl/perltesting$ cat -n A.pm
     1  package A;
     2  use warnings;
     3  use strict;
     4
```

```
     5  our $x = 1;
     6
     7  1;
```

Sin embargo la compilación de `useA.pl` *produce errores, pues* `$x` *no es visible:*

```
pl@europa:~/src/perl/perltesting$ perl -c useA.pl
Variable "$x" is not imported at useA.pl line 9.
Global symbol "$x" requires explicit package name at useA.pl line 9.
useA.pl had compilation errors.
```

El mensaje se arregla descomentando la declaración de `$x` *en la línea 8 de* `useA.pl`*:*

```
pl@europa:~/src/perl/perltesting$ perl -ce ‘sed -e ’s/#our/our/’ useA.pl‘
-e syntax OK
```

La declaración de la línea 8 hace visible la variable `$x` *en el fichero* `useA.pl`*.*

¿Cual es entonces el ámbito de la declaración de `$x` *en la línea 5 de* `A.pm`*? ¿Es todo el paquete? ¿O sólo el segmento del paquete que está en el fichero* `A.pm`*? (se supone que trabajamos con* `strict` *activado).*

Inferencia, Declaraciones Implícitas y Ámbito

Ejercicio 5.1.6. *Tanto en los lenguajes estáticos como en los dinámicos se suele requerir que exista una declaración del objeto usado que determine las propiedades del mismo.*

Aunque en los lenguajes dinámicos la creación/definición del elemento asociado con un nombre puede postergarse hasta el tiempo de ejecución, la generación de código para la sentencia de uso suele requerir un conocimiento (aunque sea mínimo) del objeto que esta siendo usado. En algunos casos, es necesario inferir la declaración a partir del uso, de manera que la declaración asociada con un uso es construida a partir del propio uso.

Los lenguajes típicamente estáticos fuertemente tipeados suelen requerir que para todo uso exista una declaración explícita y completa del nombre y de las operaciones que son válidas sobre el mismo al finalizar la fase de compilación. Sin embargo, el código para la creación/definición de algunos objetos puede ser postergado a la fase de enlace. De hecho, la resolución de ciertos enlaces pueden ocurrir durante la fase de ejecución (énlace dinámico).

El siguiente ejemplo hace uso de un typeglob selectivo en la línea 8 para definir la función `ONE`*:*

```
pl@europa:~/src/perl/testing$ cat -n glob.pl
     1  use warnings;
     2  use strict;
     3
     4  sub one {
     5    "1\n"
     6  }
     7
     8  *ONE = \&one;
     9  print ONE();
```

Al ejecutar este programa se produce la salida;

```
pl@europa:~/src/perl/testing$ perl glob.pl
1
```

¿El uso de `ONE` *en la línea 9 es un ejemplo de binding estático o dinámico? ¿Cuál es el ámbito de la declaración de* `ONE`*?*

Responda estas mismas preguntas para esta otra variante del ejemplo anterior:

```
pl@nereida:~/src/perl/perltesting$ cat -n coderef.pl
     1  use warnings;
     2  use strict;
     3
     4  *ONE = sub { "1\n" };
     5  print ONE();
```

En los ejemplos anteriores el propio uso del nombre `ONE` actúa como una declaración: Perl deduce de la presencia de paréntesis después de `ONE` que `ONE` es el nombre de una función. Esta información es suficiente para generar el código necesario. Podría decirse que la forma del uso declara al ente `ONE` y que la línea de uso conlleva una declaración implícita. Sin embargo, la creación/definición completa de `ONE` es postergada hasta la fase de ejecución.

Ejercicio 5.1.7. *La conducta del compilador de Perl cambia si se sustituye el programa anterior por este otro:*

```
pl@europa:~/src/perl/testing$ cat -n globwarn.pl
     1  use warnings;
     2  use strict;
     3
     4  sub one {
     5    "1\n"
     6  }
     7
     8  *ONE = \&one;
     9  my $x = ONE;
    10  print $x;
```

Al compilar se obtiene un error:

```
pl@europa:~/src/perl/testing$ perl -c globwarn.pl
Bareword "ONE" not allowed while "strict subs" in use at globwarn.pl line 9.
globwarn.pl had compilation errors.
```

¿Sabría explicar la causa de este cambio de conducta?

Ejercicio 5.1.8. *El error que se observa en el ejercicio anterior desaparece cuando se modifica el código como sigue:*

```
lusasoft@LusaSoft:~/src/perl/perltesting$ cat -n globheader.pl
     1  #!/usr/bin/perl
     2  use warnings;
     3  use strict;
     4
     5  sub ONE;
     6
     7  sub one {
     8    "1\n"
     9  }
    10
    11  *ONE = \&one;
    12  my $x = ONE;
    13  print $x;
```

¿Cual es el significado de la línea 5?

En el caso del lenguaje Simple C introducido en la práctica 5.3 hay una única declaración que se aplica a cada ocurrencia correcta de un nombre en el ámbito de dicha declaración.

Esto no tiene porque ser siempre así: en ciertos lenguajes una redeclaración de un cierto nombre `x` puede que sólo oculte a otra declaración previa de `x` si las dos declaraciones asignan el mismo tipo a `x`. Esta idea suele conocerse como *sobrecarga de identificadores*. De todos modos, sigue siendo cierto que para que el programa sea considerado correcto es necesario que sea posible *inferir* para cada ocurrencia de un identificador que única definición se aplica. Así una llamada a una cierta función `min(x,y)` llamaría a diferentes funciones `min` según fueran los tipos de `x` e `y`. Para resolver este caso es necesario combinar las fases de análisis de ámbito y de análisis de tipos.

Algunos lenguajes - especialmente los lenguajes funcionales - logran eliminar la mayoría de las declaraciones. Disponen de un mecanismo de inferencia que les permite - en tiempo de compilación - deducir del uso la definición y propiedades del nombre.

Véase como ejemplo de inferencia la siguiente sesión en OCaml

```
pl@nereida:~/src/perl/attributegrammar/Language-AttributeGrammar-0.08/examples$ ocaml
        Objective Caml version 3.09.2

# let minimo = fun i j -> if i<j then i else j;;
val minimo : 'a -> 'a -> 'a = <fun>
# minimo 2 3;;
- : int = 2
# minimo 4.9 5.3;;
- : float = 4.9
# minimo "hola" "mundo";;
- : string = "hola"
```

El compilador `OCaml` infiere el tipo de las expresiones. Así el tipo asociado con la función `minimo` es

```
'a -> 'a -> 'a
```

que es una *expresión de tipo* que contiene *variables de tipo*. El operador `->` es asociativo a derechas y asi la expresión debe ser leída como `'a -> ('a -> 'a)`. Básicamente dice:

El tipo de la expresión es una función que toma un argumento de tipo `'a` (donde `'a` es una variable tipo que será instanciada en el momento del uso de la función) y devuelve una función que toma elementos de tipo `'a` y retorna elementos de tipo `'a`.

Ámbito Dinámico

En el *ámbito dinámico*, cada nombre para el que se usa ámbito dinámico tiene asociada una pila de bindings. Cuando se crea un nuevo ámbito dinámico se empuja en la pila el viejo valor (que podría no estar definido). Cuando se sale del ámbito se saca de la pila el antiguo valor. La evaluación de `x` retorna siempre el valor en el top de la pila.

La sentencia `local` de Perl provee de ámbito dinámico a las variables de paquete. Una aproximación a lo que ocurre cuando se ejecuta `local` es:

DECLARACIÓN DE `local`	SIGNIFICADO

`{`
`  local($SomeVar);`
`  $SomeVar = 'My Value';`
`       ...`
`}`
	`{`
	`  my $TempCopy = $SomeVar;`
	`  $SomeVar = undef;`
	`  $SomeVar = 'My Value';`
	`       ...`
	`  $SomeVar = $TempCopy;`
	`}`

La diferencia entre ámbito dinámico y estático debería quedar mas clara observando la conducta del siguiente código

```
lhp@nereida:~/Lperl/src$ cat -n local.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3
 4  our $x;
 5
 6  sub pr { print "$x\n"; }
 7  sub titi { my $x = "titi"; pr(); }
 8  sub toto { local $x = "toto"; &pr(); &titi(); }
 9
10  $x = "global";
11  &pr();
12  &toto();
13  &titi();
```

Cuando se ejecuta, se obtiene la siguiente salida:

```
> local.pl
global
toto
toto
global
```

Ejercicio 5.1.9. *¿Es* `local` *una declaración o una sentencia? ¿Que declaraciones se aplican a los diferentes usos de* `$x` *en las líneas 6, 7, 8 y 10?.*

5.2. Descripción Eyapp del Lenguaje SimpleC

El proceso de identificar los nombres conlleva establecer enlaces entre las ocurrencias y sus declaraciones o bien - en caso de error - determinar que dicho enlace no existe. El resultado de este proceso de identificación (o análisis de ámbito y visibilidad) será utilizado durante las fases posteriores.

En este capítulo usaremos `Parse::Eyapp` para desarrollar las primeras fases - análisis léxico, análisis sintáctico y análisis de ámbito - de un compilador para un subconjunto de C al que denominaremos `Simple C`:

El Cuerpo

```
%%
program:
    definition<%name PROGRAM +>.program
      { $program }
;

definition:
    $funcDef
      { $funcDef }
  | %name FUNCTION
    $basictype $funcDef
      { $funcDef }
  | declaration
      {}
```

```
;

basictype:
    %name INT
    INT
  | %name CHAR
    CHAR
;

funcDef:
    $ID '('  $params  ')' $block
      {
         flat_statements($block);
         $block->{parameters} = [];
         $block->{function_name} = $ID;
         $block->type("FUNCTION");
         return $block;
      }
;

params:
    ( basictype ID arraySpec)<%name PARAMS * ','>
      { $_[1] }
;

block:
    '{'.bracket
     declaration<%name DECLARATIONS *>.decs statement<%name STATEMENTS *>.sts '}'
       {
         flat_statements($sts);
         $sts->type("BLOCK") if $decs->children;
         return $sts;
       }
;

declaration:
    %name DECLARATION
    $basictype $declList ';'
;

declList:
    (ID arraySpec) <%name VARLIST + ','> { $_[1] }
;

arraySpec:
    ( '[' INUM ']')* { $_[1]->type("ARRAYSPEC"); $_[1] }
;

statement:
    expression ';' { $_[1] }
  | ';'
  | %name BREAK
    $BREAK ';'
```

```
  | %name CONTINUE
     $CONTINUE ';'
  | %name EMPTYRETURN
    RETURN ';'
  | %name RETURN
     RETURN expression ';'
  | block { $_[1] }
  | %name IF
    ifPrefix statement %prec '+'
  | %name IFELSE
    ifPrefix statement 'ELSE' statement
  | %name WHILE
    $loopPrefix statement
;

ifPrefix:
    IF '(' expression ')' { $_[3] }
;

loopPrefix:
    $WHILE '(' expression ')' { $_[3] }
;

expression:
    binary <+ ','>
      {
        return $_[1]->child(0) if ($_[1]->children() == 1);
        return $_[1];
      }
;

Variable:
    %name VAR
    ID
  | %name  VARARRAY
    $ID ('[' binary ']') <%name INDEXSPEC +>
;

Primary:
    %name INUM
    INUM
  | %name CHARCONSTANT
    CHARCONSTANT
  | $Variable
      {
        return $Variable
      }
  | '(' expression ')' { $_[2] }
  | $function_call
      {
        return $function_call  # bypass
      }
;
```

```
function_call:
  %name
    FUNCTIONCALL
    ID  '(' binary <%name ARGLIST * ','> ')'
;

Unary:
    '++' Variable
  | '--' Variable
  | Primary { $_[1] }
;

binary:
    Unary { $_[1] }
  | %name PLUS
    binary '+' binary
  | %name MINUS
    binary '-' binary
  | %name TIMES
    binary '*' binary
  | %name DIV
    binary '/' binary
  | %name MOD
    binary '%' binary
  | %name LT
    binary '<' binary
  | %name GT
    binary '>' binary
  | %name GE
    binary '>=' binary
  | %name LE
    binary '<=' binary
  | %name EQ
    binary '==' binary
  | %name NE
    binary '!=' binary
  | %name AND
    binary '&' binary
  | %name EXP
    binary '**' binary
  | %name OR
    binary '|' binary
  | %name ASSIGN
    $Variable '=' binary
  | %name PLUSASSIGN
    $Variable '+=' binary
  | %name MINUSASSIGN
    $Variable '-=' binary
  | %name TIMESASSIGN
    $Variable '*=' binary
  | %name DIVASSIGN
    $Variable '/=' binary
```

```
  | %name MODASSIGN
    $Variable '%=' binary
;
```

La Cabeza

```
/*
File: Simple/Syntax.eyp
Full Type checking
To build it, Do make or:
  eyapp -m Simple::Syntax Syntax.eyp;
*/
%{
use strict;
use Carp;
use warnings;
use Data::Dumper;
use List::MoreUtils qw(firstval);
our $VERSION = "0.4";

my $debug = 1;
my %reserved = (
  int => "INT",
  char => "CHAR",
  if => "IF",
  else => "ELSE",
  break => "BREAK",
  continue => "CONTINUE",
  return => "RETURN",
  while => "WHILE"
);

my %lexeme = (
  '='  => "ASSIGN",
  '+'  => "PLUS",
  '-'  => "MINUS",
  '*'  => "TIMES",
  '/'  => "DIV",
  '%'  => "MOD",
  '|'  => "OR",
  '&'  => "AND",
  '{'  => "LEFTKEY",
  '}'  => "RIGHTKEY",
  ','  => "COMMA",
  ';'  => "SEMICOLON",
  '('  => "LEFTPARENTHESIS",
  ')'  => "RIGHTPARENTHESIS",
  '['  => "LEFTBRACKET",
  ']'  => "RIGHTBRACKET",
  '==' => "EQ",
  '+=' => "PLUSASSIGN",
  '-=' => "MINUSASSIGN",
  '*=' => "TIMESASSIGN",
  '/=' => "DIVASSIGN",
```

```
  '%=' => "MODASSIGN",
  '!=' => "NE",
  '<'  => "LT",
  '>'  => "GT",
  '<=' => "LE",
  '>=' => "GE",
  '++' => "INC",
  '--' => "DEC",
  '**' => "EXP"
);

my ($tokenbegin, $tokenend) = (1, 1);

sub flat_statements {
  my $block = shift;

  my $i = 0;
  for ($block->children) {
    if ($_->type eq "STATEMENTS") {
      splice @{$block->{children}}, $i, 1, $_->children;
    }
    $i++;
  }
}
%}

%syntactic token RETURN BREAK CONTINUE

%right '=' '+=' '-=' '*=' '/=' '%='
%left '|'
%left '&'
%left '==' '!='
%left '<' '>' '>=' '<='
%left '+' '-'
%left '*' '/' '%'
%right '**'
%right '++' '--'
%right 'ELSE'

%tree
```

La Cola

```
%%

sub _Error {
  my($token)=$_[0]->YYCurval;
  my($what)= $token ? "input: '$token->[0]' in line $token->[1]" : "end of input";
  my @expected = $_[0]->YYExpect();
  my $expected = @expected? "Expected one of these tokens: '@expected'":"";

  croak "Syntax error near $what. $expected\n";
}
```

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

  my $token;
  for ($parser->YYData->{INPUT}) {
      return(’’,undef) if !defined($_) or $_ eq ’’;

      #Skip blanks
      s{\A
         ((?:
              \s+       # any white space char
          |   /\*.*?\*/ # C like comments
          )+
         )
       }
       {}xs
      and do {
            my($blanks)=$1;

            #Maybe At EOF
            return(’’, undef) if $_ eq ’’;
            $tokenend += $blanks =˜ tr/\n//;
        };

     $tokenbegin = $tokenend;

      s/ˆ(’.’)//
              and return(’CHARCONSTANT’, [$1, $tokenbegin]);

      s/ˆ([0-9]+(?:\.[0-9]+)?)//
              and return(’INUM’,[$1, $tokenbegin]);

      s/ˆ([A-Za-z][A-Za-z0-9_]*)//
        and do {
          my $word = $1;
          my $r;
          return ($r, [$r, $tokenbegin]) if defined($r = $reserved{$word});
          return(’ID’,[$word, $tokenbegin]);
      };

      m/ˆ(\S\S)/ and  defined($token = $1) and exists($lexeme{$token})
        and do {
          s/..//;
          return ($token, [$token, $tokenbegin]);
        }; # do

      m/ˆ(\S)/ and defined($token = $1) and  exists($lexeme{$token})
        and do {
          s/.//;
          return ($token, [$token, $tokenbegin]);
        }; # do

      die "Unexpected character at $tokenbegin\n";
```

```
  } # for
}

sub compile {
 my($self)=shift;

 my ($t);

 $self->YYData->{INPUT} = $_[0];

 $t = $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
                      #yydebug => 0x1F
     );

 return $t;
}

sub TERMINAL::value {
  return $_[0]->{attr}[0];
}

########## line Support

sub TERMINAL::line {
  return $_[0]->{attr}[1];
}

sub VAR::line {
  my $self = shift;

  return $self->child(0)->{attr}[1];
}

sub PLUS::line {
  $_[0]->{lines}[0]
}

{
no warnings;
*TIMES::line = *DIV::line = *MINUS::line = *ASSIGN::line
=*GT::line
=*IF::line
=*RETURN::line
= \&PLUS::line;

*VARARRAY::line = *FUNCTIONCALL::line
=\&VAR::line;
}

############## Debugging and Display
sub show_trees {
 my ($t) = shift;
 my $debug = shift;
```

```
 $Data::Dumper::Indent = 1;
 print Dumper $t if $debug > 3;
 local $Parse::Eyapp::Node::INDENT = $debug;
 print $t->str."\n";
}

sub TERMINAL::info {
  my $a = join ':', @{$_[0]->{attr}};
  return $a
}

sub TERMINAL::save_attributes {
  # $_[0] is a syntactic terminal
  # $_[1] is the father.
  push @{$_[1]->{lines}}, $_[0]->[1]; # save the line!
}

sub WHILE::line {
  return $_[0]->{line}
}
```

Ejemplo de Árbol Construido

```
pl@nereida:~/Lbook/code/Simple-Syntax/script$ usesyntax.pl bugmatch01.c 2
test (int n)
{
  while (1) {
    if (1>0) {
      a = 2;
      break;
    }
    else if (2> 0){
      b = 3;
      continue;
    }
  }
}

PROGRAM(
  FUNCTION(
    WHILE(
      INUM(
        TERMINAL[1:3]
      ),
      STATEMENTS(
        IFELSE(
          GT(
            INUM(
              TERMINAL[1:4]
            ),
            INUM(
              TERMINAL[0:4]
            )
```

```
          ) # GT,
          STATEMENTS(
            ASSIGN(
              VAR(
                TERMINAL[a:5]
              ),
              INUM(
                TERMINAL[2:5]
              )
            ) # ASSIGN,
            BREAK
          ) # STATEMENTS,
          IF(
            GT(
              INUM(
                TERMINAL[2:8]
              ),
              INUM(
                TERMINAL[0:8]
              )
            ) # GT,
            STATEMENTS(
              ASSIGN(
                VAR(
                  TERMINAL[b:9]
                ),
                INUM(
                  TERMINAL[3:9]
                )
              ) # ASSIGN,
              CONTINUE
            ) # STATEMENTS
          ) # IF
        ) # IFELSE
      ) # STATEMENTS
    ) # WHILE
  ) # FUNCTION
) # PROGRAM
```

Ejemplo de Árbol con Aplanamiento de `STATEMENTS`

```
pl@nereida:~/Lbook/code/Simple-Syntax/script$ usesyntax.pl prueba26.c 2
int a[20],b,e[10];

g() {}

int f(char c) {
char d;
 c = 'X';
 e[d][b] = 'A'+c;
 {
   int d;
   d = c * 2;
 }
```

```
 {
   d = a + b;
   {
     c = a + 1;
   }
 }
 c = d * 2;
 return c;
}
```

```
PROGRAM(
  FUNCTION,
  FUNCTION(
    ASSIGN(
      VAR(
        TERMINAL[c:7]
      ),
      CHARCONSTANT(
        TERMINAL['X':7]
      )
    ) # ASSIGN,
    ASSIGN(
      VARARRAY(
        TERMINAL[e:8],
        INDEXSPEC(
          VAR(
            TERMINAL[d:8]
          ),
          VAR(
            TERMINAL[b:8]
          )
        ) # INDEXSPEC
      ) # VARARRAY,
      PLUS(
        CHARCONSTANT(
          TERMINAL['A':8]
        ),
        VAR(
          TERMINAL[c:8]
        )
      ) # PLUS
    ) # ASSIGN,
    BLOCK(
      ASSIGN(
        VAR(
          TERMINAL[d:11]
        ),
        TIMES(
          VAR(
            TERMINAL[c:11]
          ),
          INUM(
```

```
            TERMINAL[2:11]
          )
        ) # TIMES
      ) # ASSIGN
    ) # BLOCK,
    ASSIGN(
      VAR(
        TERMINAL[d:14]
      ),
      PLUS(
        VAR(
          TERMINAL[a:14]
        ),
        VAR(
          TERMINAL[b:14]
        )
      ) # PLUS
    ) # ASSIGN,
    ASSIGN(
      VAR(
        TERMINAL[c:16]
      ),
      PLUS(
        VAR(
          TERMINAL[a:16]
        ),
        INUM(
          TERMINAL[1:16]
        )
      ) # PLUS
    ) # ASSIGN,
    ASSIGN(
      VAR(
        TERMINAL[c:19]
      ),
      TIMES(
        VAR(
          TERMINAL[d:19]
        ),
        INUM(
          TERMINAL[2:19]
        )
      ) # TIMES
    ) # ASSIGN,
    RETURN(
      VAR(
        TERMINAL[c:20]
      )
    ) # RETURN
  ) # FUNCTION
) # PROGRAM
```

5.3. Práctica: Construcción del AST para el Lenguaje Simple C

Reproduzca el analizador sintáctico para el lenguaje SimpleC introducido en la sección 5.2.

5.4. Práctica: Análisis de Ámbito del Lenguaje Simple C

Haga una primera parte del análisis de ámbito del lenguaje Simple C presentado en la práctica 5.3.

En esta primera parte se construyen las declaraciones y las tablas de símbolos de cada bloque. Sin embargo no queda establecida la asignación desde una instancia de un objeto a su declaración.

La tabla de Símbolos Comenzaremos describiendo las estructuras de datos que van a conformar la tabla de símbolos de nuestro compilador:

1. Dote de los siguientes atributos a los nodos de tipo "bloque", esto es a aquellos nodos que definen ámbito. Por ejemplo, los nodos asociados con la variable sintáctica `block` son de tipo "bloque":

   ```
   block:
       %name BLOCK
       '{' declaration %name DECLARATIONS * statement %name STATEMENTS * '}'
   ```

   a) Atributo `symboltable`: referencia a la tabla de símbolos asociada con el bloque La tabla de símbolos asociada al bloque es un simple hash.

   b) Atributo `fatherblock`: referencia al nodo `BLOCK` que inmediatamente anida a este. Los nodos de tipo bloque quedan enlazados según el árbol de anidamiento de bloques

   c) Los bloques con declaraciones vacías pueden ser simplificados en el árbol de bloques

2. Los nodos `FUNCTION` asociados con las funciones son nodos de tipo bloque y serán tratados de manera similar a los nodos `BLOCK`, esto es, tendrán su tabla de símbolos asociada en la cual se guardarán los parámetros de la función. Este bloque es el bloque padre del bloque formado por el cuerpo de la función. Posteriormente se pueden fusionar estos dos bloques siempre que se conserve la información necesaria sobre los parámetros.

   ```
   funcDef:
       %name FUNCTION
       ID '('  param <%name PARAMS * ','> ')'
         block
   ```

3. Los identificadores de funciones van en la tabla de símbolos global, asociada con el nodo `PROGRAM`:

   ```
   pl@nereida:~/Lbook/code/Simple-SymbolTables/lib/Simple$ sed -ne '/^program:/,/^;/p' Symbo
     program:
           { reset_file_scope_vars(); }
         definition<%name PROGRAM +>.program
           {
             $program->{symboltable} = { %st };  # creates a copy of the s.t.
             for (keys %type) {
               $type{$_} = Parse::Eyapp::Node->hnew($_);
             }
             $program->{depth} = 0;
             $program->{line}  = 1;
             $program->{types} = { %type };
             $program->{lines} = $tokenend;
   ```

```
13
14          reset_file_scope_vars();
15          $program;
16        }
17  ;
```

Observe que en C una función puede ser usada antes de que aparezca su definición.

4. Las instancias de los objetos y en particular los nodos `VAR` deben tener atributos que determinen que declaración se les aplica y en que ámbito viven. Sin embargo estos atributos no serán trabajados en esta práctica:

   a) Atributo `entry`: Su entrada en la tabla de símbolos a la que pertenece

   b) Atributo `scope`: Una referencia al nodo `BLOCK` en el que fué declarada la variable

5. La tabla de símbolos es un árbol de tablas. Cada tabla esta relacionada con un bloque. Cada tabla tiene una entrada para cada identificador que fué declarado en su bloque. En dicha entrada figuran los atributos del identificador: entre estos últimos la línea en la que fue declarado y su tipo.

¿Que es una declaración? Para facilitar el análisis de ámbito y la comprobación de tipos debemos modificar la fase de construcción del AST para producir declaraciones que permitan comprobar con facilidad la *equivalencia de tipos*

La equivalencia de tipos se realiza habitualmente mediante *expresiones de tipo*. En nuestro caso vamos a elegir una representación dual mediante cadenas y árboles de las expresiones de tipo. Las cadenas serán términos árbol. Como primera aproximación, entenderemos que dos tipos son equivalentes si las cadenas que representan sus expresiones de tipo son iguales.

```
pl@nereida:~/Lbook/code/Simple-SymbolTables/lib/Simple$ sed -ne '1,/%}/p' SymbolTables.eyp | c
  1  /*
  2  File: SymbolTables.eyp
  3  Scope Analysis: Only symbol table construction
  4  */
  5  %{
  6  use strict;
  7  use Data::Dumper;
  8  use List::MoreUtils qw(firstval lastval);
  9  use Simple::Trans;
 10
 11  my %reserved = (
 ..    ..... => "....."
 20  );
 21
 22  my %lexeme = (
 ..    '..' => ".."
 53  );
 54
 55  our ($blocks);
 56
 57  sub is_duplicated {
 ..    .........
 65  }
 66
 67  sub build_type {
```

```
  ............
}

my ($tokenbegin, $tokenend);
my %type;
my $depth;

my %st; # Global symbol table

sub build_function_scope {
  ...............
}

sub reset_file_scope_vars {
  %st = (); # reset symbol table
  ($tokenbegin, $tokenend) = (1, 1);
  %type = ( INT  => 1,
            CHAR => 1,
            VOID => 1,
          );
  $depth = 0;
}

%}
```

El código asociado con `declaration` Veamos como modificamos la construcción del AST durante las declaraciones:

```
pl@nereida:~/Lbook/code/Simple-SymbolTables/lib/Simple$ sed -ne '/^declaration:/,/^;/p' Symbol
declaration:
    %name DECLARATION
    $basictype $declList ';'
      {
         my %st; # Symbol table local to this declaration
         my $bt = $basictype->type;
         my @decs = $declList->children();
         while (my ($id, $arrspec) = splice(@decs, 0, 2)) {
           my $name = $id->{attr}[0];
           my $type = build_type($bt, $arrspec);
           $type{$type} = 1; # has too much $type for me!

           # control duplicated declarations
           die "Duplicated declaration of $name at line $id->{attr}[1]\n" if exists($st{$n
           $st{$name}->{type} = $type;
           $st{$name}->{line} = $id->{attr}[1];
         }
         return \%st;
      }
;
```

El código de la función `build_type` es como sigue:

```
pl@nereida:~/Lbook/code/Simple-SymbolTables/lib/Simple$ sed -ne '/^sub build_type/,82p' Symbol
sub build_type {
```

```
  my $bt = shift;
  my @arrayspec = shift()->children();

  my $type = '';
  for my $s (@arrayspec) {
    $type .= "A_$s->{attr}[0](";
  }
  if ($type) {
    $type = "$type$bt".(")"x@arrayspec);
  }
  else {
    $type = $bt;
  }
  return $type;
}
```

Tratamiento de los Bloques La variable sintáctica `bloque` genera el lenguaje de las definiciones seguidas de sentencias:

'{' { datadefinition } { statement } '}'

El lenguaje de los bloques es modificado en consonancia:

```
pl@nereida:~/Lbook/code/Simple-SymbolTables/lib/Simple$ sed -ne '/^block:/,/^;/p' SymbolTables
block:
    '{'.bracket declaration<%name DECLARATIONS *>.decs statement<%name STATEMENTS *>.sts
       {
         my %st;

         for my $lst ($decs->children) {

             # control duplicated declarations
           my $message;
           die $message if $message = is_duplicated(\%st, $lst);

           %st = (%st, %$lst);
         }
         $sts->{symboltable} = \%st;
         $sts->{line} = $bracket->[1];
         $sts->type("BLOCK") if %st;
         return $sts;
       }

;
```

El código de la función `is_duplicated` es como sigue:

```
sub is_duplicated {
  my ($st1, $st2) = @_;

  my $id;

    defined($id=firstval{exists $st1->{$_}} keys %$st2)
  and return "Error. Variable $id at line $st2->{$id}->{line} declared twice.\n";
  return 0;
}
```

Tratamiento de las Funciones El código para las funciones es similar. Unimos la tabla de símbolos de los parámetros a la del bloque para posteriormente reconvertir el bloque en un nodo `FUNCTION`:

```
pl@nereida:~/Lbook/code/Simple-SymbolTables/lib/Simple$ sed -ne '/^funcDef:/,/^;/p' SymbolTabl
  1  funcDef:
  2      $ID '('  $params  ')'
  3        $block
  4      {
  5         my $st = $block->{symboltable};
  6         my @decs = $params->children();
  7         $block->{parameters} = [];
  8         while (my ($bt, $id, $arrspec) = splice(@decs, 0, 3)) {
  9             my $bt = ref($bt); # The string 'INT', 'CHAR', etc.
 10             my $name = $id->{attr}[0];
 11             my $type = build_type($bt, $arrspec);
 12             $type{$type} = 1; # has too much $type for me!
 13
 14             # control duplicated declarations
 15             #die "Duplicated declaration of $name at line $id->{attr}[1]\n" if exists($st-
 16             die "Duplicated declaration of $name at line $st->{$name}{line}\n" if exists($
 17             $st->{$name}->{type} = $type;
 18             $st->{$name}->{param} = 1;
 19             $st->{$name}->{line} = $id->{attr}[1];
 20             push @{$block->{parameters}}, $name;
 21         }
 22         $block->{function_name} = $ID;
 23         $block->type("FUNCTION");
 24         return $block;
 25      }
 26  ;
```

En la línea 8 se procesan las declaraciones de parámetros. La lista de parámetros estaba definida por:

```
params:
    ( basictype ID arraySpec)<%name PARAMS * ','>
      { $_[1] }
```

En una segunda fase, a la altura de `definition` se construye la entrada para la función en la tabla de símbolos global:

```
pl@nereida:~/Lbook/code/Simple-SymbolTables/lib/Simple$ sed -ne '/^definition:/,/^;/p' SymbolT
definition:
    $funcDef
      {
        build_function_scope($funcDef, 'INT');
      }
  | %name FUNCTION
    $basictype $funcDef
      {
        build_function_scope($funcDef, $basictype->type);
      }
  | declaration
     {
       #control duplicated declarations
```

```
       my $message;
       die $message if $message = is_duplicated(\%st, $_[1]);
       %st = (%st,  %{$_[1]}); # improve this code
       return undef; # will not be inserted in the AST
     }
;
```

El código de la función `build_function_scope` es como sigue:

```
pl@nereida:~/Lbook/code/Simple-SymbolTables/lib/Simple$ sed -ne '/sub build_func/,114p' Symbol
sub build_function_scope {
  my ($funcDef, $returntype) = @_;

  my $function_name = $funcDef->{function_name}[0];
  my @parameters = @{$funcDef->{parameters}};
  my $lst = $funcDef->{symboltable};
  my $numargs = scalar(@parameters);

  #compute type
  my $partype = "";
  my @types = map { $lst->{$_}{type} } @parameters;
  $partype .= join ",", @types if @types;
  my $type = "F(X_$numargs($partype),$returntype)";

  #insert it in the hash of types
  $type{$type} = 1;

  #insert it in the global symbol table
  die "Duplicated declaration of $function_name at line $funcDef-->{attr}[1]\n"
    if exists($st{$function_name});
  $st{$function_name}->{type} = $type;
  $st{$function_name}->{line} = $funcDef->{function_name}[1]; # redundant

  return $funcDef;
}
```

El Tratamiento del Programa Principal Sigue el código (incompleto) asociado con la variable sintáctica de arranque:

```
pl@nereida:~/Lbook/code/Simple-SymbolTables/lib/Simple$ sed -ne '/^program:/,/^;/p' SymbolTabl
program:
      { reset_file_scope_vars(); }
    definition<%name PROGRAM +>.program
      {
        $program->{symboltable} = { %st };  # creates a copy of the s.t.
        for (keys %type) {
          $type{$_} = Parse::Eyapp::Node->hnew($_);
        }
        $program->{depth} = 0;
        $program->{line}  = 1;
        $program->{types} = { %type };
        $program->{lines} = $tokenend;

        reset_file_scope_vars();
```

```
15          $program;
16        }
17  ;
```

Las árboles asociados con las expresiones de tipo son calculados en las líneas 6-8.

Ejemplo de Salida: Control de declaraciones duplicadas

A estas alturas del análisis de ámbito podemos controlar la aparición de declaraciones duplicadas: para el programa de entrada:

```
pl@nereida:~/Lbook/code/Simple-SymbolTables/script$ usesymboltables.pl declaredtwice.c 2
int a, b[1][2];
char d, e[1][2];
char f(int a, char b[10]) {
  int c[1][2];
  char b[10], e[9]; /* b is declared twice */

  return b[0];
}

Duplicated declaration of b at line 5
```

Sin embargo no podemos conocer los objetos no declarados (variables, funciones) hasta que hayamos finalizado el análisis de ámbito.

Un ejemplo de Volcado de un AST

Sigue un ejemplo de salida para un programa sin errores de ámbito:

```
pl@nereida:~/Lbook/code/Simple-SymbolTables/script$ usesymboltables.pl prueba10.c 2
int a,b;
f(int a, int b) {
  b = a+1;
}

int main() {
  a = 4;
  {
    int d;
    d = f(a);
  }
}

PROGRAM^{0}(
  FUNCTION[f]^{1}(
    ASSIGN(
      VAR[No declarado!](
        TERMINAL[b:3]
      ),
      PLUS(
        VAR[No declarado!](
          TERMINAL[a:3]
        ),
        INUM(
          TERMINAL[1:3]
```

```
        )
      ) # PLUS
    ) # ASSIGN
  ) # FUNCTION,
  FUNCTION[main]^{2}(
    ASSIGN(
      VAR[No declarado!](
        TERMINAL[a:7]
      ),
      INUM(
        TERMINAL[4:7]
      )
    ) # ASSIGN,
    BLOCK[8]^{3}(
      ASSIGN(
        VAR[No declarado!](
          TERMINAL[d:10]
        ),
        FUNCTIONCALL[No declarado!](
          TERMINAL[f:10],
          ARGLIST(
            VAR[No declarado!](
              TERMINAL[a:10]
            )
          ) # ARGLIST
        ) # FUNCTIONCALL
      ) # ASSIGN
    ) # BLOCK
  ) # FUNCTION
) # PROGRAM
---------------------------
0)
Types:
$VAR1 = {
  'CHAR' => bless( {
    'children' => []
  }, 'CHAR' ),
  'VOID' => bless( {
    'children' => []
  }, 'VOID' ),
  'INT' => bless( {
    'children' => []
  }, 'INT' ),
  'F(X_0(),INT)' => bless( {
    'children' => [
      bless( {
        'children' => []
      }, 'X_0' ),
      $VAR1->{'INT'}
    ]
  }, 'F' ),
  'F(X_2(INT,INT),INT)' => bless( {
    'children' => [
```

```
      bless( {
        'children' => [
          $VAR1->{'INT'},
          $VAR1->{'INT'}
        ]
      }, 'X_2' ),
      $VAR1->{'INT'}
    ]
  }, 'F' )
};
Symbol Table of the Main Program:
$VAR1 = {
  'a' => {
    'type' => 'INT',
    'line' => 1
  },
  'b' => {
    'type' => 'INT',
    'line' => 1
  },
  'f' => {
    'type' => 'F(X_2(INT,INT),INT)',
    'line' => 2
  },
  'main' => {
    'type' => 'F(X_0(),INT)',
    'line' => 6
  }
};

---------------------------
1)
Symbol Table of f
$VAR1 = {
  'a' => {
    'type' => 'INT',
    'param' => 1,
    'line' => 2
  },
  'b' => {
    'type' => 'INT',
    'param' => 1,
    'line' => 2
  }
};

---------------------------
2)
Symbol Table of main
$VAR1 = {};

---------------------------
3)
```

```
Symbol Table of block at line 8
$VAR1 = {
  'd' => {
    'type' => 'INT',
    'line' => 9
  }
};

pl@nereida:~/Lbook/code/Simple-SymbolTables/script$
```

Métodos de soporte para el Volcado del Arbol

Se han utilizado las siguientes funciones de soporte para producir el volcado:

```
pl@nereida:~/Lbook/code/Simple-SymbolTables/lib/Simple$ sed -ne '683,$p' SymbolTables.eyp
############## Debugging and Display
sub show_trees {
 my ($t) = @_;

 print Dumper $t if $debug > 1;
 $Parse::Eyapp::Node::INDENT = 2;
 $Data::Dumper::Indent = 1;
 print $t->str."\n";
}

$Parse::Eyapp::Node::INDENT = 1;
sub TERMINAL::info {
  my @a = join ':', @{$_[0]->{attr}};
  return "@a"
}

sub PROGRAM::footnote {
  return "Types:\n"
         .Dumper($_[0]->{types}).
         "Symbol Table of the Main Program:\n"
         .Dumper($_[0]->{symboltable})
}

sub FUNCTION::info {
  return $_[0]->{function_name}[0]
}

sub FUNCTION::footnote {
  my $text = '';
  $text .= "Symbol Table of $_[0]->{function_name}[0]\n" if $_[0]->type eq 'FUNCTION';
  $text .= "Symbol Table of block at line $_[0]->{line}\n" if $_[0]->type eq 'BLOCK';
  $text .= Dumper($_[0]->{symboltable});
  return $text;
}

sub BLOCK::info {
  my $info = "$_[0]->{line}";
  return $info;
}
```

```
*BLOCK::footnote = \&FUNCTION::footnote;

sub VAR::info {
  return $_[0]->{type} if defined $_[0]->{type};
  return "No declarado!";
}

*FUNCTIONCALL::info = *VARARRAY::info = \&VAR::info;
pl@nereida:~/Lbook/code/Simple-SymbolTables/lib/Simple$
```

5.5. La Dificultad de Elaboración de las Pruebas

A la hora de hacer las pruebas necesitamos comprobar que dos árboles son iguales. El problema que aparece en el diseño de un compilador es un caso particular de la regla *defina su API antes de realizar las pruebas* y de la ausencia de herramientas adecuadas.

Como el diseño de un compilador se hace por fases y las fases a veces se superponen resulta que el resultado de una llamada a la misma subrutina cambia en las diversas etapas del desarrollo. Terminada la fase de análisis sintáctico el producto es un AST. cuando posteriormente añadimos la fase de análisis de ámbito obtenemos un árbol decorado. Si hicimos pruebas para la primera fase y hemos usado `is_deeply` las pruebas no funcionaran con la versión ampliada a menos que se cambie el árbol esperado. Ello es debido a que `is_deeply` requiere la igualdad estructural total de los dos árboles.

Limitaciones de `Data::Dumper`

La descripción dada con `Data::Dumper` de una estructura de datos anidada como es el árbol es difícil de seguir, especialmente como en este ejemplo en el que hay enlaces que autoreferencian la estructura.

```
$VAR1 = bless( {
  'types' => {
    'CHAR' => bless( { 'children' => [] }, 'CHAR' ),
    'INT' => bless( { 'children' => [] }, 'INT' ),
    'F(X_0(),CHAR)' => bless( { 'children' => [
        bless( { 'children' => [] }, 'X_0' ), bless( { 'children' => [] }, 'CHAR' )
      ]
    }, 'F' ),
    'F(X_0(),INT)' => bless( { 'children' => [
        bless( { 'children' => [] }, 'X_0' ), bless( { 'children' => [] }, 'INT' )
      ]
    }, 'F' )
  },
  'symboltable' => {
    'd0' => { 'type' => 'CHAR', 'line' => 1 },
    'f'  => { 'type' => 'F(X_0(),CHAR)', 'line' => 2 },
    'g'  => { 'type' => 'F(X_0(),INT)', 'line' => 13 },
  },
  'lines' => 19,
  'children' => [
    bless( { # FUNCTION f  <----------------------------------------------------x
      'parameters' => [], 'symboltable' => {}, 'fatherblock' => $VAR1,         |
      'function_name' => [ 'f', 2 ],                                            |
      'children' => [                                                           |
        bless( {    <-----------------------------------------------------------|------x
```

```
          'line' => 3                                                          |      |
          'symboltable' => {}, 'fatherblock' => $VAR1->{'children'}[0], -------x      |
          'children' => [                                                      |      |
            bless( {                                                           |      |
              'symboltable' => {}, 'fatherblock' => $VAR1->{'children'}[0]{'children'}[0],
              'children' => [], 'line' => 4                                    |
            }, 'BLOCK' )                                                       |
          ],                                                                   |
        }, 'BLOCK' ),                                                          |
        bless( {                                                               |
          'symboltable' => {}, 'fatherblock' => $VAR1->{'children'}[0], -------x
          'children' => [
            bless( {
              'symboltable' => {}, 'fatherblock' => $VAR1->{'children'}[0]{'children'}[1],
              'children' => [], 'line' => 7
            }, 'BLOCK' )
          ],
          'line' => 6
        }, 'BLOCK' ),
        bless( {
          'symboltable' => {}, 'fatherblock' => $VAR1->{'children'}[0],
          'children' => [
            bless( {
              'symboltable' => {}, 'fatherblock' => $VAR1->{'children'}[0]{'children'}[2],
              'children' => [
                bless( {
                  'symboltable' => {}, 'fatherblock' => $VAR1->{'children'}[0]{'children'}[2]{
                  'children' => [], 'line' => 10
                }, 'BLOCK' )
              ],
              'line' => 10
            }, 'BLOCK' )
          ],
          'line' => 9
        }, 'BLOCK' )
      ],
      'line' => 2
    }, 'FUNCTION' ),
    bless( { # FUNCTION g
      'parameters' => [], 'symboltable' => {}, 'fatherblock' => $VAR1,
      'function_name' => [ 'g', 13 ],
      'children' => [
        bless( {
          'symboltable' => {}, 'fatherblock' => $VAR1->{'children'}[1],
          'children' => [], 'line' => 14
        }, 'BLOCK' ),
        bless( {
          'symboltable' => {}, 'fatherblock' => $VAR1->{'children'}[1],
          'children' => [
            bless( {
              'symboltable' => {}, 'fatherblock' => $VAR1->{'children'}[1]{'children'}[1],
              'children' => [], 'line' => 16
            }, 'BLOCK' )
```

```
          ],
          'line' => 15
        }, 'BLOCK' ),
        bless( {
          'symboltable' => {}, 'fatherblock' => $VAR1->{'children'}[1],
          'children' => [], 'line' => 18
        }, 'BLOCK' )
      ],
      'line' => 13
    }, 'FUNCTION' )
  ],
  'line' => 1
}, 'PROGRAM' );
```

La estructura es en realidad el árbol de análisis abstracto decorado para un programa SimpleC (véase el capítulo 5). Para ver el fuente que se describe y la estructura del AST consulte la tabla 5.1. Al decorar el árbol con la jerarquía de bloques (calculada en la sección **??**, página **??**) se producen númerosas referencias cruzadas que hacen difícil de leer la salida de `Data::Dumper`.

La opción `Data::Dumper::Purity`

El problema no es sólo que es dificil seguir una estructura que se autoreferencia como la anterior. Una estructura recursiva como esta no puede ser evaluada como código Perl, ya que *Perl prohibe que una variable pueda ser usada antes de que finalice su definición*. Por ejemplo, la declaración-inicialización:

```
my $x = $x -1;
```

es considerada incorrecta. Eso significa que *el texto producido por* `Data::Dumper` *de esta forma no puede ser insertado en un test de regresión*.

El módulo `Data::Dumper` dispone de la variable `Data::Dumper::Purity` la cual ayuda a subsanar esta limitación. Veamos la salida que se produce para el programa anterior cuando `Data::Dumper::Purity` esta activa:

```
$VAR1 = bless( {
  'types' => { ... },
  'symboltable' => { ... },
  'lines' => 19,
  'children' => [
    bless( {
      'function_name' => [ 'f', 2 ], 'line' => 2,
      'parameters' => [], 'symboltable' => {}, 'fatherblock' => {},
      'children' => [
        bless( {
          'symboltable' => {}, 'fatherblock' => {}, 'line' => 3
          'children' => [
            bless( { 'symboltable' => {}, 'fatherblock' => {}, 'children' => [], 'line' => 4 }
          ],
        }, 'BLOCK' ),
        bless( {
          'symboltable' => {}, 'fatherblock' => {}, 'line' => 6
          'children' => [
            bless( {
              'symboltable' => {},
              'fatherblock' => {},
              'children' => [],
```

```
              'line' => 7
            }, 'BLOCK' )
          ],
        }, 'BLOCK' ),
        bless( {
          'line' => 9, 'symboltable' => {}, 'fatherblock' => {},
          'children' => [
            bless( {
              'line' => 10, 'symboltable' => {}, 'fatherblock' => {},
              'children' => [
                bless( {
                  'line' => 10, 'symboltable' => {}, 'fatherblock' => {},
                  'children' => [],
                }, 'BLOCK' )
              ],
            }, 'BLOCK' )
          ],
        }, 'BLOCK' )
      ],
    }, 'FUNCTION' ),
    bless( {
      'function_name' => [ 'g', 13 ],
       ............................
    }, 'FUNCTION' )
  ],
  'line' => 1
}, 'PROGRAM' );
$VAR1->{'children'}[0]{'fatherblock'} = $VAR1;
$VAR1->{'children'}[0]{'children'}[0]{'fatherblock'} = $VAR1->{'children'}[0];
$VAR1->{'children'}[0]{'children'}[0]{'children'}[0]{'fatherblock'}
  = $VAR1->{'children'}[0]{'children'}[0];
$VAR1->{'children'}[0]{'children'}[1]{'fatherblock'}
  = $VAR1->{'children'}[0];
$VAR1->{'children'}[0]{'children'}[1]{'children'}[0]{'fatherblock'}
  = $VAR1->{'children'}[0]{'children'}[1];
$VAR1->{'children'}[0]{'children'}[2]{'fatherblock'}
  = $VAR1->{'children'}[0];
$VAR1->{'children'}[0]{'children'}[2]{'children'}[0]{'fatherblock'}
  = $VAR1->{'children'}[0]{'children'}[2];
$VAR1->{'children'}[0]{'children'}[2]{'children'}[0]{'children'}[0]{'fatherblock'}
  = $VAR1->{'children'}[0]{'children'}[2]{'children'}[0];
$VAR1->{'children'}[1]{'fatherblock'} = $VAR1;
$VAR1->{'children'}[1]{'children'}[0]{'fatherblock'} = $VAR1->{'children'}[1];
$VAR1->{'children'}[1]{'children'}[1]{'fatherblock'} = $VAR1->{'children'}[1];
$VAR1->{'children'}[1]{'children'}[1]{'children'}[0]{'fatherblock'}
  = $VAR1->{'children'}[1]{'children'}[1];
$VAR1->{'children'}[1]{'children'}[2]{'fatherblock'}
  = $VAR1->{'children'}[1];
```

Observe que este segundo código elimina las definiciones recursivas, retrasa las asignaciones y es código correcto. Es por tanto apto para reproducir la estructura de datos en un programa de prueba con, por ejemplo, `is_deeply`.

Por que no Usar `str` en la Elaboración de las Pruebas

Puede que parezca una buena idea volcar los dos árboles esperado y obtenido mediante `str` y proceder a comparar las cadenas. Esta estrategia es inadecuado ya que depende de la versión de `Parse::Eyapp` con la que se esta trabajando. Si un usuario de nuestro módulo ejecuta la prueba con una versión distinta de `Parse::Eyapp` de la que hemos usado para la construcción de la prueba, puede obtener un ”falso fallo” debido a que su version de `str` trabaja de forma ligeramente distinta.

El Método `equal` de los Nodos

El método `equal` (version 1.094 de Parse::Eyapp o posterior) permite hacer comparaciones ”difusas” de los nodos. El formato de llamada es:

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

Dos nodos se consideran iguales si:

1. Sus raices `$tree1` y `$tree2` pertenecen a la misma clase
2. Tienen el mismo número de hijos
3. Para cada una de las claves especificadas `attr1`, `attr2`, etc. la existencia y definición es la misma en ambas raíces
4. Supuesto que en ambos nodos el atributo `attr` existe y está definido el manejador `handler($tree1, $tree2)` retorna cierto cuando es llamado
5. Los hijos respectivos de ambos nodos son iguales (en sentido recursivo o inductivo)

Sigue un ejemplo:

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

Usando `equal` en las Pruebas

Cuando desarrolle pruebas y desee obtener una comparación parcial del AST esperado con el AST obtenido puede usar la siguiente metodología:

1. Vuelque el árbol desde su programa de desarrollo con `Data::Dumper`
2. Compruebe que el árbol es correcto
3. Decida que atributos quiere comparar
4. Escriba la comparación de las estructuras y de los atributos pegando la salida de `Data::Dumper` y usando `equal` (versión 1.096 de `Parse::Eyapp`) como en el siguiente ejemplo:

```
pl@nereida:~/LEyapp/examples$ cat -n testequal.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use Parse::Eyapp::Node;
 4  use Data::Dumper;
 5  use Data::Compare;
 6
 7  my $debugging = 0;
 8
 9  my $handler = sub {
10    print Dumper($_[0], $_[1]) if $debugging;
11    Compare($_[0], $_[1])
12  };
```

El manejador `$handler` es usado para comparar atributos (véanse las líneas 104-109). Copie el resultado de `Data::Dumper` en la variable `$t1`:

```
14  my $t1 = bless( {
15                   ’types’ => {
16                                ’CHAR’ => bless( { ’children’ => [] }, ’CHAR’ ),
17                                ’VOID’ => bless( { ’children’ => [] }, ’VOID’ ),
18                                ’INT’ => bless( { ’children’ => [] }, ’INT’ ),
19                                ’F(X_0(),INT)’ => bless( {
20                                   ’children’ => [
21                                      bless( { ’children’ => [] }, ’X_0’ ),
22                                      bless( { ’children’ => [] }, ’INT’ ) ]
23                                 }, ’F’ )
24                              },
25                   ’symboltable’ => { ’f’ => { ’type’ => ’F(X_0(),INT)’, ’line’ => 1 }
26                   ’lines’ => 2,
27                   ’children’ => [
28                                   bless( {
29                                            ’symboltable’ => {},
30                                            ’fatherblock’ => {},
31                                            ’children’ => [],
32                                            ’depth’ => 1,
33                                            ’parameters’ => [],
34                                            ’function_name’ => [ ’f’, 1 ],
35                                            ’symboltableLabel’ => {},
36                                            ’line’ => 1
37                                          }, ’FUNCTION’ )
38                                 ],
```

```
                 ’depth’ => 0,
                 ’line’ => 1
               }, ’PROGRAM’ );
$t1->{’children’}[0]{’fatherblock’} = $t1;
```

Para ilustrar la técnica creamos dos árboles `$t2` y `$t3` similares al anterior que compararemos con `$t1`. De hecho se han obtenido del texto de `$t1` suprimiendo algunos atributos. El árbol de `$t2` comparte con `$t1` los atributos ”comprobados” mientras que no es así con `$t3`:

```
# Tree similar to $t1 but without some attttributes (line, depth, etc.)
my $t2 = bless( {
                 ’types’ => {
                              ’CHAR’ => bless( { ’children’ => [] }, ’CHAR’ ),
                              ’VOID’ => bless( { ’children’ => [] }, ’VOID’ ),
                              ’INT’ => bless( { ’children’ => [] }, ’INT’ ),
                              ’F(X_0(),INT)’ => bless( {
                                 ’children’ => [
                                    bless( { ’children’ => [] }, ’X_0’ ),
                                    bless( { ’children’ => [] }, ’INT’ ) ]
                               }, ’F’ )
                            },
                 ’symboltable’ => { ’f’ => { ’type’ => ’F(X_0(),INT)’, ’line’ => 1 }
                 ’children’ => [
                                 bless( {
                                          ’symboltable’ => {},
                                          ’fatherblock’ => {},
                                          ’children’ => [],
                                          ’parameters’ => [],
                                          ’function_name’ => [ ’f’, 1 ],
                                        }, ’FUNCTION’ )
                               ],
               }, ’PROGRAM’ );
$t2->{’children’}[0]{’fatherblock’} = $t2;

# Tree similar to $t1 but without some attttributes (line, depth, etc.)
# and without the symboltable attribute
my $t3 = bless( {
                 ’types’ => {
                              ’CHAR’ => bless( { ’children’ => [] }, ’CHAR’ ),
                              ’VOID’ => bless( { ’children’ => [] }, ’VOID’ ),
                              ’INT’ => bless( { ’children’ => [] }, ’INT’ ),
                              ’F(X_0(),INT)’ => bless( {
                                 ’children’ => [
                                    bless( { ’children’ => [] }, ’X_0’ ),
                                    bless( { ’children’ => [] }, ’INT’ ) ]
                               }, ’F’ )
                            },
                 ’children’ => [
                                 bless( {
                                          ’symboltable’ => {},
                                          ’fatherblock’ => {},
                                          ’children’ => [],
                                          ’parameters’ => [],
                                          ’function_name’ => [ ’f’, 1 ],
```

```
                                        }, ’FUNCTION’ )
                               ],
               }, ’PROGRAM’ );

$t3->{’children’}[0]{’fatherblock’} = $t2;
```

Ahora realizamos las comprobaciones de igualdad mediante `equal`:

```
# Without attributes
if (Parse::Eyapp::Node::equal($t1, $t2)) {
  print "\nNot considering attributes: Equal\n";
}
else {
  print "\nNot considering attributes: Not Equal\n";
}

# Equality with attributes
if (Parse::Eyapp::Node::equal(
      $t1, $t2,
      symboltable => $handler,
      types => $handler,
    )
   ) {
      print "\nConsidering attributes: Equal\n";
}
else {
  print "\nConsidering attributes: Not Equal\n";
}

# Equality with attributes
if (Parse::Eyapp::Node::equal(
      $t1, $t3,
      symboltable => $handler,
      types => $handler,
    )
   ) {
      print "\nConsidering attributes: Equal\n";
}
else {
  print "\nConsidering attributes: Not Equal\n";
}
```

Dado que los atibutos usados son `symboltable` y `types`, los árboles `$t1` y `$t2` son considerados equivalentes. No asi `$t1` y `$t3`. Observe el modo de llamada `Parse::Eyapp::Node::equal` como subrutina, no como método. Se hace así porque `$t1`, `$t2` y `$t3` no son objetos `Parse::Eyapp::Node`. La salida de `Data::Dumper` reconstruye la forma estructural de un objeto pero no reconstruye la información sobre la jerarquía de clases.

```
pl@nereida:~/LEyapp/examples$ testequal.pl

Not considering attributes: Equal

Considering attributes: Equal

Considering attributes: Not Equal
```

5.6. Análisis de Ámbito con `Parse::Eyapp::Scope`

Para calcular la relación entre una instancia de un identificador y su declaración usaremos el módulo `Parse::Eyapp::Scope`:

```
l@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne '1,/^ *$/p' Scope.eyp | cat -n
/*
File: lib/Simple/Scope.eyp
Full Scope Analysis
Test it with:
  lib/Simple/
  eyapp -m Simple::Scope Scope.eyp
  treereg -nonumbers -m Simple::Scope Trans
  script/
  usescope.pl prueba12.c
*/
%{
use strict;
use Data::Dumper;
use List::MoreUtils qw(firstval lastval);
use Simple::Trans;
use Parse::Eyapp::Scope qw(:all);
```

Véamos el manejo de las reglas de `program`:

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne '/^program:/,/^;$/p' Scope.eyp | cat
program:
      {
        reset_file_scope_vars();
      }
    definition<%name PROGRAM +>.program
      {
        $program->{symboltable} = { %st };  # creates a copy of the s.t.
        for (keys %type) {
          $type{$_} = Parse::Eyapp::Node->hnew($_);
        }
        $program->{depth} = 0;
        $program->{line}  = 1;
        $program->{types} = { %type };
        $program->{lines} = $tokenend;

        my ($nondec, $declared) = $ids->end_scope($program->{symboltable}, $program, 'type

        if (@$nondec) {
          warn "Identifier ".$_->key." not declared at line ".$_->line."\n" for @$nondec;
          die "\n";
        }

        # Type checking: add a direct pointer to the data-structure
        # describing the type
        $_->{t} = $type{$_->{type}} for @$declared;

        my $out_of_loops = $loops->end_scope($program);
        if (@$out_of_loops) {
          warn "Error: ".ref($_)." outside of loop at line $_->{line}\n" for @$out_of_loop
```

```
     die "\n";
    }

    # Check that are not dangling breaks
    reset_file_scope_vars();

    $program;
   }
;
```

Antes de comenzar la construcción del AST se inicializan las variables visibles en todo el fichero:

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne ’/^sub reset_file_scope_vars/,/^}$/p’
 1  sub reset_file_scope_vars {
 2    %st = (); # reset symbol table
 3    ($tokenbegin, $tokenend) = (1, 1);
 4    %type = ( INT  => 1,
 5              CHAR => 1,
 6              VOID => 1,
 7            );
 8    $depth = 0;
 9    $ids = Parse::Eyapp::Scope->new(
10             SCOPE_NAME => ’block’,
11             ENTRY_NAME => ’info’,
12             SCOPE_DEPTH => ’depth’,
13    );
14    $loops = Parse::Eyapp::Scope->new(
15             SCOPE_NAME => ’exits’,
16    );
17    $ids->begin_scope();
18    $loops->begin_scope(); # just for checking
19  }
```

El Método `Parse::Eyapp::Scope->new`

El método `Parse::Eyapp::Scope->new` crea un objeto del tipo manejador de ámbito.

Un *manejador de ámbito* es un objeto que ayuda en el cómputo de la función que asigna a cada *nodo instancia* de un nombre (variable, función, etiqueta, constante, identificador de tipo, etc.) su *declaración*, esto es su entrada en su tabla de símbolos.

En la línea 9 creamos el manejador de ámbito de los objetos identificadores `$ids` (ámbito de variables, funciones, etc.). En la línea 14 creamos un manejador de ámbito para los bucles (sentencias `CONTINUE`, `break`, etc.). El manejo de ámbito de los bucles consiste en asignar cada ocurrencia de una sentencia `BREAK` o `CONTINUE` con el bucle en el que ocurre. Por ejemplo:

```
pl@nereida:~/Lbook/code/Simple-Scope/script$ usescope.pl outbreak.c 2
  1 test (int n, int m)
  2 {
  3   break;
  4   while (n > 0) {
  5     if (n>m) {
  6       break;
  7     }
  8     else if (m>n){
  9       continue;
 10     }
```

```
 11     n = n-1;
 12   }
 13 }
Error: BREAK outside of loop at line 3
```

En esta sección mostraremos como usar `Parse::Eyapp::Scope` cuando trabajemos en el análisis de condiciones dependientes del contexto.

La filosofía de Parse::Eyapp::Scope es que existen tres tipos de nodos en el AST:

1. Nodos que definen un ámbito y que tienen asociado un atributo 'tabla de símbolos', por ejemplo: Nodos programa, nodos función, nodos bloque conteniendo declaraciones, etc. Además del atributo tabla de símbolos es común que dichos nodos dispongan de un atributo 'profundidad de ámbito' que indique su nivel de anidamiento cuando el lenguaje siendo analizado usa ámbitos anidados.

2. Nodos que conllevan un uso de un nombre (*nodos de uso*): por ejemplo, nodos de uso de una variable en una expresión. El propósito del análisis de ámbito es dotar a cada uno de estos nodos de uso con un atributo 'scope' que referencia al *nodo ámbito* en el que se ha guardado la información que define las propiedades del objeto. Es posible que además queramos tener en dicho nodo un atributo 'entry' que sea una referencia directa a la entrada en la tabla de símbolos asociada con el nombre.

3. Otros tipos de nodo. Estos últimos pueden ser ignorados desde el punto de vista del análisis de ámbito

La asignación de ámbito se implanta a través de atributos que se añaden a las nodos de uso y a los nodos ámbito.

Algunos de los nombres de dichos atributos pueden ser especificados mediante los parámetros de `new`. En concreto:

- `SCOPE_NAME` es el nombre del atributo de la instancia que contendrá la referencia al nodo 'bloque' en el que ocurre esta instancia. Si no se especifica toma el valor `scope`.

  En el ejemplo:

  ```
  $ids = Parse::Eyapp::Scope->new(
           SCOPE_NAME => 'block',
           ENTRY_NAME => 'info',
           SCOPE_DEPTH => 'depth',
  );
  ```

  cada nodo asociado con una instancia de uso tendrá un atributo con clave `block` que será una referencia al nodo `BLOCK` en el que fué declarado el identificador.

- `ENTRY_NAME` es el nombre del atributo de la instancia que contendrá la referencia a la entrada de símbolos que corresponde a esta instancia. Si no se especifica tomará el valor `entry`.

  En el ejemplo que estamos trabajando cada nodo `BLOCK` tendrá un atributo `symboltable` que será una referencia a la tabla de símbolos asociada con el bloque. Dado que el atributo `block` de un nodo `$n` asociado con una instancia de un identificador `$id` apunta al nodo `BLOCK` en el que se define, siempre sería posible acceder a la entrada de la tabla de símbolos mediante el código:

  ```
  $n->{block}{symboltable}{$id}
  ```

  El atributo `entry` crea una referencia directa en el nodo:

  ```
  $n->{entry} = $n->{block}{symboltable}{$id}
  ```

de esta forma es mas directo acceder a la entrada de símbolos de una instancia de uso de un identificador.

Obviamente el atributo `entry` no tiene sentido en aquellos casos en que el análisis de ámbito no requiere de la presencia de tablas de símbolos, como es el caso del análisis de ámbito de las sentencias de cambio de flujo en bucles:

```
$loops = Parse::Eyapp::Scope->new(
         SCOPE_NAME => 'exits',
);
```

- `SCOPE_DEPTH` es el nombre del atributo del nodo ámbito (nodo bloque) y contiene la profundidad de anidamiento del ámbito. Es opcional. Si no se especifica no será guardado.

El Método `begin_scope`

Este método debe ser llamado cada vez que se entra en una nueva región de ámbito. Parse::Eyapp::Scope asume un esquema de ámbitos léxicos anidados como ocurre en la mayoría de los lenguajes de programación: se supone que todos los nodos declarados mediante llamadas al método `scope_instance` entre dos llamadas consecutivas a `begin_scope` y `end_scope` definen la región del ámbito. Por ejemplo, en el analisis de ámbito de SimpleC llamamos a `begin_scope` cada vez que se entra en una definición de función:

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne '/^funcDef:/,/^;$/p' Scope.eyp | cat
  1  funcDef:
  2      $ID
  3         {
  4           $ids->begin_scope();
  5         }
  6      '('  $params  ')'
  7      $block
  8        {
 ..           ........................................
 35        }
 36  ;
```

y cada vez que se entra en un bloque:

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne '/^block:/,/^;$/p' Scope.eyp | cat -n
  1  block:
  2      '{'.bracket
  3         { $ids->begin_scope(); }
  4       declaration<%name DECLARATIONS *>.decs statement<%name STATEMENTS *>.sts '}'
  5         {
 25           ................
 26         }
 27
 28  ;
```

En el caso del análisis de ámbito de bucles la entrada en un bucle crea una nueva región de ámbito:

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne '/^loopPrefix:/,/^;$/p' Scope.eyp | c
  1  loopPrefix:
  2      $WHILE '(' expression ')'
  3        {
  4          $loops->begin_scope;
  5          $_[3]->{line} = $WHILE->[1];
```

```
     $_[3]
    }
;
```

El Método end_scope

En el código que sigue el método `end_scope` es llamado con tres argumentos:

```
my ($nodec, $dec) = $ids->end_scope($st, $block, 'type');
```

El significado de estos argumentos es el siguiente:

1. Una referencia a un hash `$st`. Este hash es la tabla de símbolos asociada con el bloque actual. Se asume que la clave de entrada de un nodo `$n` del árbol se obtiene mediante una llamada al método del nodo `key` definido por el programador: `$st->{$n->key}`. Se asume también que los valores del hash son una referencia a un hash conteniendo los atributos asociados con la clave `$n->key`.
2. Una referencia al nodo bloque `$block` o nodo de ámbito asociado con el hash. En nuestro ejemplo del análisis de ámbito de los identificadores las instancias de identificadores declarados en el presente bloque seran decorados con un atributo `block` que apunta a dicho nodo. El nombre es `block` porque así se específico en la llamada a `new`:

   ```
   $ids = Parse::Eyapp::Scope->new(
       SCOPE_NAME => 'block',
       ENTRY_NAME => 'info',
       SCOPE_DEPTH => 'depth',
   );
   ```

3. Los argumentos adicionales como `'type'` que sean pasados a `end_scope` son interpretados como claves del hash apuntado por su entrada en la tabla de símbolos `$n->key`. Para cada uno de esos argumentos se crean referencias directas en el nodo `$n` a los mismos:

   ```
   $n->{type} = $st->{$n->key}{type}
   ```

La llamada `$ids->end_scope($st, $block, 'type')` ocurre después del análisis de todo el bloque de la función:

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne '/^funcDef:/,/^;$/p' Scope.eyp | cat
funcDef:
  $ID
    {
     $ids->begin_scope();
    }
  '('  $params  ')'
  $block
   {
     my $st = $block->{symboltable};
     my @decs = $params->children();
     $block->{parameters} = [];
     while (my ($bt, $id, $arrspec) = splice(@decs, 0, 3)) {
      ..............................................
     }
     $block->{function_name} = $ID;
     $block->type("FUNCTION");

```

```
         my ($nodec, $dec) = $ids->end_scope($st, $block, 'type');

         # Type checking: add a direct pointer to the data-structure
         # describing the type
         $_->{t} = $type{$_->{type}} for @$dec;

         return $block;
      }
;
```

Todos los nodos `$n` del árbol que fueron declarados como instancias mediante llamadas al método `scope_instance` desde la última llamada a `begin_scope` son buscados en el hash referenciado por `$st`. Si la clave `$n->key` asociada con el nodo `$n` se encuentra entre las claves del hash (esto es, `exists $st->{$n->key}`) el nodo se anota como *declarado*. Los nodos-instancia para los cuales no existe una entrada en la tabla de símbolos se consideran *no declarados*.

Cuando un nodo se determina como declarado se establecen los atributos de declaración:

- Una referencia a su entrada en la tabla de símbolos (cuyo nombre fue establecido mediante el parámetro `ENTRY_NAME` de `new`)

  ```
  $n->{$ids->{ENTRY_NAME}} = $st->{$n->key}
  ```

  Si por ejemplo el nodo siendo usado se corresponde con el uso de una variable de nombre `a`, entonces `$n->key` deberá retornar `a` y la asignación anterior, asumiendo la declaración previa del manejador de ámbito `$ids`, sería:

  ```
  $n->{info} = $st->{a}
  ```

- Una referencia al bloque en el que fué declarado (el nombre del atributo será el establecido en `SCOPE_NAME`).

  ```
  $n->{$ids->{SCOPE_NAME}} = $block
  ```

  Si por ejemplo el nodo siendo usado se corresponde con el uso de una variable de nombre `a`, la asignación anterior, asumiendo la declaración previa del manejador de ámbito `$ids`, sería:

  ```
  $n->{block} = $block
  ```

  donde `$block` es el nodo asociado con el bloque en el que fue declarado `$a`.

- Cualesquiera argumentos adicionales - como `'type'` en el ejemplo - que sean pasados a `end_scope` son interpretados como claves del hash apuntado por su entrada en la tabla de símbolos `$st->{$n->key}`. Para cada uno de esos argumentos se crean referencias directas en el nodo `$n` a los mismos.

  ```
  $n->{type} = $st->{$n->key}{type}
  ```

  esto es:

  ```
  $n->{type} = $st->{a}{type}
  ```

Se supone que antes de la llamada a `end_scope` las declaraciones de los diferentes objetos han sido procesadas y la tabla hash ha sido rellenada con las mismas.

Los nodos `$n` que no aparezcan entre los declarados en este ámbito son apilados en la esperanza de que hayan sido declarados en un ámbito superior.

Para un ejemplo de uso véase la línea 28 en el código anterior. El tercer argumento `type` indica que para cada instancia de variable global que ocurre en el ámbito de `program` queremos que se cree una referencia desde el nodo a su entrada `type` en la tabla de símbolos. De este modo se puede conseguir un acceso mas rápido al tipo de la instancia si bien a costa de aumentar el consumo de memoria.

En un contexto de lista `end_scope` devuelve un par de referencias. La primera es una referencia a la lista de instancias/nodos del ámbito actual que no aparecen como claves del hash. La segunda a los que aparecen.

En un contexto escalar devuelve la lista de instancias/nodos no declarados.

El Método `key` para Nodos con Ámbito

El programador deberá proveer a cada clase de nodo (subclases de `Parse::Eyapp::Node`) que pueda ser instanciado/usado en un ámbito o bloque de un método `key`. En nuestro caso los nodos del tipo `VAR`, `VARARRAY` y `FUNCTIONCALL` son los que pueden ser usados dentro de expresiones y sentencias.

El método `key` se usa para computar el valor de la clave de entrada en el hash para esa clase de nodo.

Para la tabla de símbolos de los identificadores en SimpleC necesitamos definir el método `key` para los nodos `VAR`, `VARARRAY` y `FUNCTIONCALL`:

```
sub VAR::key {
  my $self = shift;

  return $self->child(0)->{attr}[0];
}

*VARARRAY::key = *FUNCTIONCALL::key = \&VAR::key;
```

Si tiene dudas repase la definición de `Variable` en la descripción de la gramática en la página 422:

```
Variable:
    %name VAR
    ID
  | %name  VARARRAY
    $ID ('[' binary ']') <%name INDEXSPEC +>
;
```

y el subárbol para la asignación de `a = 2` en la página 430.

```
ASSIGN(
  VAR(
    TERMINAL[a:5]
  ),
  INUM(
    TERMINAL[2:5]
  )
) # ASSIGN,
```

El Modo de Llamada Simplificado a `end_scope`

En el caso de un análisis de ámbito simple como es el análisis de ámbito de los bucles no se requiere de la presencia de una tabla de símbolos. El único argumento de `end_scope` es la referencia al nodo que define el ámbito.

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne '/^statement:/,/^;$/p' Scope.eyp | ca
  1  statement:
  2      expression ';' { $_[1] }
```

```
  | ';'
  | %name BREAK
    $BREAK ';'
      {
        my $self = shift;
        my $node = $self->YYBuildAST(@_);
        $node->{line} = $BREAK->[1];
        $loops->scope_instance($node);
        return $node;
      }
  | %name CONTINUE
     $CONTINUE ';'
      {
        ...............................
      }
  .....................................
  | %name WHILE
    $loopPrefix statement
      {
        my $self = shift;
        my $wnode = $self->YYBuildAST(@_);
        $wnode->{line} = $loopPrefix->{line};
        my $breaks = $loops->end_scope($wnode);
        return $wnode;
      }
;
```

El Método `scope_instance`

Este método del objeto Parse::Eyapp::Scope empuja el nodo que se le pasa como argumento en la cola de instancias del manejador de ámbito. El nodo se considerará una ocurrencia de un objeto dentro del ámbito actual (el que comienza en la última ejecución de `begin_scope`). En el compilador de `SimpleC` debemos hacer:

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne '/^Primary:/,/^;$/p' Scope.eyp | cat
Primary:
    %name INUM
    INUM
  | %name CHARCONSTANT
    CHARCONSTANT
  | $Variable
      {
        $ids->scope_instance($Variable);
        return $Variable
      }
  | '(' expression ')' { $_[2] }
  | $function_call
      {
        $ids->scope_instance($function_call);
        return $function_call  # bypass
      }
;
```

Otros lugares en los que ocurren instancias de identificadores son las asignaciones:

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne '/^binary:/,/^;$/p' Scope.eyp | cat -
binary:
    Unary { $_[1] }
  | %name PLUS
    binary '+' binary
  ..................
  | %name ASSIGN
    $Variable '=' binary
      {
        goto &declare_instance_and_build_node;
      }
  | %name PLUSASSIGN
    $Variable '+=' binary
      {
        goto &declare_instance_and_build_node;
      }
  | %name MINUSASSIGN
    $Variable '-=' binary
      {
        goto &declare_instance_and_build_node;
      }
  | %name TIMESASSIGN
    $Variable '*=' binary
      {
        goto &declare_instance_and_build_node;
      }
  | %name DIVASSIGN
    $Variable '/=' binary
      {
        goto &declare_instance_and_build_node;
      }
  | %name MODASSIGN
    $Variable '%=' binary
      {
        goto &declare_instance_and_build_node;
      }
;
```

Como indica su nombre, la función `declare_instance_and_build_node` declara la instancia y crea el nodo del AST:

```
sub declare_instance_and_build_node {
  my ($parser, $Variable) = @_[0,1];

  $ids->scope_instance($Variable);
  goto &Parse::Eyapp::Driver::YYBuildAST;
}
```

En el caso del análisis de ámbito en bucles las instancias ocurren en las sentencias de `break` y `continue`:

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne '/^statement:/,/^;$/p' Scope.eyp | ca
statement:
    expression ';' { $_[1] }
  | ';'
```

```
  | %name BREAK
    $BREAK ';'
      {
        my $self = shift;
        my $node = $self->YYBuildAST(@_);
        $node->{line} = $BREAK->[1];
        $loops->scope_instance($node);
        return $node;
      }
  | %name CONTINUE
     $CONTINUE ';'
      {
        my $self = shift;
        my $node = $self->YYBuildAST(@_);
        $node->{line} = $CONTINUE->[1];
        $loops->scope_instance($node);
        return $node;
      }
  .....................................
  | %name WHILE
    $loopPrefix statement
      {
        my $self = shift;
        my $wnode = $self->YYBuildAST(@_);
        $wnode->{line} = $loopPrefix->{line};
        my $breaks = $loops->end_scope($wnode);
        return $wnode;
      }
;
```

Cálculo del Ámbito en Bloques El modo de uso se ilustra en el manejo de los bloques

```
block: '{' declaration * statement * '}'
```

La computación del ámbito requiere las siguientes etapas:

1. Es necesario introducir una acción intermedia (línea 3) para indicar que una "llave abrir" determina el comienzo de un bloque.
2. Durante las sucesivas visitas a `declaration` construímos tablas de símbolos que son mezcladas en las líneas 8-15.
3. Todas las instancias de nodos-nombre que ocurren cuando se visitan los hijos de `statements` son declaradas como instanciaciones con `scope_instance` (véanse los acciones semánticas para `Primary` en la página 459).
4. La llamada a `end_scope` de la línea 19 produce la computación parcial de la función de asignación de ámbito para los nodos/nombre que fueron instanciados en este bloque.

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne '/^block:/,/^;$/p' Scope.eyp | cat -n
 1  block:
 2      '{'.bracket
 3         { $ids->begin_scope(); }
 4       declaration<%name DECLARATIONS *>.decs statement<%name STATEMENTS *>.sts '}'
 5         {
 6           my %st;
```

```

         for my $lst ($decs->children) {

             # control duplicated declarations
           my $message;
           die $message if $message = is_duplicated(\%st, $lst);

           %st = (%st, %$lst);
         }
         $sts->{symboltable} = \%st;
         $sts->{line} = $bracket->[1];
         $sts->type("BLOCK") if (%st);
         my ($nondec, $dec) = $ids->end_scope(\%st, $sts, 'type');

         # Type checking: add a direct pointer to the data-structure
         # describing the type
         $_->{t} = $type{$_->{type}} for @$dec;

         return $sts;
       }


;
```

Actualmente las acciones intermedias tal y como la que se ven en el código anterior se usan habitualmente para producir efectos laterales en la construcción del árbol. No dan lugar a la inserción de un nodo. Esto es, la variable auxiliar/temporal generada para dar lugar a la acción intermedia no es introducida en el árbol generado.

La Jerarquía de Bloques

Para completar el análisis de ámbito queremos decorar cada nodo `BLOCK` con un atributo `fatherblock` que referencia al nodo bloque que lo contiene. Para esta fase usaremos el lenguaje `Treeregexp`. Las frases de este lenguaje permiten definir conjuntos de árboles. Cuando van en un fichero aparte los programas `Treeregexp` suelen tener la extensión `trg`. Por ejemplo, el siguiente programa

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ cat -n Trans.trg
     1  /* Scope Analysis */
     2  blocks:  /BLOCK|FUNCTION|PROGRAM/
     3
     4  retscope: /FUNCTION|RETURN/
     5
     6  loop_control: /BREAK|CONTINUE|WHILE/
```

define tres expresiones árbol a las que nos podremos referir en el cliente mediante las variables `$blocks`, `$retscope` y `loops_control`. La primera expresión árbol tiene por nombre `blocks` y casará con cualesquiera árboles de las clases `BLOCK`, `FUNCTION` o `PROGRAM`. Es necesario compilar el programa `Treeregexp`:

```
l@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ treereg -m Simple::Scope Trans
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ ls -ltr | tail -2
-rw-r--r-- 1 pl users   122 2007-12-10 11:49 Trans.trg
-rw-r--r-- 1 pl users  1544 2007-12-10 11:49 Trans.pm
```

El módulo generado `Trans.pm` contendrá una subrutina por cada expresión regular árbol en el programa `Treeregexp`. La subrutina devuelve cierto si el nodo que se le pasa como primer argumento casa con la expresión árbol. La subrutina espera como primer argumento el nodo, como segundo argumento el padre del nodo y como tercero el objeto `Parse::Eyapp::YATW` que encapsula la transformación:

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne ’19,31p’ Trans.pm
  sub blocks {
    my $blocks = $_[3]; # reference to the YATW pattern object
    my $W;

    {
      my $child_index = 0;

      return 0 unless ref($W = $_[$child_index]) =~ m{\bBLOCK\b|\bFUNCTION\b|\bPROGRAM\b}x;

    } # end block of child_index
    1;

  } # end of blocks
```

No solo se construye una subrutina `blocks` sino que en el espacio de nombres correspondiente se crean objetos `Parse::Eyapp::YATW` por cada una de las transformaciones especificadas en el programa `trg`, como muestra el siguiente fragmento del módulo conteniendo el código generado por `treeregexp`:

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne ’16p’ Trans.pm
our @all = ( our $blocks, our $retscope, our $loop_control, ) =
  Parse::Eyapp::YATW->buildpatterns(
    blocks => \&blocks,
    retscope => \&retscope,
    loop_control => \&loop_control,
  );
```

Estos objetos `Parse::Eyapp::YATW` estan disponibles en el programa eyapp ya que hemos importado el módulo:

```
l@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne ’1,/^ *$/p’ Scope.eyp | cat -n
 1  /*
 2  File: lib/Simple/Scope.eyp
 3  Full Scope Analysis
 4  Test it with:
 5    lib/Simple/
 6    eyapp -m Simple::Scope Scope.eyp
 7    treereg -nonumbers -m Simple::Scope Trans
 8    script/
 9    usescope.pl prueba12.c
10  */
11  %{
12  use strict;
13  use Data::Dumper;
14  use List::MoreUtils qw(firstval lastval);
15  use Simple::Trans;
16  use Parse::Eyapp::Scope qw(:all);
```

Es por eso que, una vez realizadas las fases de análisis léxico, sintáctico y de ámbito podemos usar el objeto `$blocks` y el método `m` (por `matching`) de dicho objeto (véase la línea 14 en el código de `compile`):

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne ’/sub comp/,/^}/p’ Scope.eyp | cat -n
 1  sub compile {
 2   my($self)=shift;
 3
```

```
  my ($t);

  $self->YYData->{INPUT} = shift;

  $t = $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
                       #yydebug => 0x1F
      );

  # Scope Analysis: Block Hierarchy
  our $blocks;
  my @blocks = $blocks->m($t);
  $_->node->{fatherblock} = $_->father->{node} for (@blocks[1..$#blocks]);

  .................................

  return $t;
}
```

El método `m` para árboles y expresiones regulares árbol `YATW` funciona de manera parecida al método `m` para cadenas y expresiones regulares convencionales. El resultado de un matching en árboles es un árbol. Si se trabaja en un contexto de lista es una lista de árboles.

La llamada `$blocks->m($t)` permite la búsqueda de los nodos de `$t` que casan con la expresión regular árbol para `blocks`. En un contexto de lista `m` devuelve una lista con nodos del tipo `Parse::Eyapp::Node::Match` que referencian a los nodos que han casado. Los nodos `Parse::Eyapp::Node::Match` son a su vez nodos (heredan de) `Parse::Eyapp::Node`. Los nodos aparecen en la lista retornada en orden primero profundo de recorrido del árbol `$t`.

Los nodos en la lista se estructuran según un árbol (atributos `children` y `father`) de manera que el padre de un nodo `$n` del tipo `Parse::Eyapp::Node::Match` es el nodo `$f` que referencia al inmediato antecesor en el árbol que ha casado.

Un nodo `$r` de la clase `Parse::Eyapp::Node::Match` dispone de varios atributos y métodos:

- El método `$r->node` retorna el nódo del árbol `$t` que ha casado
- El método `$r->father` retorna el nodo padre en el árbol the matching. Se cumple que `$r->father->node` es una referencia al antepasado mas cercano en `$t` de `$r->node` que casa con el patrón árbol.
- El método `$r->coord` retorna las coordenadas del nódo del árbol que ha casado usando una notación con puntos. Por ejemplo la coordenada `".1.3.2"` denota al nodo `$t->child(1)->child(3)->child(2)`, siendo `$t` la raíz del árbol de búsqueda.
- El método `$r->depth` retorna la profundidad del nódo del árbol que ha casado

En un contexto escalar `m` devuelve el primer elemento de la lista de nodos `Parse::Eyapp::Node::Match`.

```
pl@nereida:~/Lbook/code/Simple-Scope/script$ perl -wd usescope.pl blocks.c 2

Loading DB routines from perl5db.pl version 1.28
Editor support available.

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

main::(usescope.pl:6):  my $filename = shift || die "Usage:\n$0 file.c\n";
  DB<1> c Simple::Scope::compile

   1 test (int n)
   2 {
```

```
   3   int a;
   4
   5   while (1) {
   6     if (1>0) {
   7       int a;
   8       break;
   9     }
  10     else if (2>0){
  11       char b;
  12       continue;
  13     }
  14   }
  15 }
Simple::Scope::compile(Scope.eyp:784):
784:     my($self)=shift;
 DB<2> l 783,797
783     sub compile {
784==>   my($self)=shift;
785
786:     my ($t);
787
788:     $self->YYData->{INPUT} = shift;
789
790:     $t = $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
791                          #yydebug => 0x1F
792          );
793
794      # Scope Analysis: Block Hierarchy
795:     our $blocks;
796:     my @blocks = $blocks->m($t);
797:     $_->node->{fatherblock} = $_->father->{node} for (@blocks[1..$#blocks]);
  DB<3> c 797
Simple::Scope::compile(Scope.eyp:797):
797:     $_->node->{fatherblock} = $_->father->{node} for (@blocks[1..$#blocks]);
  DB<4> x Parse::Eyapp::Node->str(@blocks)
0  '
Match[[PROGRAM:0:blocks]](
  Match[[FUNCTION:1:blocks:test]](
    Match[[BLOCK:6:blocks:10:4]],
    Match[[BLOCK:5:blocks:6:4]]
  )
)'
1  '
Match[[FUNCTION:1:blocks:test]](
  Match[[BLOCK:6:blocks:10:4]],
  Match[[BLOCK:5:blocks:6:4]]
)'
2  '
Match[[BLOCK:5:blocks:6:4]]'
3  '
Match[[BLOCK:6:blocks:10:4]]'
```

La información que aparece en los nodos `Match` es como sigue:

- La clase del nodo al que referencia `PROGRAM`, `FUNCTION`, etc.

- La profundidad del nodo que ha casado en `$t`
- Los nombres de las transformaciones con las que casó el nodo
- Si el nodo tiene un método `info` la información asociada. En este caso la línea y la profundidad en los bloques.

```
 DB<5> p $t->str

PROGRAM^{0}(
  FUNCTION[test]^{1}(                                     #  test (int n) {
    WHILE(                                                #    while (1) {
      INUM(TERMINAL[1:5]),
      STATEMENTS(
        IFELSE(                                           #      if (1>0) {
          GT(INUM( TERMINAL[1:6]), INUM(TERMINAL[0:6])),
          BLOCK[6:4]^{2}(
            BREAK                                         #         break;
          ),                                              #      }
          IF(                                             #      else if (2>0){
            GT(INUM(TERMINAL[2:10]), INUM(TERMINAL[0:10])),
            BLOCK[10:4]^{3}(
              CONTINUE                                    #         continue
            )                                             #      }
          )
        )
      )
    )
  )
)
....
  DB<6> x map {$_->coord} @blocks
0  ''
1  '.0'
2  '.0.0.1.0.1'
3  '.0.0.1.0.2.1'
  DB<7>  p $t->descendant('.0.0.1.0.2.1')->str

BLOCK[10:4]^{0}(
  CONTINUE
)
---------------------------
0)
Symbol Table of block at line 10
$VAR1 = {
          'b' => {
                   'type' => 'CHAR',
                   'line' => 11
                 }
        };
 DB<8> x map {$_->depth} @blocks
0  0
1  1
2  5
```

```
3  6
  DB<9> x  map {ref($_->node) } @blocks
0  'PROGRAM'
1  'FUNCTION'
2  'BLOCK'
3  'BLOCK'
  DB<10> x  map {ref($_->father) } @blocks
0  ''
1  'Parse::Eyapp::Node::Match'
2  'Parse::Eyapp::Node::Match'
3  'Parse::Eyapp::Node::Match'
  DB<11> x  map {ref($_->father->node) } @blocks[1..3]
0  'PROGRAM'
1  'FUNCTION'
2  'FUNCTION'
  DB<12> x  $blocks[2]->father->node->{function_name}
0  ARRAY(0x86ed84c)
   0  'test'
   1  1
```

El Método `names`

La clase `Parse::Eyapp::Node::Match` dispone además de otros métodos para el caso en que se usan varios patrones en la misma búsqueda. Por ejemplo, el método `$r->names` retorna una referencia a los nombres de los patrones que han casado con el nodo. En el ejemplo que sigue aparecen tres transformaciones `fold`, `zxw` y `wxz`. La llamada a `m` de la línea 19 toma una forma diferente. ahora `m` es usado como método del árbol `$t` y recibe como argumentos la lista con los objetos expresiones regulares árbol (`Parse::Eyapp::YATW`).

```
pl@nereida:~/LEyapp/examples$ cat -n m2.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use Rule6;
 4  use Parse::Eyapp::Treeregexp;
 5
 6  Parse::Eyapp::Treeregexp->new( STRING => q{
 7    fold: /TIMES|PLUS|DIV|MINUS/(NUM, NUM)
 8    zxw: TIMES(NUM($x), .) and { $x->{attr} == 0 }
 9    wxz: TIMES(., NUM($x)) and { $x->{attr} == 0 }
10  })->generate();
11
12  # Syntax analysis
13  my $parser = new Rule6();
14  my $input = "0*0*0";
15  my $t = $parser->Run(\$input);
16  print "Tree:",$t->str,"\n";
17
18  # Search
19  my $m = $t->m(our ($fold, $zxw, $wxz));
20  print "Match Node:\n",$m->str,"\n";
```

La primera expresión regular árbol `fold` casa con cualquier árbol tal que la clase del nodo raíz case con la expresión regular (clásica) `/TIMES|PLUS|DIV|MINUS/` y tenga dos hijos de clase `NUM`.

La segunda expresión regular `zxw` (por *zero times whatever*) casa con cualquier árbol tal que la clase del nodo raíz es `TIMES` y cuyo primer hijo pertenece a la clase `NUM`. Se debe cumplir además que el hijo del nodo `NUM` (el nodo `TERMINAL` proveído por el analizador léxico) debe tener su atributo `attr` a cero. La notación `NUM($x)` hace que en `$x` se almacene una referencia al nodo hijo de `NUM`. La expresión regular árbol `wxz` es la simétrica de `zxw`.

Cuando se ejecuta, el programa produce la salida:

```
pl@nereida:~/LEyapp/examples$ m2.pl
Tree:TIMES(TIMES(NUM(TERMINAL),NUM(TERMINAL)),NUM(TERMINAL))
Match Node:
Match[[TIMES:0:wxz]](Match[[TIMES:1:fold,zxw,wxz]])
```

5.7. Resultado del Análisis de Ámbito

Como se ha comentado en la sección 5.5 el volcado de los AST con `Data::Dumper` es insuficiente. Para la visualización utilizaremos el método `str` presentado en la sección 5.8.

Código de Apoyo a la Visualización

Sigue el código de apoyo a la visualización:

```
pl@nereida:~/Lbook/code/Simple-Scope/lib/Simple$ sed -ne '845,$p' Scope.eyp | cat -n
############## Debugging and Display
sub show_trees {
 my ($t) = @_;

 print Dumper $t if $debug > 1;
 $Parse::Eyapp::Node::INDENT = 2;
 $Data::Dumper::Indent = 1;
 print $t->str."\n";
}

$Parse::Eyapp::Node::INDENT = 1;
sub TERMINAL::info {
  my @a = join ':', @{$_[0]->{attr}};
  return "@a"
}

sub PROGRAM::footnote {
  return "Types:\n"
         .Dumper($_[0]->{types}).
         "Symbol Table of the Main Program:\n"
         .Dumper($_[0]->{symboltable})
}

sub FUNCTION::info {
  return $_[0]->{function_name}[0]
}

sub FUNCTION::footnote {
  my $text = '';
  $text .= "Symbol Table of $_[0]->{function_name}[0]\n" if $_[0]->type eq 'FUNCTION';
  $text .= "Symbol Table of block at line $_[0]->{line}\n" if $_[0]->type eq 'BLOCK';
  $text .= Dumper($_[0]->{symboltable});
  return $text;
}

```

```
sub BLOCK::info {
  my $info = "$_[0]->{line}:$_[0]->{depth}";
  my $fatherblock = $_[0]->{fatherblock};
    $info .= ":".$fatherblock->info
  if defined($fatherblock) and UNIVERSAL::can($fatherblock, 'info');
  return $info;
}

*BLOCK::footnote = \&FUNCTION::footnote;

sub VAR::info {
  return $_[0]->{type} if defined $_[0]->{type};
  return "No declarado!";
}

*FUNCTIONCALL::info = *VARARRAY::info = \&VAR::info;
```

Un Ejemplo con Errores de Ámbito

Utilizando este código obtenemos una descripción completa y concisa del árbol anotado. Si hay errores de ámbito se mostrarán.

```
pl@nereida:~/Lbook/code/Simple-Scope/script$ usescope.pl prueba27.c 2

   1 int a,b;
   2
   3 int f(char c) {
   4  a[2] = 4;
   5  {
   6    int d;
   7    d = a + b;
   8  }
   9  c = d * 2;
  10  return g(c);
  11 }
Identifier d not declared at line 9
Identifier g not declared at line 10
```

Mostrando el Resultado del Análisis de Ámbito

Si no hay errores se obtiene un listado enumerado del programa fuente, el árbol, la tabla de tipos y las tablas de símbolos:

```
pl@nereida:~/Lbook/code/Simple-Scope/script$  usescope.pl prueba25.c 2

   1 int a[20],b,e[10];
   2
   3 g() {}
   4
   5 int f(char c) {
   6 char d;
   7  c = 'X';
   8  e[b] = 'A'+c;
   9  {
  10    int d;
  11    d = d + b;
```

```
  12  }
  13  c = d * 2;
  14  return c;
  15 }
  16
  17
PROGRAM^{0}(
  FUNCTION[g]^{1},
  FUNCTION[f]^{2}(
    ASSIGN(
      VAR[CHAR](
        TERMINAL[c:7]
      ),
      CHARCONSTANT(
        TERMINAL['X':7]
      )
    ) # ASSIGN,
    ASSIGN(
      VARARRAY[A_10(INT)](
        TERMINAL[e:8],
        INDEXSPEC(
          VAR[INT](
            TERMINAL[b:8]
          )
        ) # INDEXSPEC
      ) # VARARRAY,
      PLUS(
        CHARCONSTANT(
          TERMINAL['A':8]
        ),
        VAR[CHAR](
          TERMINAL[c:8]
        )
      ) # PLUS
    ) # ASSIGN,
    BLOCK[9:3:f]^{3}(
      ASSIGN(
        VAR[INT](
          TERMINAL[d:11]
        ),
        PLUS(
          VAR[INT](
            TERMINAL[d:11]
          ),
          VAR[INT](
            TERMINAL[b:11]
          )
        ) # PLUS
      ) # ASSIGN
    ) # BLOCK,
    ASSIGN(
      VAR[CHAR](
        TERMINAL[c:13]
```

```
      ),
      TIMES(
        VAR[CHAR](
          TERMINAL[d:13]
        ),
        INUM(
          TERMINAL[2:13]
        )
      ) # TIMES
    ) # ASSIGN,
    RETURN(
      VAR[CHAR](
        TERMINAL[c:14]
      )
    ) # RETURN
  ) # FUNCTION
) # PROGRAM
---------------------------
0)
Types:
$VAR1 = {
  'A_10(INT)' => bless( {
    'children' => [
      bless( {
        'children' => []
      }, 'INT' )
    ]
  }, 'A_10' ),
  'F(X_1(CHAR),INT)' => bless( {
    'children' => [
      bless( {
        'children' => [
          bless( {
            'children' => []
          }, 'CHAR' )
        ]
      }, 'X_1' ),
      $VAR1->{'A_10(INT)'}{'children'}[0]
    ]
  }, 'F' ),
  'CHAR' => $VAR1->{'F(X_1(CHAR),INT)'}{'children'}[0]{'children'}[0],
  'VOID' => bless( {
    'children' => []
  }, 'VOID' ),
  'INT' => $VAR1->{'A_10(INT)'}{'children'}[0],
  'A_20(INT)' => bless( {
    'children' => [
      $VAR1->{'A_10(INT)'}{'children'}[0]
    ]
  }, 'A_20' ),
  'F(X_0(),INT)' => bless( {
    'children' => [
      bless( {
```

```
        'children' => []
      }, 'X_0' ),
      $VAR1->{'A_10(INT)'}{'children'}[0]
    ]
  }, 'F' )
};
Symbol Table of the Main Program:
$VAR1 = {
  'e' => {
    'type' => 'A_10(INT)',
    'line' => 1
  },
  'a' => {
    'type' => 'A_20(INT)',
    'line' => 1
  },
  'b' => {
    'type' => 'INT',
    'line' => 1
  },
  'g' => {
    'type' => 'F(X_0(),INT)',
    'line' => 3
  },
  'f' => {
    'type' => 'F(X_1(CHAR),INT)',
    'line' => 5
  }
};

---------------------------
1)
Symbol Table of g
$VAR1 = {};

---------------------------
2)
Symbol Table of f
$VAR1 = {
  'c' => {
    'type' => 'CHAR',
    'param' => 1,
    'line' => 5
  },
  'd' => {
    'type' => 'CHAR',
    'line' => 6
  }
};

---------------------------
3)
Symbol Table of block at line 9
```

```
$VAR1 = {
  'd' => {
    'type' => 'INT',
    'line' => 10
  }
};

pl@nereida:~/Lbook/code/Simple-Scope/script$
```

Observe como se anotan las referencias a pie de árbol usando la notación `^{#}`. La salida se completa con las *notas a pie de árbol*.

5.8. Usando el Método `str` para Analizar el Árbol

El Método `str` de los Nodos

Para dar soporte al análisis de los árboles y su representación, `Parse::Eyapp` provee el método `str`. El siguiente ejemplo con el depurador muestra el uso del método `str`. El método `Parse::Eyapp::Node::str` retorna una cadena que describe como término el árbol enraizado en el nodo que se ha pasado como argumento.

El Método `info`

El método `str` cuando visita un nodo comprueba la existencia de un método `info` para la clase del nodo. Si es así el método será llamado. Obsérvese como en las líneas 5 y 6 proveemos de métodos `info` a los nodos `TERMINAL` y `FUNCTION`. La consecuencia es que en la llamada de la línea 7 el término que describe al árbol es decorado con los nombres de funciones y los atributos y números de línea de los terminales.

Los nodos de la clase `Parse::Eyapp::Node:Match` vienen con un método `info` asociado el cual muestra el tipo del nodo apuntado, su profundidad y el nómbre del patrón que casó (línea 9 de la sesión).

Veamos un ejemplo que ilustra también algunos "trucos"de depuración. Se han introducido comentarios en el texto de salida:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code> perl -wd usesimple4.pl
```

la opción `-w` por warning y `-d` para activar el debugger

```
Loading DB routines from perl5db.pl version 1.28
Editor support available.
```

El mensaje `Editor support available` nos señala la presencia de los servicios de edición de la entrada. Si no sale el mensaje instale `Term::ReadLine::Gnu` en su máquina!

```
main::(usesimple4.pl:6):  $Data::Dumper::Indent = 1;
```

La información entre paréntesis indica el fichero en que estamos y la línea (`usesimple4.pl`)

```
DB<1> f Simple4.eyp  # <-- Quiero poner un break en Simple4.eyp,
```

El comando `f` es esencial cuando se usa `eyapp`. Le indicamos al debugger que queremos usar como fichero por defecto el de nuestra gramática. Los siguientes comandos se referirán a `Simple4.eyp`:

```
DB<2> b 654     # <--- break en la línea 654 de Simple4.eyp
DB<3> l 654,656 # <-- Se trata de las líneas en las que se establece
                #     la jerarquía de bloques
654:b        my @blocks = $SimpleTrans::blocks->m($t);
655:            $_->node->{fatherblock} = $_->father->{node} \
             for (@blocks[1..$#blocks]);
```

```
656:         $Data::Dumper::Deepcopy = 1;
DB<4> c    # c indica que se continúa la ejecución del programa
           # hasta encontrar un "break"
```

La ejecución del programa continúa produciendo la salida del fuente hasta encontrar el punto de break:

```
****************
char d,e[1][2];
char f() {
  int c[2];
  char d;

  return d;
}
Simple4::Run(Simple4.eyp:654):       my @blocks = $SimpleTrans::blocks->m($t);
DB<4> n   # <--- next ejecuta la siguiente instrucción. Si es una sub no se entra.
Simple4::Run(Simple4.eyp:655):       $_->node->{fatherblock} = $_->father->{node} \
                                   for (@blocks[1..$#blocks]);
DB<4>     # La simple pulsación de un retorno de carro
          # hace que se repita la última instrucción s o n
Simple4::Run(Simple4.eyp:655):       $_->node->{fatherblock} = $_->father->{node} \
                                   for (@blocks[1..$#blocks]);
DB<4>
Simple4::Run(Simple4.eyp:656):       $Data::Dumper::Deepcopy = 1;
DB<4>
Simple4::Run(Simple4.eyp:657):       print Dumper $t;
DB<4> x $t->str # El método str para los objetos Parse::Eyapp::Node devuelve
                # una descripción del árbol
0  'PROGRAM(FUNCTION(RETURN(TERMINAL,VAR(TERMINAL))))'
DB<5> sub TERMINAL::info { @a = join ':', @{$_[0]->{attr}}; "[@a]"}
DB<6> sub FUNCTION::info { "[".$_[0]->{function_name}[0]."]" }
DB<7> x $t->str
0  'PROGRAM(FUNCTION[f](RETURN(VAR(TERMINAL[d:6]))))'
      # La variable @Parse::Eyapp::Node::PREFIXES Controla que
      # prefijos se deben eliminar en la presentación
DB<8> p @Parse::Eyapp::Node::PREFIXES
Parse::Eyapp::Node::
DB<9> x $blocks[0]->str
0  'Match[PROGRAM:0:blocks](Match[FUNCTION:1:blocks:[f]])'
   # La información de un nodo Match es la
   # terna [type:depth:nombre de la treeregexp]
```

Modo de uso de `str`

El método `str` ha sido concebido como una herramienta de ayuda a la depuración y verificación de los programas árbol. La metodología consiste en incorporar las pequeñas funciones `info` al programa en el que trabajamos. Así en `Simple4.eyp` añadimos el siguiente código:

```
640  sub Run {
641   my($self)=shift;
...   ................
663  }
664
665  sub TERMINAL::info {
666    my @a = join ':', @{$_[0]->{attr}};
```

```
  return "[@a]"
}

sub FUNCTION::info {
  return "[".$_[0]->{function_name}[0]."]"
}

sub BLOCK::info {
  return "[".$_[0]->{line}."]"
}
```

Un ejemplo de Salida con Bloques Anidados

En el código mostrado no se ha realizado la optimización consistente en no incluir en el árbol de bloques a aquellos bloques cuya sección de declaraciones es vacía. Asi podemos estudiar la estructura de datos resultante para un pequeño programa como el que sigue a continuación.

Cuadro 5.1: Representación de AST con `str`

Programa

```
 1 char d0;
 2 char f() {
 3   {
 4     {}
 5   }
 6   {
 7     { }
 8   }
 9   {
10     {{}}
11   }
12 }

13 g() {
14  {}
15  {
16    {}
17  }
18  {}
19 }
```

`$t->str`

```
PROGRAM(
    FUNCTION[f](
      BLOCK[3](
        BLOCK[4]
      ),
      BLOCK[6](
        BLOCK[7]),
        BLOCK[9](
          BLOCK[10](
            BLOCK[10]
          )
        )
    ),
    FUNCTION[g](
      BLOCK[14],
      BLOCK[15](
        BLOCK[16]
      ),
      BLOCK[18]
    )
)
```

`$blocks[0]->str`

```
Match[PROGRAM:0:blocks](
  Match[FUNCTION:1:blocks:[f]](
    Match[BLOCK:2:blocks:[3]](
      Match[BLOCK:3:blocks:[4]]
      ),
    Match[BLOCK:2:blocks:[6]](
      Match[BLOCK:3:blocks:[7]]
      ),
    Match[BLOCK:2:blocks:[9]](
      Match[BLOCK:3:blocks:[10]](
        Match[BLOCK:4:blocks:[10]]
      )
    )
  ),
  Match[FUNCTION:1:blocks:[g]](
    Match[BLOCK:2:blocks:[14]],
    Match[BLOCK:2:blocks:[15]](
      Match[BLOCK:3:blocks:[16]]
    ),
    Match[BLOCK:2:blocks:[18]]
  )
)
```

La tabla 5.1 muestra el fuente y el resultado de `$blocks[0]->str` después de la llamada
`my @blocks = $SimpleTrans::blocks->m($t);`
Obsérvese que los nodos `Match` son referencias a los nodos actuales del árbol.

Usando el Depurador y `str` Para Ver los AST

Veamos un ejemplo mas de sesión con el depurador cuando la entrada proveída es el programa en la tabla 5.1:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code> eyapp Simple4 ; perl -wd usesimple4.pl
```

```
main::(usesimple4.pl:6):        $Data::Dumper::Indent = 1;
DB<1> f Simple4.eyp        #<-- establecemos el fichero de trabajo
DB<2> l 643                #<-- nos paramos antes de la lectura
643:       $self->YYData->{INPUT} = $_;
DB<3> b 643 $_=~ m{char d0} # <-- nos detendremos sólo si se trata de este programa
DB<4> c                     # <-- a correr!
..................          # salida correspondiente a los fuentes anteriores
Simple4::Run(Simple4.eyp:643):     $self->YYData->{INPUT} = $_;
DB<5> b Parse::Eyapp::Node::str # <-- Especificación completa de la subrutina en la que parar
DB<6> c
........................
Parse::Eyapp::Node::str(/home/pl/src/perl/YappWithDefaultAction/lib//Parse/Eyapp/Node.pm:543):
543:      my $self = CORE::shift; # root of the subtree
DB<7> L    # <-- Listar todos los breakpoints
/home/pl/src/perl/YappWithDefaultAction/lib//Parse/Eyapp/Node.pm:
 543:     my $self = CORE::shift; # root of the subtree
   break if (1)
Simple4.eyp:
 643:      $self->YYData->{INPUT} = $_;
   break if ($_=~ m{char d0})
DB<8> B *    # <-- Eliminar todos los breakpoints
Deleting all breakpoints...
DB<9> b 654  # <-- detengámonos después del matching árbol
DB<10> c
Simple4::Run(Simple4.eyp:654):       $_->node->{fatherblock} = $_->father->{node} for (@blocks
DB<4> x $Parse::Eyapp::Node::INDENT # <-- La variable INDENT controla el estilo en str
0  1
DB<11> x $t->str                   # <--- Salida con sangrado
0  '
PROGRAM(
  FUNCTION[f](
    BLOCK[3](
      BLOCK[4]
    ),
    BLOCK[6](
      BLOCK[7]
    ),
    BLOCK[9](
      BLOCK[10](
        BLOCK[10]
      )
    )
  ),
  FUNCTION[g](
    BLOCK[14],
    BLOCK[15](
      BLOCK[16]
    ),
    BLOCK[18]
  )
)'
DB<12> $Parse::Eyapp::Node::INDENT = 0 # <-- Estilo compacto. Por defecto
DB<13> x $t->str                       # <-- Todo en una línea
```

```
0  'PROGRAM(FUNCTION[f](BLOCK[3](BLOCK[4]),BLOCK[6](BLOCK[7]),BLOCK[9](BLOCK[10](BLOCK[10]))),
FUNCTION[g](BLOCK[14],BLOCK[15](BLOCK[16]),BLOCK[18]))'
DB<14> .                              # Donde estoy? Que línea se va a ejecutar?
Simple4::Run(Simple4.eyp:654): $_->node->{fatherblock} = $_->father->{node} for (@blocks[1..$#
DB<15> c 655      # <-- Ejecutemos hasta la siguiente
Simple4::Run(Simple4.eyp:655):       $Data::Dumper::Deepcopy = 1;
DB<16> x $blocks[0]->str   # <-- Veamos la forma del nodo de matching
0  'Match[PROGRAM:0:blocks](Match[FUNCTION:1:blocks:[f]](Match[BLOCK:2:
blocks:[3]](Match[BLOCK:3:blocks:[4]]),Match[BLOCK:2:blocks:[6]](Match[
BLOCK:3:blocks:[7]]),Match[BLOCK:2:blocks:[9]](Match[BLOCK:3:blocks:[1
0]](Match[BLOCK:4:blocks:[10]]))),Match[FUNCTION:1:blocks:[g]](Match[B
LOCK:2:blocks:[14]],Match[BLOCK:2:blocks:[15]](Match[BLOCK:3:blocks:[1
6]]),Match[BLOCK:2:blocks:[18]]))' # <-- Todo está en una línea
DB<17> $Parse::Eyapp::Node::INDENT = 1
DB<18> x $blocks[0]->str
0  '
Match[PROGRAM:0:blocks](
  Match[FUNCTION:1:blocks:[f]](
    Match[BLOCK:2:blocks:[3]](
      Match[BLOCK:3:blocks:[4]]
    ),
    Match[BLOCK:2:blocks:[6]](
      Match[BLOCK:3:blocks:[7]]
    ),
    Match[BLOCK:2:blocks:[9]](
      Match[BLOCK:3:blocks:[10]](
        Match[BLOCK:4:blocks:[10]]
      )
    )
  ),
  Match[FUNCTION:1:blocks:[g]](
    Match[BLOCK:2:blocks:[14]],
    Match[BLOCK:2:blocks:[15]](
      Match[BLOCK:3:blocks:[16]]
    ),
    Match[BLOCK:2:blocks:[18]]
  )
)'
```

Variables que Gobiernan la Conducta de `str`

Las variables de paquete que gobiernan la conducta de `str` y sus valores por defecto aparecen en la siguiente lista:

- `our @PREFIXES = qw(Parse::Eyapp::Node::)`

  La variable de paquete `Parse::Eyapp::Node::PREFIXES` contiene la lista de prefijos de los nombres de tipo que serán eliminados cuando `str` muestra el término. Por defecto contiene la cadena `'Parse::Eyapp::Node::'`. Es por esta razón que los nodos de matching que pertenecen a la clase `'Parse::Eyapp::Node::Match` son abreviados a la cadena `Match` (línea 9).

- `our $INDENT = 0` La variable `$Parse::Eyapp::Node::INDENT` controla el formato de presentación usado por `Parse::Eyapp::Node::str`:

  1. Si es `0` la representación es compacta.
  2. Si es 1 se usará un sangrado razonable.

3. Si es 2 cada paréntesis cerrar que este a una distancia mayor de `$Parse::Eyapp::Node::LINESEP` líneas será comentado con el tipo del nodo. Por defecto la variable `$Parse::Eyapp::Node::LINESEP` está a 4. Véase la página 481 para un ejemplo con `$INDENT` a 2 y `$Parse::Eyapp::Node::LINESEP` a 4.

- `our $STRSEP = ','`

  Separador de nodos en un término. Por defecto es la coma.

- `our $DELIMITER = '['`

  Delimitador de presentación de la información proveída por el método `info`. Por defecto los delimitadores son corchetes. Puede elegirse uno cualquiera de la lista:

  `'[', '{', '(', '<'`

  si se define como la cadena vacía o `undef` no se usarán delimitadores.

- `our $FOOTNOTE_HEADER = "\n---------------------------\n"`

  Delimitador para las notas a pie de página.

- `our $FOOTNOTE_SEP = ")\n"` Separador de la etiqueta/número de la nota al pie

- `our $FOOTNOTE_LEFT = '^{'`, `our $FOOTNOTE_RIGHT = '}'`

  Definen el texto que rodea al número de referencia en el nodo.

Footnotes

El siguiente ejemplo ilustra el manejo de *notas a pié de árbol* usando `str`. Se han definido los siguientes métodos:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code> sed -ne '677,$p' Simple6.eyp
$Parse::Eyapp::Node::INDENT = 1;
sub TERMINAL::info {
  my @a = join ':', @{$_[0]->{attr}};
  return "@a"
}

sub PROGRAM::footnote {
  return "Types:\n"
         .Dumper($_[0]->{types}).
         "Symbol Table:\n"
         .Dumper($_[0]->{symboltable})
}

sub FUNCTION::info {
  return $_[0]->{function_name}[0]
}

sub FUNCTION::footnote {
  return Dumper($_[0]->{symboltable})
}

sub BLOCK::info {
  return $_[0]->{line}
}
```

```
sub VAR::info {
  return $_[0]->{definition}{type} if defined $_[0]->{definition}{type};
  return "No declarado!";
}

*FUNCTIONCALL::info = *VARARRAY::info = \&VAR::info;
```

El resultado para el programa de entrada:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code> cat salida
int a,b;

int f(char c) {
 a[2] = 4;
 b[1][3] = a + b;
 c = c[5] * 2;
 return g(c);
}
```

produce una salida por `stderr`:

```
Identifier g not declared
```

y la siguiente descripción de la estructura de datos:

```
PROGRAM^{0}(
  FUNCTION{f}^{1}(
    ASSIGN(
      VARARRAY{INT}(
        TERMINAL{a:4},
        INDEXSPEC(
          INUM(
            TERMINAL{2:4}
          )
        )
      ),
      INUM(
        TERMINAL{4:4}
      )
    ),
    ASSIGN(
      VARARRAY{INT}(
        TERMINAL{b:5},
        INDEXSPEC(
          INUM(
            TERMINAL{1:5}
          ),
          INUM(
            TERMINAL{3:5}
          )
        )
      ),
      PLUS(
        VAR{INT}(
          TERMINAL{a:5}
        ),
```

```
        VAR{INT}(
          TERMINAL{b:5}
        )
      )
    ),
    ASSIGN(
      VAR{CHAR}(
        TERMINAL{c:6}
      ),
      TIMES(
        VARARRAY{CHAR}(
          TERMINAL{c:6},
          INDEXSPEC(
            INUM(
              TERMINAL{5:6}
            )
          )
        ),
        INUM(
          TERMINAL{2:6}
        )
      )
    ),
    RETURN(
      FUNCTIONCALL{No declarado!}(
        TERMINAL{g:7},
        ARGLIST(
          VAR{CHAR}(
            TERMINAL{c:7}
          )
        )
      )
    )
  )
)
---------------------------
0)
Types:
$VAR1 = {
  'F(X_1(CHAR),INT)' => bless( {
    'children' => [
      bless( {
        'children' => [
          bless( {
            'children' => []
          }, 'CHAR' )
        ]
      }, 'X_1' ),
      bless( {
        'children' => []
      }, 'INT' )
    ]
  }, 'F' ),
```

```
  'CHAR' => bless( {
    'children' => []
  }, 'CHAR' ),
  'INT' => bless( {
    'children' => []
  }, 'INT' )
};
Symbol Table:
$VAR1 = {
  'a' => {
    'type' => 'INT',
    'line' => 1
  },
  'b' => {
    'type' => 'INT',
    'line' => 1
  },
  'f' => {
    'type' => 'F(X_1(CHAR),INT)',
    'line' => 3
  }
};

---------------------------
1)
$VAR1 = {
  'c' => {
    'type' => 'CHAR',
    'param' => 1,
    'line' => 3
  }
};
```

Usando `str` sobre una Lista de Áboles

Use `str` como método de clase

`Parse::Eyapp::Node->str(@forest)`

cuando quiera convertir a su representación término mas de un árbol. El código:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code> sed -ne '652,658p' Simple4.eyp \
                                            | cat -n
     1          local $Parse::Eyapp::Node::INDENT = 2;
     2          local $Parse::Eyapp::Node::DELIMITER = "";
     3          print $t->str."\n";
     4          {
     5            local $" = "\n";
     6            print Parse::Eyapp::Node->str(@blocks)."\n";
     7          }
```

produce la salida:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code> eyapp Simple4 ;\
                                          treereg SimpleTrans.trg ;\
```

`usesimple4.pl`

```
****************
f() {
  int a,b[1][2],c[1][2][3];
  char d[10];
  b[0][1] = a;
}

PROGRAM(
  FUNCTION[f](
    ASSIGN(
      VARARRAY(
        TERMINAL[b:4],
        INDEXSPEC(
          INUM(
            TERMINAL[0:4]
          ),
          INUM(
            TERMINAL[1:4]
          )
        ) # INDEXSPEC
      ) # VARARRAY,
      VAR(
        TERMINAL[a:4]
      )
    ) # ASSIGN
  ) # FUNCTION
) # PROGRAM

Match[PROGRAM:0:blocks](
  Match[FUNCTION:1:blocks:[f]]
)
```

Obsérvense los comentarios `# TIPODENODO` que acompañan a los paréntesis cerrar cuando estos están lejos (esto es, a una distancia de mas de `$Parse::Eyapp::Node::LINESEP` líneas) del correspondiente paréntesis abrir. Tales comentarios son consecuencia de haber establecido el valor de `$Parse::Eyapp::Node::INDENT` a 2.

5.9. Práctica: Establecimiento de la relación uso-declaración

En la segunda parte del análisis de ámbito se resuelve el problema de la identificación de los nombres estableciendo la relación entre el uso de un nombre y la declaración que se le aplica (dada por los atributos `scope` y `type`):

Añada las siguientes comprobaciones dependientes del contexto al compilador de Simple C:

- Resuelva el problema del ámbito de los identificadores: asocie cada uso de un identificador con la declaración que se le aplica.
- Asocie cada nodo `RETURN` con el nodo de la subrutina que lo engloba. Añádale un atributo al nodo `RETURN` que indique el tipo que la rutina debe retornar.
- Asocie cada nodo `CONTINUE` y cada nodo `BREAK` con el nodo del bucle que lo engloba. Indique el error si la sentencia (`CONTINUE` o `BREAK`) no esta dentro de un bucle. Añada a dichos nodos un atributo que referencia al nodo del bucle que lo engloba.

- Añada etiquetas de salto y la sentencia `goto` al lenguaje Simple C. Toda etiqueta va asociada con una sentencia. La sintáxis de statement debe ser extendida para incluir sentencias etiquetadas:

```
statement:
    expression ';' { $_[1] }
  | ';'
  | %name BREAK
    $BREAK ';'
      { ...  }
  | %name CONTINUE
     $CONTINUE ';'
      { ... }
  | 'goto' ID ';'
  | ID ':' statement
      { ... }
;
```

  Las siguientes indicaciones tienen por objeto ayudarle en el análisis de ámbito de etiquetas:

  - Sólo se permiten saltos dentro de la misma subrutina.
  - Será necesario crear un manejador de ámbito de etiquetas mediante `Parse::Eyapp::Scope->new`
  - Al comienzo de cada función comienza un ámbito de etiquetas.
  - Cada vez que se encuentra una sentencia etiquetada la etiqueta ID es guardada en una tabla de símbolos de etiquetas.
  - El valor asociado con la entrada para la etiqueta es una referencia a un hash. La clave mas importante en ese hash es la referencia a la sentencia que etiqueta.
  - Cada aparición de un goto es una instancia de ámbito.
  - La llamada a `end_scope` (al terminar el análisis de la función) deberá decorar automáticamente el nodo GOTO con la referencia a la sentencia asociada con la etiqueta
  - La tabla de etiquetas es un atributo del nodo asociado con la función.

5.10. Práctica: Establecimiento de la Relación Uso-Declaración Usando Expresiones Regulares Árbol

En la práctica anterior resolvió los problemas de ámbito usando Parse::Eyapp::Scope. En esta se pide resolverlos usando expresions regulares árbol:

- Asocie cada nodo `RETURN` con el nodo de la subrutina que lo engloba. Añádale un atributo al nodo `RETURN` que indique el tipo que la rutina debe retornar.
- Asocie cada nodo `CONTINUE` y cada nodo `BREAK` con el nodo del bucle que lo engloba. Indique el error si la sentencia (`CONTINUE` o `BREAK`) no esta dentro de un bucle. Añada a dichos nodos un atributo que referencia al nodo del bucle que lo engloba.
- Añada etiquetas de salto y la sentencia `goto` al lenguaje Simple C. Toda etiqueta va asociada con una sentencia.

5.11. Práctica: Estructuras y Análisis de Ámbito

Los registros o estructuras introducen espacios de nombres. Los selectores de campo viven en los espacios de nombres asociados con los registros. Modifique de manera apropiada la gramática de `SimpleC` para introducir el tipo de datos `struct`. Complete el análisis sintáctico.

- Es necesario introducir nuevas variables sintácticas:

  ```
  type_specifier:
       basictype
     | struct
  ;

  struct:
      'struct' ID '{' struct_decl_list '}'
    | 'struct' ID
  ;

  struct_decl_list: (type_specifier declList ';') +
  ;
  ```

  Recuerde que:

  ```
  declList: (ID arraySpec) <%+ ','>
  ;
  ```

- Es necesario tambien modificar algunas categorías gramáticales existentes, como `declaration`:

  ```
  declaration:
      %name DECLARATION
      $type_specifier $declList ';'
  ;
  ```

  También es necesario modificar `Variable`:

  ```
  Variable:
      ID
    | Variable ('[' binary ']')
    | Variable '.' ID
  ;
  ```

  Sigue un ejemplo de derivación generando una variable de la forma `a[i].x[4].b`:

  ```
  Variable => Variable.ID => Variable[binary].ID
    => Variable.ID[binary].ID => Variable[binary].ID[binary].ID
    => ID[binary].ID[binary].ID
  ```

  Veamos otro ejemplo de derivación generando una variable de la forma `a[i].x[4]`:

  ```
  Variable => Variable[binary] => Variable.ID[binary] => Variable[binary].ID[binary]
     => ID[binary].ID[binary]
  ```

- Es necesario extender las expresiones de tipo para incluir las `struct`. Un registro es similar al producto cartesiano (usado en las expresiones de tipo de las funciones), sólo que los campos tienen nombres. Asi:

  ```
  struct row {
    int address;
    char lexeme[10];
  } table[100];
  ```

la expresión de tipo para `struct row` podría ser:

```
STRUCT_row(address, INT, lexeme, A_10(CHAR))
```

o bien:

```
STRUCT_row(X_2(address, INT), X_2(lexeme, A_10(CHAR)))
```

Como se ve en el ejemplo el identificador de campo (`address`, `lexeme`) forma parte de la descripción del tipo.

- La introducción de las `struct` produce una cierta sobrecarga de identificadores. El siguiente programa es correcto:

  ```
  struct x {
    int z;
    int b;
  } x;

  main() {
    x;
  }
  ```

  Aqui `x` es usado como nombre de un tipo y como nombre de variable. Plantee soluciones al problema del almacenamiento en la tabla de símbolos. Una posibilidad es que la entrada en la tabla de símbolos para el tipo sea `struct x` mientras que para la variable es `x`.

- El cálculo de que declaración se aplica a una `struct` puede hacerse de manera similar a como ha sido ilustrado en las prácticas anteriores. Otra cosa son los identificadores de campo. Por ejemplo en

  ```
  a[i+j].x = 4;
  ```

  Es necesario calcular que `a[i+j]` tiene tipo `struct` y comprobar en ese momento que dicha `struct` tiene un campo `x` que es de tipo `int`. Hasta que no hemos calculado el tipo asociado con la expresión `a[i+j]` no sabemos si el uso de `x` es legal o no. Realmente el ámbito del identificador `x` no se hace en tiempo de análisis de ámbito sino en tiempo de comprobación de tipos. El cálculo del ámbito para estos identificadores de campo se mezcla con el cálculo de tipos. En cierto modo el uso de `x` en `a[i+j].x = 4` forma parte de la descripción del tipo:

  ```
  STRUCT_point(X_2(x, INT), X_2(y, INT))
  ```

  Por tanto en las reglas de `Variable`:

  ```
  Variable:
      ID
    | Variable ('[' binary ']')
    | Variable '.' ID
  ;
  ```

  A estas alturas solo estamos en condiciones de asociarle su definición al identificador principal, esto es, al de la regla $Variable \rightarrow ID$. Los demás deberán esperar a la fase de análisis de tipos.

- Hasta ahora casi lo único que guardabamos en las tablas de símbolos era el tipo y el número de línea (otros valores posibles son `param` para determinar si es un parámetro, o una referencia a la tabla de símbolos de la funcion en el caso de las funciones). Esto era suficiente porque sólo teniámos dos tipos de identificadores: los de variable y los de función. Ahora que tenemos tres tipos: funciones, variables y tipos es conveniente añadir un atributo `kind` que guarde la clase de objeto que es.

- C requiere que los nombres de tipo sean declarados antes de ser usados (con la excepción de las referencias a tipos `struct` no declarados). Por ejemplo, el siguiente programa:

  ```
  struct x {
    struct y z;
    int b;
  } w;

  struct y {
    int a;
  };
  ```

  produce un mensaje de error:

  ```
  prueba1.c:2: error: field ‘z’ has incomplete type
  ```

  Esto impide definiciones recursivas como:

  ```
  struct x {
    struct y z;
    int b;
  } w;

  struct y {
    struct x a;
  };
  ```

  A estas alturas del análisis estamos en condiciones de comprobar que no existan usos de declaraciones no definidas.

- Para simplificar el análisis de tipos posterior, las funciones en Simple C devuelven un tipo básico. En C las funciones pueden devolver un `struct` aunque no pueden devolver un array ni una función. Si que pueden sin embargo devolver un punero a un array o a una función.

  La restricción que imponemos de que el tipo retornado sea un tipo básico no significa que necesariamente la regla sintáctica deba especificar tal condición.

- Para simplificar la práctica consideraremos que todos los identificadores de tipos struct son globales. Por tanto es un error tener dos declaraciones de tipo struct con el mismo nombre. Claro está, si desea implantar ámbitos en los identificadores de tipos puede hacerlo.

- Consulte las siguientes fuentes:
  - Para ver una gramática de C consulte la página Cgrammar.html.
  - Una gramática de ANSI C se encuentra en http://www.lysator.liu.se/c/ANSI-C-grammar-y.html.

Capítulo 6

Análisis de Tipos

6.1. Análisis de Tipos: Conceptos Básicos

En la mayoría de los lenguajes los objetos manipulados son declarados en alguna parte del programa y usados en otras. Ya dijimos que el análisis de ámbito es el cálculo de la función que asigna a un uso de un objeto la definición que se le aplica.

El análisis de tipos tiene por objetivo asegurar que el uso de los objetos definidos es correcto: esto es, que su uso se atiene a la semántica de su definición; por ejemplo, que un array de enteros no es llamado como función o que no se intenta incrementar una función o que el valor retornado por una función es de la naturaleza descrita en su definición.

Expresiones de Tipo, Sistemas de Tipos y Comprobadores de Tipos

Definición 6.1.1. *Una forma adecuada de representar los tipos dentro de un* compilador *es usando un lenguaje de* expresiones de tipo*. Un lenguaje de las expresiones de tipo debe describir de manera clara y sencilla los tipos del lenguaje fuente. No confunda este lenguaje con el sub-lenguaje del lenguaje fuente que consiste en las declaraciones o definiciones. No tienen por que ser iguales. El compilador traduce las declaraciones de tipo en expresiones de tipo. El lenguaje de las expresiones de tipo es la representación interna que el compilador tiene de estas declaraciones y depende del compilador. El lenguaje de las declaraciones no.*

Definición 6.1.2. *Un* sistema de tipos *de un lenguaje/compilador es el conjunto de reglas del lenguaje (que es traducido e interpretado por el compilador que permiten asignar expresiones de tipo a las instancias de uso de los objetos del programa.*

Si bien el sistema de tipos es una propiedad del lenguaje, no es raro que los compiladores introduzcan modificaciones en el sistema de tipos del lenguaje. Por ejemplo en Pascal el tipo de un array incluye los índices del array[1]). Esto y las reglas de equivalencia de tipos de Pascal limitan gravemente la genericidad de las funciones en Pascal. Por eso algunos compiladores Pascal permiten en una llamada a función la compatibilidad de tipos entre arrays de diferente tamaño y diferentes conjuntos de índices. Desgraciadamente la forma en la que lo hacen puede diferir de compilador a compilador.

Definición 6.1.3. *Un* comprobador de tipos *verifica que el uso de los objetos en los constructos de uso se atiene a lo especificado en sus declaraciones o definiciones de acuerdo a las reglas especificadas por el* sistema de tipos.

Tipado Estático y Tipado Dinámico

Definición 6.1.4. *Un lenguaje de programación tiene* tipado estático *si su comprobación de tipos ocurre en tiempo de compilación sin tener que comprobar equivalencias en tiempo de ejecución.*

Un lenguaje de programación tiene tipado dinámico *si el lenguaje realiza comprobaciones de tipo en tiempo de ejecución. En un sistema de tipos dinámico* los tipos suelen estár asociados con los valores no con las variables.

---

[1] Traduttore, traditore!

Definición 6.1.5. *El tipado dinámico hace mas sencilla la escritura de* metaprogramas*: programas que reciben como datos otros códigos y los manipulan para producir nuevos códigos.* `Parse::Eyapp` *y* `Parse::Treeregexp` *son ejemplos de metaprogramación. Cada vez que escribimos un procesador de patrones, templates o esqueletos de programación estamos haciendo meta-programación.*

El lenguaje en el que se escribe el metaprograma se denomina metalenguaje*. El lenguaje al que se traduce el metaprograma se denomina* lenguaje objeto*. La capacidad de un lenguaje de programación para ser su propio metalenguaje se denomina* reflexividad*. Para que haya reflexión es conveniente que el código sea un tipo de estructura de datos soportado por el lenguaje al mismo nivel que otros tipos básicos y que sea posible traducir dinámicamente texto a código.*

Tipado Fuerte y Tipado Débil

Definición 6.1.6. *Aunque el significado de los términos* Fuertemente Tipado *y su contrario* Débilmente Tipado *varían con los autores, parece haber consenso en que los lenguajes con tipado fuerte suelen reunir alguna de estas características:*

- *La comprobación en tiempo de compilación de las violaciones de las restricciones impuestas por el sistema de tipos. El compilador asegura que para cualesquiera operaciones los operandos tienen los tipos válidos.*
- *Toda operación sobre tipos inválidos es rechazada bien en tiempo de compilación o de ejecución.*
- *Algunos autores consideran que el término implica desactivar cualquier conversión de tipos implícita. Si el programador quiere una conversión deberá explicitarla.*
- *La ausencia de modos de evadir al sistema de tipos.*
- *Que el tipo de un objeto de datos no varíe durante la vida del objeto. Por ejemplo, una instancia de una clase no puede ver su clase alterada durante la ejecución.*

Sobrecarga, Polimorfismo e Inferencia de Tipos

Un símbolo se dice *sobrecargado* si su significado varía dependiendo del contexto. En la mayoría de los lenguajes Los operadores aritméticos suelen estar sobrecargados, dado que se sustancian en diferentes algoritmos según sus operandos sean enteros, flotantes, etc.

En algunos lenguajes se permite la sobrecarga de funciones. así es posible tener dos funciones llamadas `min`:

```
int min(int a, int b) {
  if (a < b) return a;
  return b;
}
```

```
int min(string a, string b) {
  if (strcmp(a, b) < 0) return a;
  return b;
}
```

A la hora de evaluar el tipo de las expresiones es el contexto de la llamada el que determina el tipo de la expresión:

```
float x,y;
int a,b;
string c,d;

u = min(x,y); /* Puede que correcto: x e y seran truncados a enteros. Tipo entero */
v = min(a,b); /* Correcto: Tipo devuelto es entero */
w = min(c,d); /* Correcto: Tipo devuelto es string */
t = min(x,c); /* Error */
```

Ejercicio 6.1.1. *¿Como afecta al análisis de ámbito la sobrecarga de operadores?*

Definición 6.1.7. *La* inferencia de tipos *hace referencia a aquellos algoritmos que deducen automáticamente en tiempo de compilación - sin información adicional del programador, o bien con anotaciones parciales del programador - el tipo asociado con un uso de un objeto del programa. Un buen número de lenguajes de programación funcional permiten implantar inferencia de tipos (Haskell, OCaml, ML, etc).*

Véase como ejemplo de inferencia de tipos la siguiente sesión en *OCaml*:

```
pl@nereida:~/src/perl/attributegrammar/Language-AttributeGrammar-0.08/examples$ ocaml
        Objective Caml version 3.09.2

# let minimo = fun i j -> if i<j then i else j;;
val minimo : 'a -> 'a -> 'a = <fun>
# minimo 2 3;;
- : int = 2
# minimo 4.9 5.3;;
- : float = 4.9
# minimo "hola" "mundo";;
- : string = "hola"
```

El compilador `OCaml` infiere el tipo de las expresiones. Así el tipo asociado con la función `minimo` es `'a -> 'a -> 'a` que es una *expresión de tipo* que contiene *variables de tipo*. El operador `->` es asociativo a derechas y asi la expresión debe ser leída como `'a -> ('a -> 'a)`. Básicamente dice: es una función que toma un argumento de tipo `'a` (donde `'a` es una variable tipo que será instanciada en el momento del uso de la función) y devuelve una función que toma elementos de tipo `'a` y retorna elementos de tipo `'a`.

Definición 6.1.8. *Aunque podría pensarse que una descripción mas adecuada del tipo de la función* `minimo` *fuera* `'a x 'a -> 'a`*, lo cierto es que en algunos lenguajes funcionales es usual que todas las funciones sean consideradas como funciones de una sóla variable. La función de dos variables* `'a x 'a -> 'a` *puede verse como una función* `'a -> ('a -> 'a)`*. En efecto la función* `minimo` *cuando recibe un argumento retorna una función:*

```
# let min_mundo = minimo "mundo";;
val min_mundo : string -> string = <fun>
# min_mundo "pedro";;
- : string = "mundo"
# min_mundo "antonio";;
- : string = "antonio"
# min_mundo 4;;
This expression has type int but is here used with type string
# min_mundo(string_of_int(4));;
- : string = "4"
```

Esta estrategia de reducir funciones de varias variables a funciones de una variable que retornan funciones de una variable se conoce con el nombre de currying *o* aplicación parcial*.*

Definición 6.1.9. *El* polimorfismo *es una propiedad de ciertos lenguajes que permite una interfaz uniforme a diferentes tipos de datos.*

Se conoce como función polimorfa *a una función que puede ser aplicada o evaluada sobre diferentes tipos de datos.*

Un tipo de datos se dice polimorfo *si es un tipo de datos generalizado o no completamente especificado. Por ejemplo, una lista cuyos elementos son de cualquier tipo.*

Definición 6.1.10. *Se llama* Polimorfismo Ad-hoc *a aquel en el que el número de combinaciones que pueden usarse es finito y las combinaciones deben ser definidas antes de su uso. Se habla de* polimorfismo paramétrico *si es posible escribir el código sin mención específica de los tipos, de manera que el código puede ser usado con un número arbitrario de tipos.*

Por ejemplo, la herencia y la sobrecarga de funciones y métodos son mecanismos que proveen polimorfismo ad-hoc. Los lenguajes funcionales, como OCaml suelen proveer polimorfismo paramétrico. En OOP el polimorfismo paramétrico suele denominarse *programación genérica*

En el siguiente ejemplo en `OCaml` construimos una función similar al `map` de Perl. La función `mymap` ilustra el polimorfismo paramétrico: la función puede ser usada con un número arbitrario de tipos, no hemos tenido que hacer ningún tipo de declaración explícita y sin embargo el uso incorrecto de los tipos es señalado como un error:

```
# let rec mymap f list =
  match list with
      [] -> []
    | hd :: tail -> f hd :: mymap f tail;;
val mymap : ('a -> 'b) -> 'a list -> 'b list = <fun>
# mymap (function  n -> n*2) [1;3;5];;
- : int list = [2; 6; 10]
# mymap (function  n -> n.[0]) ["hola"; "mundo"];;
- : char list = ['h'; 'm']
# mymap (function  n -> n*2) ["hola"; "mundo"];;
This expression has type string but is here used with type int
```

Equivalencia de Expresiones de Tipo

La introducción de nombres para las expresiones de tipo introduce una ambiguedad en la interpretación de la equivalencia de tipos. Por ejemplo, dado el código:

```
typedef int v10[10];
v10 a;
int b[10];
```

¿Se considera que `a` y `b` tienen tipos compatibles?

Definición 6.1.11. *Se habla de* equivalencia de tipos estructural *cuando los nombres de tipo son sustituidos por sus definiciones y la equivalencia de las expresiones de tipo se traduce en la equivalencia de sus árboles sintácticos o DAGs. Si los nombres no son sustituidos se habla de* equivalencia por nombres *o de* equivalencia de tipos nominal*.*

Si utilizamos la opción de sustituir los nombres por sus definiciones y permitimos en la definición de tipo el uso de nombres de tipo no declarados se pueden producir ciclos en el grafo de tipos.

El lenguaje C impide la presencia de ciclos en el grafo de tipos usando dos reglas:

1. Todos los identificadores de tipo han de estar definidos antes de su uso, con la excepción de los punteros a registros no declarados

2. Se usa equivalencia estructural para todos los tipos con la excepción de las `struct` para las cuales se usa equivalencia nominal

Por ejemplo, el siguiente programa:

```
nereida:~/src/perl/testing> cat -n typeequiv.c
  1  #include <stdio.h>
  2
  3  typedef struct {
  4     int x, y;
```

```
 5      struct record *next;
 6   } record;
 7
 8   record z,w;
 9
10   struct recordcopy {
11      int x, y;
12      struct recordcopy *next;
13   } r,k;
14
15
16   main() {
17     k = r; /* no produce error */
18     z = w; /* no produce error */
19     r = z;
20   }
```

Produce el siguiente mensaje de error:

```
nereida:~/src/perl/testing> gcc -fsyntax-only typeequiv.c
typeequiv.c: En la función 'main':
typeequiv.c:19: error: tipos incompatibles en la asignación
```

En lenguajes dinámicos una forma habitual de equivalencia de tipos es el tipado pato:

Definición 6.1.12. *Se denomina* duck typing *o* tipado pato *a una forma de tipado dinámico en la que el conjunto de métodos y propiedades del objeto determinan la validez de su uso. Esto es: dos objetos pertenecen al mismo tipo-pato si implementan/soportan la misma interfaz independientemente de si tienen o no una relación en la jerarquía de herencia.*

El término hace referencia al llamado test del pato*:* If it waddles like a duck, and quacks like a duck, it's a duck!*.*

6.2. Conversión de Tipos

El comprobador de tipos modifica el árbol sintáctico para introducir *conversiones de tipo* donde sean necesarias. Por ejemplo, si tenemos

```
int i;
float x;

x+i;
```

Dado el árbol de la expresión `PLUS(VAR, VAR)`, el analizador de tipos introducirá un nodo intermedio `INT2FLOAT` para indicar la necesidad de la conversión y especificará el tipo de `PLUS` que se usa:

`PLUSFLOAT(VAR, INT2FLOAT(VAR)).`

Una transformación árbol de optimización que entra en este punto es la conversión de tipo en tiempo de compilación de las constantes. Por ejemplo, dados los dos programas:

```
float X[N];
int i;

for(i=0; i<N; i++) {
  X[i] = 1;
}
```

```
float X[N];
int i;

for(i=0; i<N; i++) {
  X[i] = 1.0;
}
```

los efectos sobre el rendimiento serán lamentables si el compilador no realiza la conversión de la constante entera `1` del programa de la izquierda en tiempo de compilación sino que la conversión se deja a una subrutina de conversión que es llamada en tiempo de ejecución. En tal caso se obtendrían rendimientos completamente diferentes para los programas en la izquierda y en la derecha.

Ejercicio 6.2.1. *Escriba el seudocódigo de las transformaciones árbol Treeregexp que implementan la conversión de tipos de constantes en tiempo de compilación*

6.3. Expresiones de Tipo en Simple C

En las secciones anteriores dedicadas al análisis de ámbito introdujimos nuestro lenguaje de expresiones de tipo.

Así la expresión `F(X_2(INT,CHAR),INT)` describe el tipo de las funciones de 2 parámetros que devuelven enteros. El primer parámetro debe ser entero y el segundo un carácter.

Otro ejemplo es la expresión de tipo `A_10(A_20(INT))` que describe un vector 10x20 de enteros.

Ejercicio 6.3.1. *Describa la gramática independiente del contexto que genera las expresiones de tipo usadas por el compilador de Simple C tal y como fueron presentadas en las secciones previas*

Observe que en tanto en cuanto las expresiones de tipo son frases de un lenguaje - cuya semántica es describir los tipos del lenguaje fuente - tiene un árbol sintáctico asociado. En las secciones anteriores sobre el análisis de ámbito presentamos una versión inicial de las expresiones de tipo y las utilizamos mediante una representación dual: como cadena y como árbol. Para ello usábamos el método `Parse::Eyapp::Node->new` el cual nos permite obtener una representación árbol de la expresión de tipo. En un contexto escalar este método devuelve la referencia al árbol sintáctico construido:

```
nereida:~/src/perl/testing> perl -MParse::Eyapp::Node -de 0
Loading DB routines from perl5db.pl version 1.28
Editor support available.
main::(-e:1):   0
  DB<1> x scalar(Parse::Eyapp::Node->new('F(X_2(INT,CHAR),INT)'))
0  F=HASH(0x859fb58)
   'children' => ARRAY(0x85d52ec)
      0  X_2=HASH(0x859fb70)
         'children' => ARRAY(0x859fb28)
            0  INT=HASH(0x85a569c)
               'children' => ARRAY(0x852cc84)
                    empty array
            1  CHAR=HASH(0x859fb34)
               'children' => ARRAY(0x850a7cc)
                    empty array
      1  INT=HASH(0x859fabc)
         'children' => ARRAY(0x852c9d8)
              empty array
```

6.4. Construcción de las Declaraciones de Tipo con `hnew`

Una manera aún mas conveniente de representar una expresión de tipo es atraves de un *grafo dirigido acíclico* o *DAG* (*Directed Acyclic Graph*).

`hnew`

Parse::Eyapp provee el método `Parse::Eyapp::Node->hnew` el cual facilita la construcción de DAGs. El método `hnew` trabaja siguiendo lo que en programación funcional se denomina *hashed consing*. En *Lisp* la función de asignación de memoria es `cons` y la forma habitual de compartición es a través de una tabla hash. Hash consing es una técnica para compartir datos que son estructuralmente iguales. De esta forma se ahorra memoria. La técnica esta relacionada con - puede considerarse una aplicación concreta de - la técnica conocida con el nombre de *memoization* [2]. Si alguno de los subárboles (o el árbol) fueron creados anteriormente no serán creados de nuevo sino que `hnew` retorna una referencia al ya existente. Esto hace mas rápida la comprobación de tipos: *en vez de comprobar que los dos árboles son estructuralmente equivalentes basta con comprobar que las referencias son iguales.* Asi la comprobación de la equivalencia de tipos pasa a requerir tiempo constante.

Observe los siguientes comandos:

```
 DB<2> x $a = Parse::Eyapp::Node->hnew('F(X_3(A_3(A_5(INT)), CHAR, A_5(INT)),CHAR)')
0  F=HASH(0x85f6a20)
   'children' => ARRAY(0x85e92e4)
   |- 0  X_3=HASH(0x83f55fc)
   |     'children' => ARRAY(0x83f5608)
   |     |- 0  A_3=HASH(0x85a0488)
   |     |     'children' => ARRAY(0x859fad4)
   |     |        0  A_5=HASH(0x85e5d3c)
   |     |           'children' => ARRAY(0x83f4120)
   |     |              0  INT=HASH(0x83f5200)
   |     |                 'children' => ARRAY(0x852ccb4)
   |     |                      empty array
   |     |- 1  CHAR=HASH(0x8513564)
   |     |     'children' => ARRAY(0x852cad4)
   |     |          empty array
   |     `- 2  A_5=HASH(0x85e5d3c)
   |           -> REUSED_ADDRESS
   `- 1  CHAR=HASH(0x8513564)
         -> REUSED_ADDRESS
 DB<3> x $a->str
0  'F(X_3(A_3(A_5(INT)),CHAR,A_5(INT)),CHAR)'
```

La segunda aparición de `A_5(INT)` aparece etiquetada como reusada (tercer hijo de `X_3`). Lo mismo ocurre con la segunda aparición de `CHAR`.

`hnew` y el Manejador de Atributos

El valor asignado a los atributos no influye en la memoización de `hnew`: si los términos tienen la misma forma se devolverán los mismos árboles:

```
pl@nereida:~/src/perl/testing$ perl -MParse::Eyapp::Node -wde 0
main::(-e:1):   0
  DB<1> @x = Parse::Eyapp::Node->hnew('A(B,C(D))', sub { $_->{n} = $i++ for @_ })
  DB<2> @y = Parse::Eyapp::Node->hnew('C(D)', sub { $_->{t} = 10-$j++ for @_ })
  DB<3> x @x
0  A=HASH(0x850935c)
   'children' => ARRAY(0x8509530)
      0  B=HASH(0x85095a8)
         'children' => ARRAY(0x8326800)
              empty array
         'n' => 1
      1  C=HASH(0x8509410)
         'children' => ARRAY(0x85094dc)
            0  D=HASH(0x85094b8)
               'children' => ARRAY(0x85095f0)
                    empty array
```

```
               'n' => 3
               't' => 9
         'n' => 2
         't' => 10
   'n' => 0
1  B=HASH(0x85095a8)
   -> REUSED_ADDRESS
2  C=HASH(0x8509410)
   -> REUSED_ADDRESS
3  D=HASH(0x85094b8)
   -> REUSED_ADDRESS
  DB<4> x @y
0  C=HASH(0x8509410)
   'children' => ARRAY(0x85094dc)
      0  D=HASH(0x85094b8)
         'children' => ARRAY(0x85095f0)
              empty array
         'n' => 3
         't' => 9
   'n' => 2
   't' => 10
1  D=HASH(0x85094b8)
   -> REUSED_ADDRESS
```

La Función build_function_scope

Sigue el código de la función `build_function_scope`:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ \
                  sed -ne '/sub bu.*scop/,/^}/p' Types.eyp | cat -n
 1  sub build_function_scope {
 2    my ($funcDef, $returntype) = @_;
 3
 4    my $function_name = $funcDef->{function_name}[0];
 5    my @parameters = @{$funcDef->{parameters}};
 6    my $lst = $funcDef->{symboltable};
 7    my $numargs = scalar(@parameters);
 8
 9    #compute type
10    my $partype = "";
11    my @types = map { $lst->{$_}{type} } @parameters;
12    $partype .= join ",", @types if @types;
13
14    my $type = "F(X_$numargs($partype),$returntype)";
15
16    #insert it in the hash of types
17    $type{$type} = Parse::Eyapp::Node->hnew($type);
18    $funcDef->{type} = $type;
19    $funcDef->{t} = $type{$type};
20
21    #insert it in the global symbol table
22    die "Duplicated declaration of $function_name at line $funcDef->{function_name}[1]\n"
23      if exists($st{$function_name});
24    $st{$function_name}->{type} = $type;
25    $st{$function_name}->{line} = $funcDef->{function_name}[1];
```

```
26
27    return $funcDef;
28  }
```

Obsérvese que para acelerar los accesos al atributo tipo dotamos a los nodos del atributo `t` que permite acceder directamente al DAG representando el tipo.

6.5. Inicio de los Tipos Básicos

Al comienzo y al final del análisis del programa fuente se inician mediante la llamada a `reset_file_scope_vars` los tipos básicos `INT`, `CHAR` y `VOID`. Este último se utilizará en las expresiones de tipo para representar la ausencia de tipo.

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ \
                    sed -ne '/^program:/,/^;/p' Types.eyp | cat -n
 1  program:
 2        {
 3          reset_file_scope_vars();
 4        }
 5      definition<%name PROGRAM +>.program
 6        {
 7          $program->{symboltable} = { %st };  # creates a copy of the s.t.
 8          $program->{depth} = 0;
 9          $program->{line}  = 1;
10          $program->{types} = { %type };
11          $program->{lines} = $tokenend;
12
13          my ($nondec, $declared) = $ids->end_scope($program->{symboltable}, $program, 'type
14
15          if (@$nondec) {
16            warn "Identifier ".$_->key." not declared at line ".$_->line."\n" for @$nondec;
17            die "\n";
18          }
19
20          # Type checking: add a direct pointer to the data-structure
21          # describing the type
22          $_->{t} = $type{$_->{type}} for @$declared;
23
24          my $out_of_loops = $loops->end_scope($program);
25          if (@$out_of_loops) {
26            warn "Error: ".ref($_)." outside of loop at line $_->{line}\n" for @$out_of_loop
27            die "\n";
28          }
29
30          # Check that are not dangling breaks
31          reset_file_scope_vars();
32
33          $program;
34        }
35  ;
```

Se procede a establecer el atributo `t` como enlace directo a la expresión de tipo. Lo mismo se hace en las funciones y en los bloques.

Sigue el código de `reset_file_scope_vars`:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ \
                  sed -ne '/^sub re.*vars/,/^}/p' Types.eyp | cat -n
sub reset_file_scope_vars {
  %st = (); # reset symbol table
  ($tokenbegin, $tokenend) = (1, 1);
  %type = ( INT  => Parse::Eyapp::Node->hnew('INT'),
            CHAR => Parse::Eyapp::Node->hnew('CHAR'),
            VOID => Parse::Eyapp::Node->hnew('VOID'),
          );
  $depth = 0;
  $ids = Parse::Eyapp::Scope->new(
           SCOPE_NAME => 'block',
           ENTRY_NAME => 'info',
           SCOPE_DEPTH => 'depth',
  );
  $loops = Parse::Eyapp::Scope->new(
           SCOPE_NAME => 'exits',
  );
  $ids->begin_scope();
  $loops->begin_scope(); # just for checking
}
```

6.6. Comprobación Ascendente de los Tipos

La comprobación de tipos se hace mediante una visita ascendente del AST: primero se comprueban y computan los tipos de los hijos del nodo aplicándole las reglas del sistema de tipos que hemos implantado usando transformaciones árbol. Despúes pasamos a computar el tipo del nodo. Para la visita usamos el método `bud` (por bottom-up decorator[2]) de `Parse::Eyapp::Node`.

La llamada `$t->bud(@typecheck)` hace que se visite cada uno de los nodos de `$t` en orden bottom-up. Para cada nodo se busca una transformación árbol en la lista `@typecheck` que se pueda aplicar. Tan pronto como se encuentra una se aplica y se procede con el siguiente nodo.

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ \
                   sed -ne '/^sub compile/,/^}/p' Types.eyp | cat -n
sub compile {
 my($self)=shift;

 my ($t);

 $self->YYData->{INPUT} = $_[0];

 $t = $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
                      #yydebug => 0x1F
     );

 # Scope Analysis: Block Hierarchy
 our $blocks;
 my @blocks = $blocks->m($t);
 $_->node->{fatherblock} = $_->father->{node} for (@blocks[1..$#blocks]);
```

---

[2] De acuerdo al diccionario:

> bud: n. Botany. A small protuberance on a stem or branch, sometimes enclosed in protective scales and containing an undeveloped shoot, leaf, or flower.

```
16
17   # Scope Analysis: Return-Function
18   our $retscope; # retscope: /FUNCTION|RETURN/
19   my @returns = $retscope->m($t);
20   for (@returns) {
21     my $node = $_->node;
22     if (ref($node) eq 'RETURN') {
23       my $function = $_->father->node;
24       $node->{function}  = $function;
25       $node->{t} = $function->{t}->child(1);
26     }
27   }
28
29   # Type checking
30   set_types($t);        # Init basic types
31
32   my @typecheck = (     # Check typing transformations for
33     our $inum,          # - Numerical constantss
34     our $charconstant,  # - Character constants
35     our $bin,           # - Binary Operations
36     our $arrays,        # - Arrays
37     our $assign,        # - Assignments
38     our $control,       # - Flow control sentences
39     our $functioncall,  # - Function calls
40     our $statements,    # - Those nodes with void type
41                         #   (STATEMENTS, PROGRAM, etc.)
42     our $returntype,    # - Return
43   );
44
45   $t->bud(@typecheck);
46
47   # The type checking for trees RETURN exp is made
48   # in adifferent way. Just for fun
49   #our $bind_ret2function;
50   #my @FUNCTIONS = $bind_ret2function->m($t);
51
52   return $t;
53  }
```

La llamada a la subrutina `set_types($t)` tiene por objeto establecer la comunicación entre el programa árbol en `Trans.trg` y el compilador en `Types.eyp`. La subrutina se encuentra en el código de apoyo en el programa árbol:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ \
                                 sed -ne '17,62p' Trans.trg | cat -n
 1  {
 2
 3    my $types; # reference to the hash containing the type table
 4    my ($INT, $CHAR, $VOID);
 5
 6    sub type_error {
 7      my $msg = shift;
 8      my $line = shift;
 9       die "Type Error at line $line: $msg\n"
```

```
  }

  sub set_types {
    my $root = shift;
    $types = $root->{types};
    $INT = $types->{INT};
    $CHAR = $types->{CHAR};
    $VOID = $types->{VOID};
  }

  sub char2int {
    ................;
  }

  sub int2char {
    ................;
  }

}
```

La subrutina `set_types` inicializa la variable léxica `$types` que referencia a la tabla de tipos: No olvide que estamos en el fichero `Trans.trg` separado del que contiene las restantes fases de análisis léxico, sintáctico y de ámbito. Además la rutina inicia las referencias `$INT`, `$CHAR` y `$VOID` que serán los árboles/hoja que representen a los tipos básicos. Siguiendo el clásico texto del Dragón de Aho, Hopcroft y Ullman [6] introducimos el tipo `VOID` para asociarlo a aquellos objetos que no tienen tipo (sentencias, etc.).

6.7. Análisis de Tipos: Mensajes de Error

El Problema

Durante esta fase pasamos a asignar un tipo a los nodos del AST. Pueden producirse errores de tipo y es importante que los mensajes de diagnóstico sea precisos. Por ejemplo, dado el programa:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/script> cat -n prueba17.c
     1  int c[20][30], d;
     2
     3  int f(int a, int b) {
     4    return
     5       (a+b)*
     6       d*
     7       c[5];
     8  }
```

queremos producir un mensaje de error adecuado como:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/script> usetypes.pl prueba17.c
Type Error at line 6: Incompatible types with operator '*'
```

indicando asi que el error se produce en la multiplicación de la línea 6, ya que `c[5]` es de tipo array.

El problema que tenemos a estas alturas de la construcción de nuestro compilador es que el número de línea asociado con la multiplicación de la línea 6 no figura como atributo del nodo `TIMES`: fue obviado por la directiva `%tree` ya que el terminal `*` fué tratado como un *syntactic token*.

El método `TERMINAL::save_attributes`

Afortunadamente `Parse::Eyapp` provee un mecanismo para permitir guardar la información residente en un terminal sintáctico en el nodo padre del mismo. Si el programador provee a la clase `TERMINAL` de un método `save_attributes` dicho método será llamado durante la construcción del AST en el momento de la eliminación del terminal:

```
TERMINAL::save_attributes($terminal, $lhs)
```

El primer argumento es la referencia al nodo `TERMINAL` y el segundo al node padre del terminal.

Por tanto, para resolver el problema de conocer el número de línea en el que ocurre un operador, proveeremos a los nodos `TERMINAL` del siguiente método `save_attributes`:

```
sub TERMINAL::save_attributes {
  # $_[0] is a syntactic terminal
  # $_[1] is the father.
  push @{$_[1]->{lines}}, $_[0]->[1]; # save the line!
}
```

La Función `insert_method`

La subrutina de mensajes de error de tipo `type_error` la llamaremos proporcionando el mensaje y el número de línea. Sigue un ejemplo de llamada:

```
type_error("Incompatible types with operator ’".($_[0]->lexeme)."’", $_[0]->line);
```

El método `line` para este tipo de nodos tiene la forma:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ \
                      sed -ne ’/sub P.*ne/,/^)/p’ Types.eyp | cat -n
     1  sub PLUS::line {
     2    $_[0]->{lines}[0]
     3  }
     4
     5  insert_method(
     6    qw{TIMES DIV MINUS ASSIGN GT IF RETURN},
     7    ’line’,
     8    \&PLUS::line
     9  );
```

La función `insert_method` es proveída por `Parse::Eyapp::Base` (versiones de Parse::Eyapp posteriores a la 1.099). Recibe como argumentos una lista de clases, el nombre del método (`’line’` en el ejemplo) y la referencia a la función que la implementa. Dotará a cada una de las clases especificadas en la lista (`TIMES`, `DIV` etc.) con métodos `line` implantados a través de la función especificada (`\&PLUS::line`).

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ \
                            sed -ne ’/use/,/^$/p’ Types.eyp | cat -n
     1  use strict;
     2  use Carp;
     3  use warnings;
     4  use Data::Dumper;
     5  use List::MoreUtils qw(firstval);
     6  use Simple::Trans;
     7  use Parse::Eyapp::Scope qw(:all);
     8  use Parse::Eyapp::Base qw(insert_function insert_method);
     9  our $VERSION = "0.4";
    10
```

El Lexema Asociado con Una Clase

El método `lexeme` que aparece en la llamada a `type_error` devuelve para un tipo de nodo dado el lexema asociado con ese nodo. Por ejemplo, para los nodos `TIMES` retorna la cadena `*`. Para implantarlo se usan los hashes `lexeme` y `rlexeme`:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ \
                                  sed -ne '31,64p' Types.eyp | cat -n
     1  my %lexeme = (
     2    '='  => "ASSIGN",
     3    '+'  => "PLUS",
     4    '-'  => "MINUS",
     5    '*'  => "TIMES",
     6    '/'  => "DIV",
     7    '%'  => "MOD",
     8    '|'  => "OR",
     9    '&'  => "AND",
    10    '{'  => "LEFTKEY",
    11    '}'  => "RIGHTKEY",
    12    ','  => "COMMA",
    13    ';'  => "SEMICOLON",
    14    '('  => "LEFTPARENTHESIS",
    15    ')'  => "RIGHTPARENTHESIS",
    16    '['  => "LEFTBRACKET",
    17    ']'  => "RIGHTBRACKET",
    18    '==' => "EQ",
    19    '+=' => "PLUSASSIGN",
    20    '-=' => "MINUSASSIGN",
    21    '*=' => "TIMESASSIGN",
    22    '/=' => "DIVASSIGN",
    23    '%=' => "MODASSIGN",
    24    '!=' => "NE",
    25    '<'  => "LT",
    26    '>'  => "GT",
    27    '<=' => "LE",
    28    '>=' => "GE",
    29    '++' => "INC",
    30    '--' => "DEC",
    31    '**' => "EXP"
    32  );
    33
    34  my %rlexeme = reverse %lexeme;
```

Cada una de las clases implicadas es dotada de un método `lexeme` mediante una llamada a `insert_method`:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ \
                       sed -ne '/sub.*xeme/,/^)/p' Types.eyp | cat -n
 1  sub lexeme {
 2    return $rlexeme{ref($_[0])} if defined($rlexeme{ref($_[0])});
 3    return ref($_[0]);
 4  }
 5
 6  insert_method(
 7    # Gives the lexeme for a given operator
 8    qw{
 9      PLUS TIMES DIV MINUS
```

```
    GT  EQ NE
    IF
    IFELSE
    WHILE
    VARARRAY VAR ASSIGN
    FUNCTIONCALL
  },
  'lexeme',
  \&lexeme
);
```

Listas y `save_attributes`

El método `save_attributes` no se ejecuta automáticamente en las reglas que usan los operadores de listas `+`, `*` y `?`. En tales casos el programador deberá proveer código explícito que salve el número de línea:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> \
    sed -ne '353,363p' Types.eyp  | cat -n
Variable:
    %name VAR
    ID
  | %name  VARARRAY
    $ID ('[' binary ']') <%name INDEXSPEC +>
      {
        my $self = shift;
        my $node =  $self->YYBuildAST(@_);
        $node->{line} = $ID->[1];
        return $node;
      }
```

En el ejemplo se pone explícitamente como atributo `line` del nodo `VARARRAY` el número de línea del terminal `ID`.

6.8. Comprobación de Tipos: Las Expresiones

La comprobación de tipos la haremos expandiendo nuestro programa árbol con nuevas transformaciones para cada uno de los tipos de nodos.

Expresiones Binarias

Las operaciones binarias requieren que sus operandos sean de tipo entero. Si el tipo de alguno de los operandos es `CHAR` haremos una conversión explícita al tipo `INT` (líneas 17-18):

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> sed -ne '66,90p' Trans.trg |
 1  bin: / PLUS
 2         |MINUS
 3         |TIMES
 4         |DIV
 5         |MOD
 6         |GT
 7         |GE
 8         |LE
 9         |EQ
10         |NE
11         |LT
```

```
12        |AND
13        |EXP
14        |OR
15       /($x, $y)
16    => {
17      $x = char2int($_[0], 0);
18      $y = char2int($_[0], 1);
19
20      if (($x->{t} == $INT) and ( $y->{t} == $INT)) {
21        $_[0]->{t} = $INT;
22        return 1;
23      }
24      type_error("Incompatible types with operator ’".($_[0]->lexeme)."’", $_[0]->line);
25    }
```

Modificación de la Semántica de las Expresiones Regulares

Obsérve la expresión regular lineal en las líneas 1-15. La semántica de las expresiones regulares lineales *es modificada ligéramente por* `Parse::Eyapp::Treeregexp`. Note que no se ha especificado la opción `x`. El compilador de expresiones regulares árbol la insertará por defecto. Tampoco es necesario añadir anclas de frontera de palabra `\b` a los identificadores que aparecen en la expresión regular lineal: de nuevo el compilador de expresiones regulares árbol las insertará.

Introducción de Nodos de Conversión de Tipos

Las subrutinas de conversión `char2int` e `int2char` se proveen - junto con la subrutina para emitir mensajes de error - como código de apoyo a las expresiones árbol. Recuerde que en un programa Treeregexp puede incluir código Perl en cualquier lugar sin mas que aislarlo entre llaves. Dicho código será insertado - respetando el orden de aparición en el fuente - en el módulo generado por `treereg`:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ sed -ne '/^{$/,/^}$/p' Tr
 1  {
 2
 3    my $types; # reference to the hash containing the type table
 4    my ($INT, $CHAR, $VOID);
 5
 6    sub type_error {
 7      my $msg = shift;
 8      my $line = shift;
 9       die "Type Error at line $line: $msg\n"
10    }
11
12    sub set_types {
..      ......................
18    }
19
20    sub char2int {
21      my ($node, $i) = @_;
22
23      my $child = $node->child($i);
24      return $child unless $child->{t} == $CHAR;
25
26      my $coherced = Parse::Eyapp::Node->new('CHAR2INT', sub { $_[0]->{t} = $INT });
27      $coherced->children($child);  # Substituting $node(..., $child, ...)
28      $node->child($i, $coherced);  # by          $node(..., CHAR2INT($child), ...)
```

```
29
30      return $coherced;
31    }
32
33    sub int2char {
34      my ($node, $i) = @_;
35
36      my $child = $node->child($i);
37      return $child unless $child->{t} == $INT;
38
39      my $coherced = Parse::Eyapp::Node->new(’INT2CHAR’, sub { $_[0]->{t} = $CHAR });
40      $coherced->children($child); # Substituting $node(..., $child, ... )
41      $node->child($i, $coherced); # by          $node(..., INT2CHAR($child), ...)
42
43      return $coherced;
44    }
45
46  }
```

Si el hijo i-ésimo del nodo `$node` es de tipo `$CHAR`, la línea 26 creará un nuevo nodo. Mediante el manejador pasado como segundo argumento a `Parse::Eyapp::Node->new` se dota al nodo del atributo `t`, el cual se establece a `$INT`. El nuevo nodo se interpone entre padre e hijo (líneas 27 y 28). De este modo se facilita la labor de la fase posterior de generación de código, explicitando la necesidad de generar código de conversión.

Si el lenguaje tuviera mas tipos numéricos, `FLOAT`, `DOUBLE` etc. es útil hacer especifico en el tipo del nodo que operación binaria se esta realizando cambiando el tipo `PLUS` en `PLUSDOUBLE`, `PLUSINT` etc. según corresponda.

Expresiones Constantes

La asignación de tipo a los nodos que se corresponden con expresiones constantes es trivial:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> \
    sed -ne ’60,61p’ Trans.trg | cat -n
     1  inum: INUM($x) => { $_[0]->{t} = $INT }
     2  charconstant: CHARCONSTANT($x) => { $_[0]->{t} = $CHAR }
```

Ejemplo Sigue un ejemplo de comprobación de tipos en expresiones:

```
pl@nereida:~/Lbook/code/Simple-Types/script$ usetypes.pl prueba18.c 2
1 int c[20][30], d;
2
3 int f(int a, int b) {
4   if (c[2] > 0)
5   return
6      (a+b)*d*c[1][1];
7 }
Type Error at line 4: Incompatible types with operator ’>’
```

6.9. Comprobación de Tipos: Indexados

Guía de Ruta en la Comprobación de Tipos para los Accesos a Arrays

En la comprobación de tipos de una expresión `x[y1][y2]` que referencia a un elemento de un array hay que tener en cuenta:

1. Que el tipo declarado para la variable `x` debe ser array (líneas 4-9 del código que sigue)
2. Que el número de dimensiones utilizadas en la expresión debe ser menor o igual que el declarado para la variable (líneas 11-17).
3. Si `x` es un array de tipo `t` el tipo de `x[2]` será `t`. Por ejemplo, en la llamada `g(x[y1][y2])` la variable `x` deberá haber sido declarada con al menos dos dimensiones. Si `x` fué declarada `int x[10][20][30]` el tipo de la expresión `x[y1][y2]` será `A_30(INT)`.
4. Que los índices usados deben ser de tipo entero. Si fueran de tipo `CHAR` podrán ser ahormados.

Código de Comprobación de Tipos para los Accesos a Arrays

Sigue el código para la comprobación de tipos de expresiones de referencias a elementos de un array:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> \
    sed -ne '97,125p' Trans.trg | cat -n
 1  arrays: VARARRAY($x, INDEXSPEC(@y))
 2     => {
 3
 4       my $t = $VARARRAY->{t}; # Type declared for VARARRAY
 5         type_error(           # Must be an array type
 6           " Variable '$x->{attr}[0]' was not declared as array",
 7           $VARARRAY->line
 8         )
 9       unless is_array($t);
10
11       my ($declared_dim, $used_dim, $ret_type) = compute_dimensionality($t, @y);
12
13         type_error(
14           " Variable '$x->{attr}[0]' declared with less than $used_dim dimensions",
15           $VARARRAY->line
16         )
17       unless $declared_dim >= $used_dim;
18
19       for (0..$#y) { # check that each index is integer. Coherce it if is $CHAR
20         my $ch = char2int($INDEXSPEC, $_);
21
22           type_error("Indices must be integers",$VARARRAY->line)
23         unless ($ch->{t} == $INT);
24       }
25
26       $VARARRAY->{t} = $ret_type;
27
28       return 1;
29     }
```

Funciones de Ayuda

La función `compute_dimensionality` llamada en la línea 11 se incluye dentro de la sección de funciones de soporte en `Trans.trg`:

```
 96  { # support for arrays
 97
 98    sub compute_dimensionality {
 99      my $t = shift;
```

```
    my $i = 0;
    my $q;
    my $used_dim = scalar(@_);
    for ($q=$t; is_array($q) and ($i < $used_dim); $q=$q->child(0)) {
      $i++
    }

    croak "Error checking array type\n" unless defined($q);
    return ($i, $used_dim, $q);
  }

  sub is_array {
    my $type = shift;

    defined($type) && $type =~ /^A_\d+/;
  }

  sub array_compatible {
    my ($a1, $a2) = @_;

    return 1 if $a1 == $a2;
    # int a[10][20] and int b[5][20] are considered compatibles
    return (is_array($a1) && is_array($a2) && ($a1->child(0) == $a2->child(0)));
  }
}
```

Ejemplos

El siguiente ejemplo muestra la comprobacion de tipos de arrays en funcionamiento:

```
pl@nereida:~/Lbook/code/Simple-Types/script$ usetypes.pl prueba02.c 2
 1 int a,b,e[10];
 2
 3 g() {}
 4
 5 int f(char c) {
 6 char d;
 7  c = 'X';
 8  e[d][b] = 'A'+c;
 9  {
10    int d;
11    d = a + b;
12  }
13  c = d * 2;
14  return c;
15 }
16
Type Error at line 8:  Variable 'e' declared with less than 2 dimensions
```

El siguiente ejemplo muestra la comprobacion de tipos de que los índices de arrays deben ser enteros:

```
pl@nereida:~/Lbook/code/Simple-Types/script$ usetypes.pl prueba06.c 2
 1 int a,b,e[10][20];
 2
 3 g() {}
 4
```

```
 5 int f(char c) {
 6 char d[20];
 7  c = 'X';
 8  g(e[d], 'A'+c);
 9  {
10    int d;
11    d = a + b;
12  }
13  c = d * 2;
14  return c;
15 }
16
Type Error at line 8: Indices must be integers
```

6.10. Comprobación de Tipos: Sentencias de Control

Las sentencias de control como `IF` y `WHILE` requieren un tipo entero en la condición de control:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ \
                        sed -ne '/^control/,189p' Trans.trg | cat -n
     1  control: /IF|IFELSE|WHILE/:con($bool)
     2    => {
     3      $bool = char2int($con, 0);
     4
     5        type_error("Condition must have integer type!", $bool->line)
     6      unless $bool->{t} == $INT;
     7
     8      $con->{t} = $VOID;
     9
    10      return 1;
    11    }
```

Veamos un ejemplo de uso:

```
l@nereida:~/Lbook/code/Simple-Types/script$ usetypes.pl prueba19.c 2
1 int c[20][30], d;
2
3 int f(int a, int b) {
4   if (c[2])
5   return
6      (a+b)*d*c[1][1];
7 }
Type Error at line 4: Condition must have integer type!
pl@nereida:~/Lbook/code/Simple-Types/script$
```

6.11. Comprobación de Tipos: Sentencias de Asignación

La comprobación de tipos en las asignaciones se divide en tres fases:

1. Coherción: líneas 10 y 11
2. Comprobación de la igualdad de los tipos (DAGs) de ambos lados de la igualdad (líneas 13 y 14)

3. Comprobación de que el tipo del lado izquierdo de la asignación está entre los permitidos por el lenguaje. En ANSI C se permite también la asignación de estructuras pero estas no están presentes en Simple C.

4. Si el lenguaje permitiera expresiones mas complejas en el lado izquierdo serían necesarias comprobaciones de que la expresión en la izquierda es ’asignable’ (es un `lvalue`). Por ejemplo en Kernighan y Ritchie C (KR C) [7] la asignación `f(2,x) = 4` no es correcta pero si podría serlo `*f(2,x) = 4`. En KR C las reglas de producción que definen la parte izquierda de una asignación (`lvalue`) son mas complejas que las de SimpleC, especialmente por la presencia de punteros:

Resumen de la Gramática de Kernighan y Ritchie C (KR C) en Parse::Eyapp

Expresiones

```
lvalue:
    identifier
  | primary ’[’ exp ’]’
  | lvalue ’.’ identifier
  | primary ’->’ identifier
  | ’*’ exp
  | ’(’ lvalue ’)’
primary:
    identifier
  | constant
  | string
  | (exp)
  | primary ’(’ exp <*, ’,’> ’)’
  | primary ’[’ exp ’]’
  | lvalue ’.’ identifier
  | primary ’->’ identifier
exp:
    primary
  | ’*’ exp
  | ’&’ exp
  | lvalue ASGNOP exp
  .........................
```

Declaraciones

```
fund:
    t? dec ’(’ pr <* ’,’> ’)’ fb
declaration:
   scty * (dec ini) <* ’,’>
scty:
    scope
  | t
scope:
     tdef | ...
t:
    char
  | int
  | double
  | tdef_name
  ..................
dec:
    identifier
  | ’(’ dec ’)’
  | ’*’ dec
  | dec ’(’ ’)’
  | dec ’[’ cexp ? ’]’
ini:
    ’=’ exp
  | ’=’ ’{’ exp <* ’,’> ’,’? ’}’
```

5. En la línea 20 modificamos el tipo del nodo: de ser `ASSIGN` pasa a ser `ASSIGNINT` o `ASSIGNCHAR` según sea el tipo de la asignación. De este modo eliminamos la sobrecarga semántica del nodo y hacemos explícito en el nodo el tipo de asignación. Poco a poco nos vamos acercando a niveles mas bajos de programación.

Sigue el código de comprobación para las asignaciones:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ \
                          sed -ne ’/^assign/,180p’ Trans.trg | cat -n
 1  assign: /ASSIGN
 2           |PLUSASSIGN
 3           |MINUSASSIGN
 4           |TIMESASSIGN
 5           |DIVASSIGN
 6           |MODASSIGN
```

```
     /:asgn($lvalue, $exp)
  => {
   my $lt =  $lvalue->{t};
   $exp = char2int($asgn, 1) if $lt == $INT;
   $exp = int2char($asgn, 1) if $lt == $CHAR;

    type_error("Incompatible types in assignment!", $asgn->line)
   unless ($lt == $exp->{t});

    type_error("The C language does not allow assignments to non-scalar types!", $asgn->
   unless ($lt == $INT) or ($lt == $CHAR); # Structs will also be allowed

   # Assignments are expressions. Its type is the type of the lhs or the rhs
   $asgn->{t} = $lt;

   # Make explicit the type of assignment, i.e.  s/PLUSASSIGN/PLUSASSIGNINT/
   $asgn->type(ref($asgn).ref($lt));

   return 1;
  }
```

Veamos un ejemplo de comprobación de tipos en asignaciones:

```
pl@nereida:~/Lbook/code/Simple-Types/script$ usetypes.pl prueba12.c 2
1 int a,b[10][20],e[10][20];
2
3 int f(char c) {
4   e[5] = b[5];
5   a = b[1][2];
6 }
7
Type Error at line 4: The C language does not allow assignments to non-scalar types!
```

6.12. Comprobación de Tipos: Llamadas a Funciones

Guía de Ruta para la Comprobación de Tipos de las LLamadas

La comprobación de tipos en una llamada a función conlleva los siguientes pasos:

1. Líneas 4 y 5: comprobamos que la variable que se usa en la llamada fue efectivamente declarada como función

2. Línea 14: El número de argumentos y el número de parámetros declarados debe coincidir

3. En las líneas de la 19 a la 35 analizamos la compatibilidad de tipo de cada uno de los argumentos con su correspondiente parámetro:

    a) En la línea 21 se obtiene el nodo correspondiente al argumento. Si es necesario se harán las coherciones (líneas 22 y 23)

    b) En la línea 25 comprobamos la compatibilidad de tipos. En el caso de los arrays la compatibilidad no se limita a la igualdad de referencias: consideraremos compatibles arrays que difieren en la primera dimensión. Así las declaraciones `int a[10][20]` y `int b[5][20]` se considerarán compatibles pero `int c[10][10]` y `int d[10][20]` no.

La forma del Árbol de las LLamadas

La siguiente sesión con el depurador ilustra la forma del árbol de análisis en las llamadas:

```
pl@nereida:~/Lbook/code/Simple-Types/script$ perl -wd usetypes.pl prueba23.c 2
main::(usetypes.pl:5):  my $filename = shift || die "Usage:\n$0 file.c\n";
  DB<1> f Types.eyp
1       2       3       4       5       6       7       #line 8 "Types.eyp"
8
9:      use strict;
10:     use Carp;
  DB<2> c 598
 1 int f(char a[10], int b) {
 2   return a[5];
 3 }
 4
 5 int h(int x) {
 6   return x*2;
 7 }
 8
 9 int g() {
10   char x[5];
11   int y[19][30];
12   f(x,h(y[1][1]));
13 }
Simple::Types::compile(Types.eyp:598):
598:     set_types($t);        # Init basic types
```

Ejecutamos hasta la línea 598, justo después de terminar el análisis de ámbito y antes de comenzar con la fase de comprobación de tipos.

A continuación mostramos una versión compacta del árbol. Para ello suprimimos la salida de notas a pie de página ”vaciando” el método `footnotes`:

```
  DB<3> insert_method(qw{PROGRAM FUNCTION}, 'footnote', sub {}) # Avoid footnotes
  DB<4> p $t->str
PROGRAM(
  FUNCTION[f](RETURN(VARARRAY(TERMINAL[a:2],INDEXSPEC(INUM(TERMINAL[5:2]))))),
  FUNCTION[h](RETURN(TIMES(VAR(TERMINAL[x:6]),INUM(TERMINAL[2:6])))),
  FUNCTION[g](
    FUNCTIONCALL(
      TERMINAL[f:12],
      ARGLIST(
        VAR(TERMINAL[x:12]),
        FUNCTIONCALL(
          TERMINAL[h:12],
          ARGLIST(VARARRAY(TERMINAL[y:12],INDEXSPEC(INUM(TERMINAL[1:12]),INUM(TERMINAL[1:12]))
)))))
  DB<6> p $t->descendant(".2.0")->str
FUNCTIONCALL(
  TERMINAL[f:12],
  ARGLIST(
    VAR(TERMINAL[x:12]),
    FUNCTIONCALL(
      TERMINAL[h:12],
      ARGLIST(VARARRAY(TERMINAL[y:12],INDEXSPEC(INUM(TERMINAL[1:12]),INUM(TERMINAL[1:12]))))))
  DB<7> p $t->descendant(".2.0")->{t}->str
```

```
F(X_2(A_10(CHAR),INT),INT)
```

La sesión ilustra como la función `insert_method` puede ser utilizada para introducir métodos en las clases influyendo en la presentación del árbol. Veamos un segundo ejemplo en el que eliminamos la presentación de la información asociada con los nodos:

```
  DB<8> insert_method(qw{FUNCTION TERMINAL}, 'info', sub {})
  DB<9> x $t->str
0  'PROGRAM(
      FUNCTION(RETURN(VARARRAY(TERMINAL,INDEXSPEC(INUM(TERMINAL))))),
      FUNCTION(RETURN(TIMES(VAR(TERMINAL),INUM(TERMINAL)))),
      FUNCTION(
        FUNCTIONCALL(TERMINAL,ARGLIST(VAR(TERMINAL),i
          FUNCTIONCALL(
            TERMINAL,
            ARGLIST(VARARRAY(TERMINAL,INDEXSPEC(INUM(TERMINAL),INUM(TERMINAL))))))))'
```

Código de Comprobación de las LLamadas

Sigue el código de comprobación de tipos de las llamadas:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ \
                           sed -ne '/^func/,234p' Trans.trg | cat -n
functioncall: FUNCTIONCALL($f, ARGLIST)
  => {
    # Before type checking attribute "t" has the declaration of $f
    my $ftype = $FUNCTIONCALL->{t};

     type_error(" Variable '".$f->value."' was not declared as function", $f->line)
    unless $ftype->isa("F");

    my @partypes = $ftype->child(0)->children;

    my @args = $ARGLIST->children;    # actual arguments
    my $numargs = @args;    # Number of actual arguments
    my $numpar = @partypes; # Number of declared parameters

    # Check number of args
      type_error("Function '".$f->value."' called with $numargs args expected $numpar",$f-
    if ($numargs != $numpar);

    # Check type compatibility between args
    # Do type cohercion if needed
    for (0..$#args) {
      my $pt = shift @partypes;
      my $ch = $ARGLIST->child($_);
      $ch = char2int($ARGLIST, $_) if $pt == $INT;
      $ch = int2char($ARGLIST, $_) if $pt == $CHAR;

      my $cht = $ch->{t};
      unless (array_compatible($cht, $pt)) {
        type_error(
          "Type of argument " .($_+1)." in call to " .$f->value." differs from expected",
          $f->line
```

```
        )
      }
    }

    # Now attribute "t" has the type of the node
    $FUNCTIONCALL->{t} = $ftype->child(1);
    return 1;
  }
```

Arrays que Difieren en la Primera Dimensión

La función `array_compatible` sigue la costumbre `C` de considerar compatibles arrays que difieren en la primera dimensión. Véase la conducta de `gcc` al respecto:

```
pl@nereida:~/Lbook/code/Simple-Types/script$ cat -n ArrayChecking.c
     1  #include <stdio.h>
     2
     3  void f(int a[3][3]) {
     4    a[1][2] = 5;
     5  }
     6
     7  main() {
     8    int b[2][3];
     9    f(b);
    10    printf("%d\n",b[1][2]);
    11  }
pl@nereida:~/Lbook/code/Simple-Types/script$ gcc ArrayChecking.c
pl@nereida:~/Lbook/code/Simple-Types/script$ a.out
5
```

Recordemos el código de la función `array_compatible`:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ sed -ne '96,124p' Trans.t
 1  { # support for arrays
 2
 3    sub compute_dimensionality {
..      .......................................................
14    }
15
16    sub is_array {
..      ....................................
20    }
21
22    sub array_compatible {
23      my ($a1, $a2) = @_;
24
25      return 1 if $a1 == $a2;
26      # int a[10][20] and int b[5][20] are considered compatibles
27      return (is_array($a1) && is_array($a2) && ($a1->child(0) == $a2->child(0)));
28    }
29  }
```

Ejemplos

```
pl@nereida:~/Lbook/code/Simple-Types/script$ usetypes.pl prueba07.c 2
```

```
 1 int a,b,e[10][20];
 2
 3 g() {}
 4
 5 int f(char c) {
 6 char d;
 7  c = 'X';
 8  g(e[d], 'A'+c);
 9  if (c > 0) {
10    int d;
11    d = a + b;
12  }
13  c = d * 2;
14  return c;
15 }
16
Type Error at line 8: Function 'g' called with 2 args expected 0

pl@nereida:~/Lbook/code/Simple-Types/script$ usetypes.pl prueba22.c 2
 1 int f(char a[10], int b) {
 2   return a[5];
 3 }
 4
 5 int h(int x) {
 6   return x*2;
 7 }
 8
 9 int g() {
10   char x[5];
11   int y[19][30];
12   f(x,h(y));
13 }
Type Error at line 12: Type of argument 1 in call to h differs from expected
```

6.13. Comprobación de Tipos: Sentencia `RETURN`

La Forma Habitual

Veamos la forma del árbol en una sentencia de retorno:

```
pl@nereida:~/Lbook/code/Simple-Types/script$ perl -wd usetypes.pl prueba16.c 2
main::(usetypes.pl:5):  my $filename = shift || die "Usage:\n$0 file.c\n";
  DB<1> f Types.eyp
1       2       3       4       5       6       7       #line 8 "Types.eyp"
8
9:      use strict;
10:     use Carp;
  DB<2> c 598
1 int f() {
2   int a[30];
3
4   return a;
5 }
Simple::Types::compile(Types.eyp:598):
598:     set_types($t);        # Init basic types
```

```
  DB<3> insert_method(qw{PROGRAM FUNCTION}, 'footnote', sub {})
  DB<4> p $t->str
PROGRAM(FUNCTION[f](RETURN(VAR(TERMINAL[a:4]))))
```

Código de Comprobación de Tipos para Sentencias de Retorno

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ \
                               sed -ne '267,282p' Trans.trg | cat -n
     1  returntype: RETURN($ch)
     2    => {
     3        my $rt = $RETURN->{t};
     4
     5        $ch = char2int($RETURN, 0) if $rt == $INT;
     6        $ch = int2char($RETURN, 0) if $rt == $CHAR;
     7
     8          type_error("Type error in return statement", $ch->line)
     9        unless ($rt == $ch->{t});
    10
    11        # $RETURN->{t} has already the correct type
    12
    13        $RETURN->type(ref($RETURN).ref($rt));
    14
    15        return 1;
```

El análisis de ámbito de las sentencias de retorno se hizo de forma ligeramente distinta al resto. El siguiente fragmento de código figura en la subrutina `compile` justo despueés del resto del análisis de ámbito::

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple$ \
                               sed -ne '585,595p' Types.eyp | cat -n
     1   # Scope Analysis: Return-Function
     2   our $retscope; # retscope: /FUNCTION|RETURN/
     3   my @returns = $retscope->m($t);
     4   for (@returns) {
     5     my $node = $_->node;
     6     if (ref($node) eq 'RETURN') {
     7       my $function = $_->father->node;
     8       $node->{function}  = $function;
     9       $node->{t} = $function->{t}->child(1);
    10     }
    11   }
```

Ejemplos

```
pl@nereida:~/Lbook/code/Simple-Types/script$ usetypes.pl prueba16.c 2
1 int f() {
2   int a[30];
3
4   return a;
5 }
Type Error at line 4: Type error in return statement
```

El retorno vacío por parte de una función no se considera un error en `C`:

```
pl@nereida:~/Lbook/code/Simple-Types/script$ usetypes.pl prueba26.c 2 | head -12
1 int f() {
2   int a[30];
3
4   return;
5 }

PROGRAM^{0}(
  FUNCTION[f]^{1}(
    EMPTYRETURN
  )
) # PROGRAM
---------------------------
```

Véase la conducta de `gcc` ante el mismo programa:

```
pl@nereida:~/Lbook/code/Simple-Types/script$ gcc -c prueba26.c
pl@nereida:~/Lbook/code/Simple-Types/script$ ls -ltr | tail -1
-rw-r--r-- 1 pl users  690 2008-01-10 13:32 prueba26.o
```

Usando `m`

En esta sección seguiremos una aproximación diferente. En este caso queremos maximizar nuestro conocimiento

Para comprobar la correción del tipo del valor retornado vamos a usar el método `m`. La razón para hacerlo es familiarizarle con el uso de `m`.

Usaremos una transformación árbol `bind_ret2function` a la que llamaremos (línea 33) desde `compile` después de haber verificado el resto de la comprobación de tipos (línea 30). Podemos postponer este análisis ya que el padre de un nodo `RETURN` es un nodo `FUNCTION` y por tanto no hay nodos cuya comprobación de tipos dependa de la computación de tipo de un nodo `RETURN`.

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> \
    sed -ne '519,554p' Types.eyp | cat -n
 1  sub compile {
 2   my($self)=shift;
 3
..  .........................................
29
30   $t->bud(@typecheck);
31
32   our $bind_ret2function;
33   my @FUNCTIONS = $bind_ret2function->m($t);
34
35   return $t;
36  }
```

`bind_ret2function` realiza la comprobación de tipos en los siguientes pasos:

1. La expresión regular árbol casa con cualquier nodo de definición de función. Para cada uno de esos nodos se procede como sigue:

2. Se buscan (línea 4) todas las sentencias `RETURN` en la función. Recuerde que `Parse::Eyapp::Node::m` retorna nodos Match y que los nodos del AST se obtienen usando el método `node` sobre el nodo Match (línea 5)

3. Ponemos como atributo de la función las referencias a los nodos `RETURN` encontrados (línea 8)

4. Para cada nodo de tipo `RETURN` encontrado:
    - *a*) Ponemos como atributo `function` del nodo la función que le anida (línea 15)
    - *b*) Se realizan las coherciones que fueran necesarias (líneas 17-19)
    - *c*) Se emite un mensaje de error si el tipo de la expresión evaluada (`$exp->{t}`) no coincide con el declarado en la función (`$return_type`) (líneas 21-22)
    - *d*) En la línea 25 cambiamos la clase del nodo de `RETURN` a `RETURNINT` o `RETURNCHAR` según sea el tipo retornado. Al igual que en las asignaciones eliminamos la sobrecarga semántica del nodo acercándonos a niveles mas bajos de programación.

Sigue el código. La línea 1 muestra una simple expresión regular árbol que casa con aquellas sentencias `RETURN` en las que se ha explicitado una expresión de retorno y que es usada por `bind_ret2function`.

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> \
                              sed -ne ’196,225p’ Trans.trg | cat -n
 1  /* TIMTOWTDI when MOPping */
 2  return: RETURN(.)
 3  bind_ret2function: FUNCTION
 4    => {
 5      my @RETURNS = $return->m($FUNCTION);
 6      @RETURNS = map { $_->node } @RETURNS;
 7
 8      # Set "returns" attribute for the FUNCTION node
 9      $FUNCTION->{returns} = \@RETURNS;
10
11      my $exp;
12      my $return_type = $FUNCTION->{t}->child(1);
13      for (@RETURNS) {
14
15        # Set "function" attribute for each RETURN node
16        $_->{function} = $FUNCTION;
17
18        #always char-int conversion
19        $exp = char2int($_, 0) if $return_type == $INT;
20        $exp = int2char($_, 0) if $return_type == $CHAR;
21
22          type_error("Returned type does not match function declaration",
23                                                                 $_->line)
24        unless $exp->{t} == $return_type;
25        $_->type("RETURN".ref($return_type));
26
27      }
28
29      return 1;
30    }
```

Obsérvese que la solución no optimiza la homogeneidad ni la generalidad ni el mantenimiento. Por ello, en condiciones normales de construcción de un compilador - como es el caso de la práctica 6.16 - es aconsejable abordar el análisis de tipos de la sentencia `RETURN` con la misma aproximación seguida para el resto de los nodos.

6.14. Comprobación de Tipos: Sentencias sin tipo

Usamos el tipo `$VOID` como tipo de aquellos constructos del lenguaje que carecen de tipo:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/lib/Simple> \
                                            sed -ne '67p' Trans.trg
statements: /STATEMENTS|PROGRAM|BREAK|CONTINUE/ => { $_[0]->{t} = $VOID }
```

6.15. Ejemplo de Árbol Decorado

Consideremos el programa de ejemplo:

```
nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-Types/script> cat -n  prueba23.c
  1  int f(char a[10], int b) {
  2    return a[5];
  3  }
  4
  5  int h(int x) {
  6    return x*2;
  7  }
  8
  9  int g() {
 10    char x[5];
 11    int y[19][30];
 12    f(x,h(y[1][1]));
 13  }
```

La siguiente tabla muestra (parcialmente) el árbol decorado resultante después de la fase de análisis de tipos:

Funciones `f` y `h`

```
PROGRAM^{0}(
  FUNCTION[f](
    RETURNINT(  # f(char a[10],
      CHAR2INT( #   return a[5];
        VARARRAY(  # a[5]
          TERMINAL[a:2],
          INDEXSPEC(
            INUM(
              TERMINAL[5:2]
            )
          ) # INDEXSPEC
        ) # VARARRAY
      ) # CHAR2INT
    ) # RETURNINT
  ) # FUNCTION,
  FUNCTION[h]( # int h(int x)
    RETURNINT( # return x*2
      TIMES[INT:6](
        VAR(
          TERMINAL[x:6]
        ),
        INUM(
          TERMINAL[2:6]
        )
      ) # TIMES
    ) # RETURNINT
  ) # FUNCTION,
```

Función `g`

```
  FUNCTION[g](
    FUNCTIONCALL( # f(x,h(y[1][1]))
      TERMINAL[f:12],
      ARGLIST(
        VAR( # char x[5]
          TERMINAL[x:12]
        ),
        FUNCTIONCALL( # h(y[1][1])
          TERMINAL[h:12],
          ARGLIST(
            VARARRAY( # y[1][1]
              TERMINAL[y:12],
              INDEXSPEC(
                INUM(
                  TERMINAL[1:12]
                ),
                INUM(
                  TERMINAL[1:12]
                )
              ) # INDEXSPEC
            ) # VARARRAY
          ) # ARGLIST
        ) # FUNCTIONCALL
      ) # ARGLIST
    ) # FUNCTIONCALL
  ) # FUNCTION
) # PROGRAM
```

6.16. Práctica: Análisis de Tipos en Simple C

Realize el analizador de tipos para Simple C de acuerdo a lo explicado en las secciones anteriores. Extienda el análisis de tipos a la extensión del lenguaje con `structs` Si lo desea puede usar una aproximación OOP estableciendo un método `checktypes` para cada clase/nodo. Para realizar esta alternativa es conveniente definir la jerarquía de clases adecuada.

6.17. Práctica: Análisis de Tipos en Simple C

Extienda el análisis de tipos de la práctica anterior para incluir punteros. (repase el resumen de la gramática de KR C en la página 507).

6.18. Práctica: Análisis de Tipos en Simple C con Gramáticas Atribuidas

Reescriba el analizador de tipos para Simple C utilizando `Language::AttributeGrammars`. Repase la sección **??** (*Usando* `Language::AttributeGrammars` *con* `Parse::Eyapp`).

Es conveniente que las acciones sean aisladas en subrutinas para evitar que `Language::AttributeGrammars` se confunda. Sigue un ejemplo de como podria hacerse:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-AttributeGrammar/lib$ \
```

```
                                     sed -ne '585,631p' Trans_Scheme.eyp
  my $attgram = new Language::AttributeGrammar <<'EOG';

# Tipos Basicos
INUM:                   $/.t = { Trans_Scheme::inum }
CHARCONSTANT:           $/.t = { Trans_Scheme::char_constant }
# Variables escalares y arrays
VAR:                    $/.t = { Trans_Scheme::var($/) }
# Expresiones binarias
PLUS:                   $/.t = { Trans_Scheme::bin($<0>.t, $<1>.t, $/) }
MINUS:                  $/.t = { Trans_Scheme::bin($<0>.t, $<1>.t, $/) }
TIMES:                  $/.t = { Trans_Scheme::bin($<0>.t, $<1>.t, $/) }
DIV:                    $/.t = { Trans_Scheme::bin($<0>.t, $<1>.t, $/) }
MOD:                    $/.t = { Trans_Scheme::bin($<0>.t, $<1>.t, $/) }
GT:                     $/.t = { Trans_Scheme::bin($<0>.t, $<1>.t, $/) }
GE:                     $/.t = { Trans_Scheme::bin($<0>.t, $<1>.t, $/) }
LE:                     $/.t = { Trans_Scheme::bin($<0>.t, $<1>.t, $/) }
EQ:                     $/.t = { Trans_Scheme::bin($<0>.t, $<1>.t, $/) }
NE:                     $/.t = { Trans_Scheme::bin($<0>.t, $<1>.t, $/) }
LT:                     $/.t = { Trans_Scheme::bin($<0>.t, $<1>.t, $/) }
AND:                    $/.t = { Trans_Scheme::bin($<0>.t, $<1>.t, $/) }
OR:                     $/.t = { Trans_Scheme::bin($<0>.t, $<1>.t, $/) }
# Sentencias de control
IF:                     $/.t = { Trans_Scheme::condition($<0>.t, $/, $<1>.t) }
IFELSE:                 $/.t = { Trans_Scheme::condition($<0>.t, $/, $<1>.t, $<2>.t) }
WHILE:                  $/.t = { Trans_Scheme::condition($<0>.t, $/, $<1>.t) }
# Asignaciones
ASSIGN:                 $/.t = { Trans_Scheme::assign($<0>.t, $<1>.t, $/) }
PLUSASSIGN:             $/.t = { Trans_Scheme::assign($<0>.t, $<1>.t, $/) }
MINUSASSIGN:            $/.t = { Trans_Scheme::assign($<0>.t, $<1>.t, $/) }
TIMESASSIGN:            $/.t = { Trans_Scheme::assign($<0>.t, $<1>.t, $/) }
DIVASSIGN:              $/.t = { Trans_Scheme::assign($<0>.t, $<1>.t, $/) }
MODASSIGN:              $/.t = { Trans_Scheme::assign($<0>.t, $<1>.t, $/) }
# Llamadas a funciones
FUNCTIONCALL:           $/.t = { Trans_Scheme::functioncall($/) }
# Return
RETURN:                 $/.t = { Trans_Scheme::return($<0>.t, $/) }
# Otros nodos
LABEL:                  $/.t = { Trans_Scheme::void }
GOTO:                   $/.t = { Trans_Scheme::void }
STATEMENTS:             $/.t = { Trans_Scheme::void }
BLOCK:                  $/.t = { Trans_Scheme::void }
BREAK:                  $/.t = { Trans_Scheme::void }
CONTINUE:               $/.t = { Trans_Scheme::void }
FUNCTION:               $/.t = { Trans_Scheme::void }
PROGRAM:                $/.t = { Trans_Scheme::void }

EOG
```

Las funciones de soporte son similares a las que hemos usado en `Trans.trg`:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Simple-AttributeGrammar/lib$ sed -ne '716,721p' Tr
sub return {
  my ($t, $fathernode) = @_;
```

```
  my $child = $fathernode->child(0);
  return $t if ($types->{$fathernode->{returntype}} == $t);
  type_error("Type error in return statement", $fathernode->line);
}
```

6.19. Práctica: Sobrecarga de Funciones en Simple C

Introduzca sobrecarga de funciones en Simple C y modifique el análisis de ámbito y de tipos. Repase las secciones 6.1 y 6.2.

6.20. Análisis de Tipos de Funciones Polimorfas

El uso de variables en las expresiones de tipo nos permite hablar de tipos desconocidos. Así un tipo como:

`F(LIST(TYPEVAR::ALPHA),TYPEVAR::ALPHA)`

nos permite hablar de una función que recibe listas de objetos de un tipo desconocido `TYPEVAR::ALPHA` y retorna objetos de ese mismo tipo `TYPEVAR::ALPHA`. Una variable de tipo representa un objeto no declarado o no completamente declarado. La *Inferencia de Tipos* es el problema de determinar el tipo de los nodos de uso en el árbol sintáctico abstracto cuando no se conocen total o parcialmente los tipos de los objetos en el programa fuente.

6.20.1. Un Lenguaje con Funciones Polimorfas

En esta sección introducimos un pequeño lenguaje que permite el polimorfismo paramétrico.

El Cuerpo

Los programas generados por esta gramática consisten en secuencias de declaraciones seguidas de secuencias de las expresiones cuyos tipos vamos a inferir. Por ejemplo:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Aho-Polymorphism/lib/Aho/script$ \
                                                             cat -n prueba01.ply
     1  first :  list(ALPHA) -> ALPHA;
     2  q : list(list(int))
     3  {
     4    first(first(q))
     5  }
```

Las declaraciones admiten variables de tipo. Las variables de tipo son identificadores formados por letras en mayúsculas. Los identificadores estan formados por minúsculas. La gramática que define el lenguaje es:

```
  p: d <+ ’;’> ’{’ e <%name EXPS + ’;’> ’}’
  ;

  d: id ’:’ t
  ;

  t:  INT
    | STRING
    | TYPEVAR
    | LIST ’(’ t ’)’
    | t ’*’ t
    | t ’->’ t
```

```
  | ’(’ t ’)’
;

e:  id ’(’ optargs ’)’
  | id
;

optargs:
    /* empty */
  | args
;

args: e
   | args ’,’ e
;

id: ID
;
```

Para las expresiones nos limitamos a construir el árbol usando bypass automático. Las expresiones se reducen al uso de identificadores y llamadas a función:

```
p:
    d <+ ’;’> ’{’ e <%name EXPS + ’;’>.expressions ’}’
      {
        $expressions->{symboltable} = \%st;
        $expressions
      }
;

d:
  $id ’:’ $t
    {
      $st{$id->key} = { ts => $t };
    }
;

t:
    INT
      {
        ’INT’;
      }
  | STRING
      {
        ’STRING’;
      }
  | TYPEVAR
      {
        "Parse::Eyapp::Node::TYPEVAR::$_[1]->[0]"
      }
  | LIST ’(’ $t ’)’
      {
        "LIST($t)"
      }
```

```
 | t.left ’*’ t.right
   {
    "X($left,$right)"
   }
 | t.domain ’->’ t.image
   {
    "F($domain,$image)"
   }
 | ’(’ $t ’)’
   {
    $t;
   }
;

e:
  %name FUNCTIONCALL
   id ’(’ optargs ’)’
 | id
;

optargs:
   /* empty */
  | args
;

args:
   e
 | %name ARGS
   args ’,’ e
;

id:
  %name ID
   ID
;
```

Para cada declaración se construye un término que describe el tipo y se almacena en la tabla de símbolos `%st`. Por ejemplo, las expresiones de tipo construidas a partir de las declaraciones del programa anterior son:

```
first type: F(LIST(Parse::Eyapp::Node::TYPEVAR::ALPHA),Parse::Eyapp::Node::TYPEVAR::ALPHA)
q type: LIST(LIST(INT))
```

Usaremos el espacio de nombres `Parse::Eyapp::Node::TYPEVAR` para representar los nodos variable. El árbol resultante para el programa anterior es:

```
first :  list(ALPHA) -> ALPHA;
q : list(list(int))
{
  first(first(q))
}
```

```
EXPS(
  FUNCTIONCALL(
    ID[first],
    FUNCTIONCALL(
      ID[first],
      ID[q]
    )
  ) # FUNCTIONCALL
) # EXPS
```

Obsérvese que no hay asignación ni operaciones binarias en el lenguaje (aunque estas últimas pueden ser emuladas mediante funciones `add`, `times`, etc.). Por ello la comprobación de tipos se centra en las llamadas a función.

La Cola

En la cola figuran la subrutina de análisis léxico y la de tratamiento de errores. Son una adaptación de las presentadas en las secciones anteriores para `Simple::Types`.

El análisis de tipo para este lenguaje con variables de tipo lo haremos usando expresiones regulares árbol (fichero `Trans.trg`) asi como un módulo que implanta el algoritmo de unificación y que hemos llamado `Aho::Unify`. En concreto tenemos los siguientes ficheros en la librería:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Aho-Polymorphism/lib/Aho$ ls -l
total 112
-rw-r--r-- 1 pl users   639 2008-01-07 13:01 Makefile
-rw-r--r-- 1 pl users  3837 2008-01-11 16:47 Polymorphism.eyp
-rw-r--r-- 1 pl users   945 2008-01-11 12:20 Trans.trg
-rw-r--r-- 1 pl users  3212 2008-01-11 12:57 Unify.pm
```

Los ficheros son compilados con los comandos:

```
treereg -nonumbers -m Aho::Polymorphism Trans.trg
eyapp -m Aho::Polymorphism Polymorphism.eyp
```

La llamada `$t = $self->YYParse` de la línea 192 realiza el análisis sintáctico y el análisis de ámbito. El análisis de ámbito es trivial ya que hay un sólo ámbito.

```
115  %%
...  ..................................
186  sub compile {
187   my($self)=shift;
188
189   my ($t);
190
191   $self->YYData->{INPUT} = $_[0];
192
193   $t = $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error,
194                        #yydebug => 0x1F
195       );
196
197   my @typecheck = (
198     our $functioncall,
199     our $tuple,
200     our $id,
201   );
```

```
202
203   Aho::Unify->set(
204     key    => ’set’,
205     isvar  => sub { $_[0] =~ /^Parse::Eyapp::Node::TYPEVAR/ },
206     samebasic => sub {
207       my ($s, $t) = @_;
208
209       return (((ref($s) eq ’INT’) || (ref($s) eq’STRING’)) && (ref($s) eq ref($t)));
210     },
211     debug => $debug,
212   );
213
214   $t->bud(@typecheck);
215
216   return $t;
217  }
218
219  sub ID::key {
220    return $_[0]->{attr}[0]
221  }
222
223  insert_method(’ID’, ’info’, \&ID::key);
```

Las líneas de la 197 a la 212 se encargan de la inferencia de tipos. Volveremos sobre ellas mas adelante. Por ahora observe que hay tres transformaciones:

- `$functioncall` se encarga de la inferencia y comprobación para las llamadas,
- `$tuple` del cálculo de tipos para las expresiones `args, exp` que aparecen en los argumentos. Estas expresiones dan lugar a árboles de la forma `ARGS($x, $y)`. Por ejemplo, en el siguiente programa:

  ```
  f :  ALPHA * BETA * GAMMA -> ALPHA;
  q : int;
  r : int;
  s : list(int)
  {
    f(q,r,s)
  }
  ```

  El árbol construido para la expresión es:

  ```
  EXPS(FUNCTIONCALL(ID[f],ARGS(ARGS(ID[q],ID[r]),ID[s])))
  ```

  mientras que los tipos son:

  ```
  r type: INT
  q type: INT
  s type: LIST(INT)
  f type: F(
            X(
              X(
                Parse::Eyapp::Node::TYPEVAR::ALPHA,
                Parse::Eyapp::Node::TYPEVAR::BETA
  ```

```
                 ),
                 Parse::Eyapp::Node::TYPEVAR::GAMMA
               ),
               Parse::Eyapp::Node::TYPEVAR::ALPHA
             )
```

- La tercera transformación en `Trans.trg` es `$id`, la cual se encarga del cálculo de los tipos para los nodos `ID`. Como las anteriores, esta transformación decorará el nodo con un atributo `t` denotando el DAG que describe al tipo. En este caso se tomará el tipo de la tabla de símbolos y se crearán nuevas instancias de las variables de tipo si las hubiera.

El comportamiento del algoritmo de unificación depende de un conjunto de parámetros que es inicializado mediante la llamada a la función `set` del módulo `Aho::Unify`.

La llamada `$t->bud(@typecheck)` aplica las transformaciones en un recorrido abajo-arriba infiriendo los tipos y decorando los nodos.

La Cabecera

La cabecera muestra la carga de los módulos para la unificación (`Aho::Unify`) y para la inferencia y comprobación de tipos (`Aho::Trans`).

La directiva `%strict`

Por defecto `Parse::Eyapp` decide que aquellos símbolos que no figuran en la parte izquierda de una regla son terminales (tokens). La directiva `%strict` utilizada en la línea 7 fuerza la declaración de todos los terminales. Cuando se usa la directiva `%strict` el compilador `eyapp` emite un mensaje de advertencia si detecta algún símbolo que no figura en la parte izquierda de alguna regla y no ha sido explícitamente declarado como terminal (mediante directivas como `%token`, `%left`, etc.).

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Aho-Polymorphism/lib/Aho$ \
                                             cat -n Polymorphism.eyp
  1  /*
  2  File: Aho/Polymorphism.eyp
  3  To build it, Do make or:
  4    eyapp -m Aho::Polymorphism  Polymorphism.eyp;
  5    treereg -m Aho::Polymorphism Trans.trg
  6  */
  7  %strict
  8  %{
  9  use strict;
 10  use Carp;
 11  use warnings;
 12  use Aho::Unify qw(:all);
 13  use Aho::Trans;
 14  use Data::Dumper;
 15  our $VERSION = "0.3";
 16
 17  my $debug = 1;
 18  my %reserved = (
 19    int => "INT",
 20    string => "STRING",
 21    list => "LIST",
 22  );
 23
 24  my %lexeme = (
 25    ','  => "COMMA",
```

```
  ';'  => "SEMICOLON",
  ':'  => "COLON",
  '*'  => "TIMES",
  '('  => "LEFTPARENTHESIS",
  ')'  => "RIGHTPARENTHESIS",
  '{'  => "LEFTCURLY",
  '}'  => "RIGHTCURLY",
  '->' => "FUNCTION",
);
my ($tokenbegin, $tokenend) = (1, 1);
our %st; # Symbol table
%}

%tree bypass

%token ID TYPEVAR STRING INT LIST

%right '->'
%left '*'

%%
```

Ambiguedades

La regla de producción `t -> t '*' t` es obviamente ambigua. La directiva `%left '*'` permite deshacer la ambiguedad. Lo mismo ocurre con la regla para las funciones `t -> t '->' t`. En este caso nos decidimos por una asociatividad a derechas, de modo que una declaración como `int -> int -> int` es interpretado como una función que recibe enteros y devuelve funciones. Una tercera fuente de ambiguedad se produce en expresiones como:

```
int * int -> int
```

que puede ser interpretado como `int * (int -> int)` o bien `(int * int) -> int`. Al darle mayor prioridad al `*` nos decidimos por la segunda interpretación.

6.20.2. La Comprobación de Tipos de las Funciones Polimorfas

Reglas a Tener en Cuenta en la Inferencia de Tipos

Las reglas para la comprobación de tipos para funciones polimorfas difiere del de las funciones ordinarias en varios aspectos. Consideremos de nuevo el árbol para la llamada `first(first(q))` decorado con los tipos que deberan ser inferidos:

```
first :  list(ALPHA) -> ALPHA;
q : list(list(int))
{
  first(first(q))
}
```

```
EXPS(
  FUNCTIONCALL[int](
    ID[first:F(LIST(INT),INT)],
    FUNCTIONCALL[list(int)](
      ID[first:F(LIST(LIST(INT)) LIST(INT))],
      ID[q:LIST(LIST(INT))]
    )
  ) # FUNCTIONCALL
) # EXPS
```

Del ejercicio de hacer la inferencia para el ejemplo sacamos algunas consecuencias:

1. Distintas ocurrencias de una función polimorfa en el AST no tienen que necesariamente tener el mismo tipo.

   En la expresión `first(first(q))` la instancia interna de la función `first` tiene el tipo

   ```
   list(list(int)) -> list(int)
   ```

   mientras que la externa tiene el tipo

   ```
   list(int) -> int
   ```

   Se sigue que cada ocurrencia de `first` tiene su propia visión de la variable `ALPHA` que se instancia (se *unifica*) a un valor distinto. Por tanto, en cada ocurrencia de `first`, reemplazaremos la variable de tipo `ALPHA` por una variable *fresca* `ALPHA`. Así habrán variables frescas `ALPHA` para los nodos interno y externo de las llamadas `FUNCTIONCALL(ID[first], ...)`

2. Puesto que ahora las expresiones de tipo contienen variables debemos extender la noción de equivalencia de tipos. Supongamos que una función `f` de tipo `ALPHA -> ALPHA` es aplicada a una expresión `exp` de tipo `list(GAMMA)`. Es necesario *unificar* ambas expresiones. En este caso obtendriámos que `ALPHA = list(GAMMA)` y que `f` es una función del tipo `list(GAMMA) -> list(GAMMA)`

3. Es necesario disponer de un mecanismo para grabar el resultado de la unificación de dos árboles/dags. Si la unificación de dos expresiones de tipo `s` y `s'` resulta en la variable `ALPHA` representando al tipo `t` entonces `ALPHA` deberá seguir representando al tipo `t` conforme avancemos en la comprobación de tipos.

Programa Árbol para la Inferencia de Tipos

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Aho-Polymorphism/lib/Aho$ \
                                                       cat -n Trans.trg
/***************** Polymorphic Type Checking *****************/
  /*
  eyapp -m Aho::Polymorphism  Polymorphism.eyp;
  treereg -m Aho::Polymorphism Trans.trg
  */

{
  use Aho::Unify qw(:all);
}

functioncall: FUNCTIONCALL($f, $arg)
  => {
    my ($name, $line) = @{$f->{attr}};

    my $p = new_var();

    my $q = Parse::Eyapp::Node->hexpand("F", $arg->{t}, $p, \&selfrep);

    if (unify($f->{t}, $q)) {
      print "Unified\n";
      print "Now type of ".$f->str." is ".strunifiedtree($q)."\n";
    }
    else {
      die "Type error at $line\n";
    }

    $FUNCTIONCALL->{t} = representative($p);
    print "Type of ".$FUNCTIONCALL->str." is ".$FUNCTIONCALL->{t}->str."\n";
```

```

    return 1;
  }

tuple: ARGS($x, $y)
  => {
    $ARGS->{t} = Parse::Eyapp::Node->hexpand("X", $x->{t}, $y->{t},  \&selfrep);
  }

id: ID
  => {
    my ($name, $line) = @{$ID->{attr}};
    our %st;
    my $ts = $st{$name}->{ts};
    $ID->{t} = fresh($ts);
  }
```

- La regla para los identificadores accede al término guardado en la tabla de símbolos que describe el tipo del identificador (atributo `ts`). A partir de el se construye el DAG que representa el tipo llamando a la funcion `fresh` de `Aho::Unify`. Esta función antes de llamar a `hnew` ”refresca” todas las apariciones de variables de tipo. Una variable como `ALPHA` será sustituida por `ALPHA#number`.

  ```
  sub fresh {
    my $ts = shift;
    my $regexp = shift;
    $regexp = ’Parse::Eyapp::Node::TYPEVAR::[\w:]+’ unless defined($regexp);

    $ts =~ s/($regexp)/$1$count/g;
    $count++;
    my @t = Parse::Eyapp::Node->hnew($ts);
    representative($_, $_) for @t;
    return $t[0];
  }
  ```

  El algoritmo de unificación decora los nodos de los DAGs que estan siendo unificados con un atributo (atributo cuyo nombre viene dado por la variable `$set`) que es una referencia al representante de la clase de equivalencia a la que pertenece el nodo. Cada clase tiene un único representante al que denominaremos representante canónico. El representante de un nodo puede obtenerse/modificarse usando la función `representative` proveída por `Aho::Unify`.

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

    if (@_) {
      $t->{$set} = shift;
      return $t;
    }
    $t = $t->{$set} while defined($t->{set}) && ($t != $t->{$set});
    die "Representative ($set) not defined!".Dumper($t) unless defined($t->{set});
    return $t;
  }
  ```

  Al comienzo del algoritmo cada nodo pertenece a una clase en la que el único nodo es el mismo. Conforme se avanza en el proceso de unificación las clases van aumentando de tamaño.

- La acción asociada con la regla `tuple` correspondiente a expresiones de la forma `x, y` en los argumentos es construir el tipo producto cartesiano de los tipos de `x` e `y`.

```
tuple: ARGS($x, $y)
  => {
    $ARGS->{t} = Parse::Eyapp::Node->hexpand("X", $x->{t}, $y->{t},  \&selfrep);
  }
```

El método `hexpand` de `Parse::Eyapp::Node` permite construir un DAG compatible con los construidos con `hnew` ya que utiliza la misma cache de memoización. El método admite como último argumento un manejador-decorador de atributos que funciona de manera parecida a como lo hace el correspondiente argumento en `new` y `hnew`. El manejador es llamado por `hexpand` con el nodo que acaba de ser creado como único argumento.

Ejercicio 6.20.1. *Podíamos construir el nodo mediante una llamada* `Parse::Eyapp:Node->new('X')` *y expandiendo el nodo con hijos* `$x` *y* `$y`*. Sin embargo, este método no conservaría la propiedad de DAG de los grafos de tipo. ¿Porqué?*

Ejercicio 6.20.2. *¿Que ocurriría si el nodo se creara con* `Parse::Eyapp:Node->hnew('X')` *y luego se expandiera con hijos* `$x` *y* `$y`*?*

Ejercicio 6.20.3. *¿En que forma sin usar* `hexpand` *podríamos formar un DAG compatible con* `hnew` *para el árbol* `X($x->{t}->str, $y->{t}->str)`*?*

La función `selfrep` - también en `Aho::Unify` - tiene por función iniciar el representante del nodo al propio nodo. De hecho su código es:

```
sub selfrep {
  representative($_[0], $_[0])
};
```

- La regla `functioncall` para las llamadas `FUNCTIONCALL($f, $arg)` parte de que el tipo mas general asociado con esta clase de nodo debe tener la forma `F($arg->{t},$p)` donde `$p` es una nueva variable libre.

```
    my ($name, $line) = @{$f->{attr}};

    my $p = new_var();

    my $q = Parse::Eyapp::Node->hexpand("F", $arg->{t}, $p, \&selfrep);
```

La función `new_var` (en `Aho::Unify`) proporciona una nueva variable:

```
# The name space Parse::Eyapp::Node::TYPEVAR:: is reserved for tree variables
# Parse::Eyapp::Node::TYPEVAR::_#number is reserved for generated variables
sub new_var {
  my $p = Parse::Eyapp::Node->hnew("Parse::Eyapp::Node::TYPEVAR::_$count");
  representative($p, $p);
  $count++;
  return $p;
}
```

Las variables proveídas tiene la forma `Parse::Eyapp::Node::TYPEVAR::_#number`.

La llamada al método `hexpand` se encarga de crear el DAG asociado con `F($arg->{t},$p)`. El último argumento de la llamada a `hexpand` es el manejador de atributos. Pasamos `\&selfrep` para indicar que se trata de un árbol no unificado.

Acto seguido se pasa a *unificar* el tipo de `$f` y el tipo mas general `F($arg->{t},$p)`.

```
    if (unify($f->{t}, $q)) {
      print "Unified\n";
      print "Now type of ".$f->str." is ".strunifiedtree($q)."\n";
    }
    else {
      die "Type error at line $line\n";
    }
```

Si la unificación tiene éxito el tipo asociado con la imagen (con el valor retornado) debe ser el tipo unificado de la nueva variable `$p` que es accedido por medio de la función `representative`.

```
    $FUNCTIONCALL->{t} = representative($p);
```

- Cada vez que la unificación tiene éxito el programa imprime el árbol unificado mediante `strunifiedtree`. La función `strunifiedtree` (en `Aho::Unify`) recorre el árbol de representantes construyendo el término que lo describe:

```
sub strunifiedtree {
  my $self = shift;

  my $rep = representative($self);
  my @children = $rep->children;
  my $desc = ref($rep);
  return $desc unless @children;

  my @d;
  for (@children) {
    push @d, strunifiedtree($_);
  }
  local $" = ",";
  $desc .= "(@d)";
  return $desc;
}
```

Ejemplo

La ejecución del compilador con el programa que hemos usado como ejemplo muestra el proceso de inferencia. Empecemos repasando la estructura del árbol y la representación de las declaraciones:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Aho-Polymorphism/lib/Aho/script$ \
                                                   usepoly.pl prueba01.ply 2
first type: F(LIST(Parse::Eyapp::Node::TYPEVAR::ALPHA),Parse::Eyapp::Node::TYPEVAR::ALPHA)
q type: LIST(LIST(INT))
first :  list(ALPHA) -> ALPHA;
q : list(list(int))
{
  first(first(q))
}
```

```
....
Tree:
EXPS(
  FUNCTIONCALL(
    ID[first],
    FUNCTIONCALL(
      ID[first],
      ID[q]
    )
  ) # FUNCTIONCALL
) # EXPS
```

En primer lugar se calculan los tipos para la llamada interna:

```
Unifying F(LIST(TYPEVAR::ALPHA1),TYPEVAR::ALPHA1) and F(LIST(LIST(INT)),TYPEVAR::_3)
Unifying LIST(TYPEVAR::ALPHA1) and LIST(LIST(INT))
Unifying TYPEVAR::ALPHA1 and LIST(INT)
TYPEVAR::ALPHA1 = LIST(INT)
Unifying LIST(INT) and TYPEVAR::_3
TYPEVAR::_3 = LIST(INT)
Unified
Now type of ID[first] is F(LIST(LIST(INT)),LIST(INT))
Type of FUNCTIONCALL(ID[first],ID[q]) is LIST(INT)
```

Asi se ha inferido que el tipo del argumento de la llamada externa a `first` es `LIST(INT)`. La inferencia para el tipo del nodo externo `FUNCTIONCALL` prosigue con una nueva variable fresca `ALPHA0`:

```
Unifying F(LIST(TYPEVAR::ALPHA0),TYPEVAR::ALPHA0) and F(LIST(INT),TYPEVAR::_4)
Unifying LIST(TYPEVAR::ALPHA0) and LIST(INT)
Unifying TYPEVAR::ALPHA0 and INT
TYPEVAR::ALPHA0 = INT
Unifying INT and TYPEVAR::_4
TYPEVAR::_4 = INT
Unified
Now type of ID[first] is F(LIST(INT),INT)
Type of FUNCTIONCALL(ID[first],FUNCTIONCALL(ID[first],ID[q])) is INT
```

Por último el programa nos muestra el árbol sintáctico abstracto con los tipos inferidos:

```
Tree:
EXPS(
  FUNCTIONCALL[INT](
    ID[first:F(LIST(INT),INT)],
    FUNCTIONCALL[LIST(INT)](
      ID[first:F(LIST(LIST(INT)),LIST(INT))],
      ID[q:LIST(LIST(INT))]
    )
  ) # FUNCTIONCALL
) # EXPS
```

Un Ejemplo Con Especificación de Tipos Incompleta en C

La expresión `g(g)` en el siguiente programa C muestra la aplicación de una función a ella misma. La declaración de la línea 7 define la imagen de `g` como entera pero el tipo de los argumentos queda sin especificar.

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Aho-Polymorphism/lib/Aho/script$ \
                                                           cat -n macqueen.c
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Aho-Polymorphism/lib/Aho/script$ cat -n macqueen.c
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  int n = 5;
 5
 6  int f(int g())
 7  {
 8    int m;
 9
10    m = n;
11    if (m == 0) return 1;
12    else {
13      n = n - 1;
14      return m * g(g);
15    }
16  }
17
18  main() {
19    printf("%d factorial is %d\n", n, f(f));
20  }
```

Al ser compilado con `gcc` el programa no produce errores ni mensajes de advertencia:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Aho-Polymorphism/lib/Aho/script$ gcc macqueen.c
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Aho-Polymorphism/lib/Aho/script$ a.out
0 factorial is 120
```

El error que se aprecia en la salida se corrige cuando cambiamos el orden de los argumentos en la llamada a `printf`:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Aho-Polymorphism/lib/Aho/script$ \
                    sed -ne '/printf/p' macqueen2.c; gcc macqueen2.c ; a.out
  printf("%d is the factorial of %d\n", f(f), n);
120 is the factorial of 5
```

¿Cuál es la especificación completa del tipo de `g`?

Este ejemplo esta basado en un programa Algol dado por Ledgard en 1971 [8]. Aparece como ejercicio 6.31 en el libro de Aho, Sethi y Ullman [6].

El siguiente programa en nuestro pequeño lenguaje polimorfo modela el problema de determinar el tipo de esta función:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Aho-Polymorphism/lib/Aho/script$ \
                                               usepoly.pl exercise6_31.ply 2
m     : int;
times : int * int -> int;
g     : ALPHA
{
  times(m, g(g))
}
```

El árbol y la tabla de símbolos quedan montados después de la primera fase:

```
Tree:
```

```
EXPS(
  FUNCTIONCALL(
    ID[times],
    ARGS(
      ID[m],
      FUNCTIONCALL(
        ID[g],
        ID[g]
      )
    ) # ARGS
  ) # FUNCTIONCALL
) # EXPS
g type: Parse::Eyapp::Node::TYPEVAR::ALPHA
m type: INT
times type: F(X(INT,INT),INT)
```

En primer lugar se procede a la unificación de la llamada interna `FUNCTIONCALL(ID[g],ID[g])`:

```
Unifying TYPEVAR::ALPHA2 and F(TYPEVAR::ALPHA3,TYPEVAR::_4)
TYPEVAR::ALPHA2 = F(TYPEVAR::ALPHA3,TYPEVAR::_4)
Unified
Now type of ID[g] is F(Parse::Eyapp::Node::TYPEVAR::ALPHA3,Parse::Eyapp::Node::TYPEVAR::_4)
Type of FUNCTIONCALL(ID[g],ID[g]) is TYPEVAR::_4
```

Después se pasa a la unificación de la llamada externa `FUNCTIONCALL(ID[times],ARGS(ID[m], FUNCTIONCALL(..`

```
Unifying F(X(INT,INT),INT) and F(X(INT,TYPEVAR::_4),TYPEVAR::_5)
Unifying X(INT,INT) and X(INT,TYPEVAR::_4)
Unifying INT and TYPEVAR::_4
TYPEVAR::_4 = INT
Unifying INT and TYPEVAR::_5
TYPEVAR::_5 = INT
Unified
Now type of ID[times] is F(X(INT,INT),INT)
Type of FUNCTIONCALL(ID[times],ARGS(ID[m],FUNCTIONCALL(ID[g],ID[g]))) is INT
```

Vemos que el tipo de `g` es `F(Parse::Eyapp::Node::TYPEVAR::ALPHA3,Parse::Eyapp::Node::TYPEVAR::_4)` y que `TYPEVAR::_4 = INT`. Asi pues el tipo `ALPHA` de `g` es `F(ALPHA,INT)`. Por tanto tenemos que el tipo inferido para `g` es un tipo recursivo:

```
typedef int ALPHA(ALPHA);
```

El programa nos muestra el árbol anotado con los tipos inferidos:

```
Tree:
EXPS(
  FUNCTIONCALL[INT](
    ID[times:F(X(INT,INT),INT)],
    ARGS[X(INT,INT)](
      ID[m:INT],
      FUNCTIONCALL[INT](
        ID[g:F(Parse::Eyapp::Node::TYPEVAR::ALPHA3,INT)],
        ID[g:Parse::Eyapp::Node::TYPEVAR::ALPHA3]
      )
    ) # ARGS
  ) # FUNCTIONCALL
) # EXPS
```

Inferencia de una Variable Conocido el Tipo de la Función

La decoración ocurre de abajo-arriba y la inferencia sólo se hace en los nodos de llamada. Sin embargo, Se puede producir la inferencia del tipo de una variable después de que su nodo haya sido visitado, como muestra el ejemplo:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Aho-Polymorphism/lib/Aho/script$ \
                                           usepoly.pl reverseinference.ply 2
first :  int -> int;
q : ALPHA
{
  first(q)
}
Unifying F(INT,INT) and F(TYPEVAR::ALPHA1,TYPEVAR::_2)
Unifying INT and TYPEVAR::ALPHA1
TYPEVAR::ALPHA1 = INT
Unifying INT and TYPEVAR::_2
TYPEVAR::_2 = INT
Unified
Now type of ID[first] is F(INT,INT)
Type of FUNCTIONCALL(ID[first],ID[q]) is INT
Tree:
EXPS(
  FUNCTIONCALL[INT](
    ID[first:F(INT,INT)],
    ID[q:INT]
  )
) # EXPS
first type: F(INT,INT)
q type: Parse::Eyapp::Node::TYPEVAR::ALPHA
```

No obstante, parece razonable que el tipo de una variable que no sea una función debería ser fijo. Dado que las ventajas del polimorfismo se aprecian fundamentalmente en las funciones podemos añadir a nuestro lenguaje reglas que prohiban el polimorfismo de variables que no sean funciones.

6.20.3. El Compilador

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Aho-Polymorphism/lib/Aho/script$ cat -n  usepoly.p
#!/usr/bin/perl -I../lib -w
use strict;
use Aho::Polymorphism;
use Aho::Unify qw(:all);
use Parse::Eyapp::Base qw{:all};
use Data::Dumper;

# Read Input
my $filename = shift || die "Usage:\n$0 file.c\n";
my $debug = shift;
$debug = 0 unless defined($debug);
my $input = slurp_file($filename, "ply");
print $input if $debug;

# Syntax Analysis and Type Inference
my $parser = Aho::Polymorphism->new();
my $t = $parser->compile($input);
```

```

# Show decorated tree
$Parse::Eyapp::Node::INDENT = $debug;
push_method(qw{FUNCTIONCALL ARGS}, info => sub { strunifiedtree($_[0]->{t}) });
push_method(qw{ID}, info => sub { $_[0]->{attr}[0].":".strunifiedtree($_[0]->{t}) });
print "Tree: ".$t->str."\n";

# Print Symbol Table
my $symboltable = $t->{symboltable};
for (keys(%$symboltable)) {
  print "$_ type: "
        . $symboltable->{$_}{ts}
        ."\n";
}
```

6.20.4. Un Algoritmo de Unificación

El algoritmo de unificación recibe dos referencias `$m` y `$n` a los árboles que van a ser unificados. Retorna verdadero si la unificación puede tener lugar y falso en caso contrario. La equivalencia se mantiene utilizando un atributo cuyo nombre es `$set`. Todos los nodos en una clase de equivalencia tienen un único representante. Los atributos `$set` de los nodos en una misma clase referencian al representante (posiblemente de forma indirecta, via los campos `$set` de otros nodos). El atributo `$set` del representante canónico apunta a si mismo. Inicialmente cada nodo esta en una clase única.

```
sub unify {
  my ($m, $n) = @_;

  my $s = representative($m);
  my $t = representative($n);

  return 1 if ($s == $t);

  return 1 if $samebasic->($s, $t);

  print "Unifying ".representative($s)->str." and ".representative($t)->str."\n" if $debu
  return 1 if (mergevar($s, $t));

  if (ref($s) eq ref($t)) {
     $s->{$set} = representative($t);
     my $i = 0;
     for ($s->children) {
       my $tc = $t->child($i++);
       return 0 unless unify($_, $tc);
     }
     return 1;
  }

  return 0;
}
```

Nótese que unificar `$m` y `$n` es unificar sus representantes canónicos `$s` y `$t`. Los representantes serán iguales si `$m` y `$n` ya están en la misma clase de equivalencia.

El algoritmo de unificación usa las siguientes funciones auxiliares:

1. La función `representative` retorna una referencia al representante ccanónico de la clase:

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

  if (@_) {
    $t->{$set} = shift;
    return $t;
  }
  $t = $t->{$set} while defined($t->{set}) && ($t != $t->{$set});
  die "Representative ($set) not defined!".Dumper($t) unless defined($t->{set});
  return $t;
}
```

También permite cambiar el representante.

2. Si `$s` y `$t` representan el mismo tipo básico la función referenciada por `$samebasic` devolverá verdadero. La función `$samebasic` es definida por medio de la función `set` junto con los otros parámetros del algoritmode unificación.

   ```
   Aho::Unify->set(
     key    => ’set’,
     isvar  => sub { $_[0] =˜ /ˆParse::Eyapp::Node::TYPEVAR/ },
     samebasic => sub {
       my ($s, $t) = @_;

       return (((ref($s) eq ’INT’) || (ref($s) eq’STRING’)) && (ref($s) eq ref($t)));
     },
     debug => $debug,
   );
   ```

3. Si una de los dos árboles `$s` o `$t` es una variable la introducimos en la clase de equivalencia de la otra. Puesto que cada variable fresca es un único nodo (estamos trabajando con un DAG) la actualización de la clase de equivalencia será visible en todas las apariciones de esta variable. Además `representative` recorre la lista de enlaces de unificación retornando el tipo básico o constructor con el que la variable se ha unificado.

   La función `mergevar` mezcla las clases de equivalencia cuando uno de los representantes es una variable:

   ```
   sub mergevar {
     my ($s, $t) = @_;

     if (isvar($s))  {
       $s->{$set} = representative($t);
       print $s->str." = ".representative($t)->str."\n" if $debug;
       return 1;
     }
     if (isvar($t)) {
       $t->{$set} = representative($s);
       print $t->str." = ".representative($s)->str."\n" if $debug;
       return 1;
     }
     return 0;
   }
   ```

4. El programador puede proporcionar mediante el método `set` una función `$isvar` que permite añadir variables adicionales:

   ```
   sub isvar {
     my $x = $isvar->(@_);
     return $x if $x;
     return 1 if $_[0] =~ /^Parse::Eyapp::Node::TYPEVAR::[\w:]+$/;
   }
   ```

   El espacio de nombres `Parse::Eyapp::Node::TYPEVAR` se usa para las variables.

La Cabecera del Módulo de Unificación

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK/code/Aho-Polymorphism/lib/Aho$ cat -n Unify.pm
package Aho::Unify;
use Data::Dumper;
use Parse::Eyapp::Node;
use base qw (Exporter);
our @EXPORT_OK = qw(
  unify
  new_var
  fresh
  representative
  strunifiedtree
  hnewunifiedtree
  newunifiedtree
  selfrep
);
our %EXPORT_TAGS = ( 'all' => [ @EXPORT_OK ] );

my $count = 0;
my $set = 'representative';
my $isvar = sub { };
my $samebasic = sub { };
my $debug = 0;
```

La Función `set`

```
sub set {
   my $class = shift if @_ %2;
   $class = __PACKAGE__ unless defined($class);

   my %handler = @_;

   $set = 'representative';
   $set = $handler{key} if exists($handler{key});
   $isvar = $handler{isvar} if exists($handler{isvar});
   $samebasic = $handler{samebasic} if exists($handler{samebasic});
   $debug = $handler{debug} if exists($handler{debug});
   $count = 0;

   bless { key => $set, isvar => $isvar, samebasic => $samebasic, count => $count }, $cla
}
```

6.21. Práctica: Inferencia de Tipos

Extienda el lenguaje con tipos variables presentado en la sección 6.20 con el tipo puntero:

```
q : pointer(string);
```

Suponga añadida la función polimorfa `deref(x)` que devuelve lo apuntado por `x`. ¿Cuál es el tipo de `deref`? Extienda las expresiones con constantes y operaciones binarias. Añada código para emitir un mensaje de error en el caso de que después de la fase de inferencia alguna variable - que no sea del tipo función - tenga un tipo polimorfo. Actualice las entradas en la tabla de símbolos para las funciones con un atributo que recoge los diferentes tipos inferidos de los usos de la función.

Capítulo 7

Instrucciones Para la Carga de Módulos en la ETSII

Cuando entra a una de las máquinas de los laboratorios de la ETSII encontrará montado el directorio `/soft/perl5lib/` y en él algunas distribuciones de módulos Perl que he instalado.

- Perl nativo (5.8.* a 5.9.*, Perl que viene con la distribución de Linux)

  Para usar los módulos instalados por el profesor con `perl5.8.*` establezca las siguientes variables de entorno para poder acceder a las mismas:

  ```
  [casiano@tonga]~/src/perl/tests/gtk2/> cat ~/bin/setperl5lib
  # October 2008
  # source ~/bin/setperl5lib
  export PERL5LIB=/soft/perl5lib/share/perl/5.8.8/:/soft/perl5lib/lib/perl/5.8.8:/soft/perl
  export MANPATH=$MANPATH:/soft/perl5lib/man/
  export PATH=$PATH:/soft/perl5lib/bin
  ```

  En una `bash` ejecute:

  ```
  $ source ~/bin/setperl5lib
  ```

  para establecer las variables de entorno. También puede añadir estas líneas en su fichero `~/.profile`

- Perl 5.10

  ```
  casiano@millo:~$ cat bin/setperl5_10lib
  # October 2008
  # source ~/bin/setperl5_10lib
  export PERL5LIB=/soft/perl5lib/perl5_10_1/lib/:/soft/perl5lib/perl5_10_1/lib/perl5:/soft/
  export MANPATH=$MANPATH:/soft/perl5lib/perl5_10_1/man/
  # /soft/perl5lib/perl5_10_1/bin/ must be after ...
  export PATH=$PATH:/soft/perl5lib/perl5_10_1/bin/
  ```

No es buena idea sustituir el intérprete nativo de perl por otra versión. La distribución de Linux instalada depende de la versión del intérprete Perl. No cambie el `PATH` de manera que el intérprete `perl5.10.1` se ejecute antes que el nativo.

En general, para acceder a la documentación de un módulo use `perldoc` en vez de `man`.

Para ejecutar `perl5.10` escriba el nombre seguido de la versión:

```
casiano@millo:~$ perl5.10.1 -V
Summary of my perl5 (revision 5 version 10 subversion 1) configuration:

  Platform:
    osname=linux, osvers=2.6.24-21-generic, archname=i686-linux-thread-multi
    uname='linux millo 2.6.24-21-generic #1 smp tue oct 21 23:43:45 utc 2008 i686 gnulinux '
    config_args='-de -Dprefix=/soft/perl5lib/perl5_10_1/ -Dusethreads'
    hint=recommended, useposix=true, d_sigaction=define
    useithreads=define, usemultiplicity=define
    useperlio=define, d_sfio=undef, uselargefiles=define, usesocks=undef
    use64bitint=undef, use64bitall=undef, uselongdouble=undef
    usemymalloc=n, bincompat5005=undef
  Compiler:
    cc='cc', ccflags ='-D_REENTRANT -D_GNU_SOURCE -fno-strict-aliasing -pipe -fstack-protector
    optimize='-O2',
    cppflags='-D_REENTRANT -D_GNU_SOURCE -fno-strict-aliasing -pipe -fstack-protector -I/usr/l
    ccversion='', gccversion='4.2.4 (Ubuntu 4.2.4-1ubuntu4)', gccosandvers=''
    intsize=4, longsize=4, ptrsize=4, doublesize=8, byteorder=1234
    d_longlong=define, longlongsize=8, d_longdbl=define, longdblsize=12
    ivtype='long', ivsize=4, nvtype='double', nvsize=8, Off_t='off_t', lseeksize=8
    alignbytes=4, prototype=define
  Linker and Libraries:
    ld='cc', ldflags =' -fstack-protector -L/usr/local/lib'
    libpth=/usr/local/lib /lib /usr/lib
    libs=-lnsl -ldl -lm -lcrypt -lutil -lpthread -lc
    perllibs=-lnsl -ldl -lm -lcrypt -lutil -lpthread -lc
    libc=/lib/libc-2.7.so, so=so, useshrplib=false, libperl=libperl.a
    gnulibc_version='2.7'
  Dynamic Linking:
    dlsrc=dl_dlopen.xs, dlext=so, d_dlsymun=undef, ccdlflags='-Wl,-E'
    cccdlflags='-fPIC', lddlflags='-shared -O2 -L/usr/local/lib -fstack-protector'


Characteristics of this binary (from libperl):
  Compile-time options: MULTIPLICITY PERL_DONT_CREATE_GVSV
                        PERL_IMPLICIT_CONTEXT PERL_MALLOC_WRAP USE_ITHREADS
                        USE_LARGE_FILES USE_PERLIO USE_REENTRANT_API
  Built under linux
  Compiled at Aug 30 2009 09:34:30
  %ENV:
    PERL5LIB="/soft/perl5lib/share/perl/5.8.8/:/soft/perl5lib/lib/perl/5.8.8:/soft/perl5lib/li
  @INC:
    /soft/perl5lib/share/perl/5.8.8/
    /soft/perl5lib/lib/perl/5.8.8
    /soft/perl5lib/lib/perl/5.8
    /soft/perl5lib/share/perl/5.8/
    /soft/perl5lib/perl5_10_1/lib/5.10.1/i686-linux-thread-multi
    /soft/perl5lib/perl5_10_1/lib/5.10.1
    /soft/perl5lib/perl5_10_1/lib/site_perl/5.10.1/i686-linux-thread-multi
    /soft/perl5lib/perl5_10_1/lib/site_perl/5.10.1
```

Para acceder a la documentación relacionada con Perl 5.10 use el script `perl5.10.1.doc`:

```
casiano@millo:~$ perl5.10.1.doc -h
```

```
perldoc [options] PageName|ModuleName|ProgramName...
perldoc [options] -f BuiltinFunction
perldoc [options] -q FAQRegex

Options:
    -h   Display this help message
    -V   report version
    -r   Recursive search (slow)
    -i   Ignore case
    -t   Display pod using pod2text instead of pod2man and nroff
             (-t is the default on win32 unless -n is specified)
    -u   Display unformatted pod text
    -m   Display module's file in its entirety
    -n   Specify replacement for nroff
    -l   Display the module's file name
    -F   Arguments are file names, not modules
    -v   Verbosely describe what's going on
    -T   Send output to STDOUT without any pager
    -d output_filename_to_send_to
    -o output_format_name
    -M FormatterModuleNameToUse
    -w formatter_option:option_value
    -L translation_code   Choose doc translation (if any)
    -X   use index if present (looks for pod.idx at /soft/perl5lib/perl5_10_1/lib/5.10.1/i686-
    -q   Search the text of questions (not answers) in perlfaq[1-9]

PageName|ModuleName...
         is the name of a piece of documentation that you want to look at. You
         may either give a descriptive name of the page (as in the case of
         `perlfunc') the name of a module, either like `Term::Info' or like
         `Term/Info', or the name of a program, like `perldoc'.

BuiltinFunction
         is the name of a perl function.  Will extract documentation from
         `perlfunc'.

FAQRegex
         is a regex. Will search perlfaq[1-9] for and extract any
         questions that match.

Any switches in the PERLDOC environment variable will be used before the
command line arguments.  The optional pod index file contains a list of
filenames, one per line.
                                                      [Perldoc v3.14_04]
```

Visite esta página de vez en cuando. Es posible que añada algún nuevo camino de búsqueda de librerías y/o ejecutables.

Capítulo 8

Usando Subversion

En esta sección indicamos los pasos para utilizar subversión bajo el protocolo SSH. Se asume que ha configurado sub conexión SSH con el servidor de subversion para que admita autentificación automática. Véase El capítulo sobre la SSH en los apuntes de Programación en Paralelo II (http://nereida.deioc.ull.es/ pp2/perlexamples/chapter_ssh.html) para aprender a establecer autentificación automática vía SSH.

Use Subversion: Creación de un Repositorio Parece que en `banot` esta instalado subversion. Para crear un repositorio emita el comando `svnadmin create`:

```
-bash-3.1$ uname -a
Linux banot.etsii.ull.es 2.6.24.2 #3 SMP Fri Feb 15 10:39:28 WET 2008 i686 i686 i386 GNU/Linux
-bash-3.1$ svnadmin create /home/loginname/repository/
-bash-3.1$ ls -l repository/
total 28
drwxr-xr-x 2 loginname apache 4096 feb 28 11:58 conf
drwxr-xr-x 2 loginname apache 4096 feb 28 11:58 dav
drwxr-sr-x 5 loginname apache 4096 feb 28 12:09 db
-r--r--r-- 1 loginname apache    2 feb 28 11:58 format
drwxr-xr-x 2 loginname apache 4096 feb 28 11:58 hooks
drwxr-xr-x 2 loginname apache 4096 feb 28 11:58 locks
-rw-r--r-- 1 loginname apache  229 feb 28 11:58 README.txt
```

Una alternativa a considerar es ubicar el repositorio en un dispositivo de almacenamiento portable (pendriver)

Añadiendo Proyectos

Ahora esta en condiciones de añadir proyectos al repositorio creado usando `svn import`:

```
[loginname@tonga]~/src/perl/> uname -a
Linux tonga 2.6.24.2 #1 SMP Thu Feb 14 15:37:31 WET 2008 i686 i686 i386 GNU/Linux
[loginname@tonga]~/src/perl/> pwd
/home/loginname/src/perl
[loginname@tonga]~/src/perl/> ls -ld /home/loginname/src/perl/Grammar-0.02
drwxr-xr-x 5 loginname Profesor 4096 feb 28  2008 /home/loginname/src/perl/Grammar-0.02
[loginname@tonga]~/src/perl/> svn import -m 'Grammar Extended Module' \
                                   Grammar-0.02/ \
                                   svn+ssh://banot/home/loginname/repository/Grammar
Añadiendo      Grammar-0.02/t
Añadiendo      Grammar-0.02/t/Grammar.t
Añadiendo      Grammar-0.02/lib
Añadiendo      Grammar-0.02/lib/Grammar.pm
```

```
Añadiendo      Grammar-0.02/MANIFEST
Añadiendo      Grammar-0.02/META.yml
Añadiendo      Grammar-0.02/Makefile.PL
Añadiendo      Grammar-0.02/scripts
Añadiendo      Grammar-0.02/scripts/grammar.pl
Añadiendo      Grammar-0.02/scripts/Precedencia.yp
Añadiendo      Grammar-0.02/scripts/Calc.yp
Añadiendo      Grammar-0.02/scripts/aSb.yp
Añadiendo      Grammar-0.02/scripts/g1.yp
Añadiendo      Grammar-0.02/Changes
Añadiendo      Grammar-0.02/README

Commit de la revisión 2.
```

En general, los pasos para crear un nuevo proyecto son:

```
* mkdir /tmp/nombreProyecto
* mkdir /tmp/nombreProyecto/branches
* mkdir /tmp/nombreProyecto/tags
* mkdir /tmp/nombreProyecto/trunk
* svn mkdir file:///var/svn/nombreRepositorio/nombreProyecto -m 'Crear el proyecto nombreProye
* svn import /tmp/nombreProyecto \
            file:///var/svn/nombreRepositorio/nombreProyecto \
            -m "Primera versión del proyecto nombreProyecto"
```

Obtener una Copia de Trabajo

La copia en `Grammar-0.02` ha sido usada para la creación del proyecto, pero no pertenece aún al proyecto. Es necesario descargar la copia del proyecto que existe en el repositorio. Para ello usamos `svn checkout`:

```
[loginname@tonga]~/src/perl/> rm -fR Grammar-0.02
[loginname@tonga]~/src/perl/> svn checkout svn+ssh://banot/home/loginname/repository/Grammar G
A    Grammar/t
A    Grammar/t/Grammar.t
A    Grammar/MANIFEST
A    Grammar/META.yml
A    Grammar/lib
A    Grammar/lib/Grammar.pm
A    Grammar/Makefile.PL
A    Grammar/scripts
A    Grammar/scripts/grammar.pl
A    Grammar/scripts/Calc.yp
A    Grammar/scripts/Precedencia.yp
A    Grammar/scripts/aSb.yp
A    Grammar/scripts/g1.yp
A    Grammar/Changes
A    Grammar/README
Revisión obtenida: 2
```

Ahora disponemos de una copia de trabajo del proyecto en nuestra máquina local:

```
[loginname@tonga]~/src/perl/> tree Grammar
Grammar
|-- Changes
|-- MANIFEST
```

```
|-- META.yml
|-- Makefile.PL
|-- README
|-- lib
|   `-- Grammar.pm
|-- scripts
|   |-- Calc.yp
|   |-- Precedencia.yp
|   |-- aSb.yp
|   |-- g1.yp
|   `-- grammar.pl
`-- t
    `-- Grammar.t

3 directories, 12 files
[loginname@tonga]~/src/perl/>
[loginname@tonga]~/src/perl/> cd Grammar
[loginname@tonga]~/src/perl/Grammar/> ls -la
total 44
drwxr-xr-x 6 loginname Profesor 4096 feb 28  2008 .
drwxr-xr-x 5 loginname Profesor 4096 feb 28  2008 ..
-rw-r--r-- 1 loginname Profesor  150 feb 28  2008 Changes
drwxr-xr-x 3 loginname Profesor 4096 feb 28  2008 lib
-rw-r--r-- 1 loginname Profesor  614 feb 28  2008 Makefile.PL
-rw-r--r-- 1 loginname Profesor  229 feb 28  2008 MANIFEST
-rw-r--r-- 1 loginname Profesor  335 feb 28  2008 META.yml
-rw-r--r-- 1 loginname Profesor 1196 feb 28  2008 README
drwxr-xr-x 3 loginname Profesor 4096 feb 28  2008 scripts
drwxr-xr-x 6 loginname Profesor 4096 feb 28  2008 .svn
drwxr-xr-x 3 loginname Profesor 4096 feb 28  2008 t
```

Observe la presencia de los subdirectorios de control `.svn`.

Actualización del Proyecto

Ahora podemos modificar el proyecto y hacer públicos los cambios mediante `svn commit`:

```
loginname@tonga]~/src/perl/Grammar/> svn rm META.yml
D         META.yml
[loginname@tonga]~/src/perl/Grammar/> ls -la
total 40
drwxr-xr-x 6 loginname Profesor 4096 feb 28  2008 .
drwxr-xr-x 5 loginname Profesor 4096 feb 28 12:34 ..
-rw-r--r-- 1 loginname Profesor  150 feb 28 12:34 Changes
drwxr-xr-x 3 loginname Profesor 4096 feb 28 12:34 lib
-rw-r--r-- 1 loginname Profesor  614 feb 28 12:34 Makefile.PL
-rw-r--r-- 1 loginname Profesor  229 feb 28 12:34 MANIFEST
-rw-r--r-- 1 loginname Profesor 1196 feb 28 12:34 README
drwxr-xr-x 3 loginname Profesor 4096 feb 28 12:34 scripts
drwxr-xr-x 6 loginname Profesor 4096 feb 28  2008 .svn
drwxr-xr-x 3 loginname Profesor 4096 feb 28 12:34 t
[loginname@tonga]~/src/perl/Grammar/> echo "Modifico README" >> README
[loginname@tonga]~/src/perl/Grammar/> svn commit -m 'Just testing ...'
Eliminando     META.yml
Enviando       README
```

```
Transmitiendo contenido de archivos .
Commit de la revisión 3.
```

Observe que ya no es necesario especificar el lugar en el que se encuentra el repositorio: esa información esta guardada en los subdirectorios de administración de subversion `.svn`

El servicio de subversion parece funcionar desde fuera de la red del centro. Véase la conexión desde una maquina exterior:

```
pp2@nereida:/tmp$  svn checkout svn+ssh://loginname@banot.etsii.ull.es/home/loginname/reposito
loginname@banot.etsii.ull.es's password:
loginname@banot.etsii.ull.es's password:
A    Grammar/t
A    Grammar/t/Grammar.t
A    Grammar/MANIFEST
A    Grammar/lib
A    Grammar/lib/Grammar.pm
A    Grammar/Makefile.PL
A    Grammar/scripts
A    Grammar/scripts/grammar.pl
A    Grammar/scripts/Calc.yp
A    Grammar/scripts/Precedencia.yp
A    Grammar/scripts/aSb.yp
A    Grammar/scripts/g1.yp
A    Grammar/Changes
A    Grammar/README
Revisión obtenida: 3
```

Comandos Básicos

- Añadir y eliminar directorios o ficheros individuales al proyecto

  ```
  svn add directorio_o_fichero
  svn remove directorio_o_fichero
  ```

- Guardar los cambios

  ```
  svn commit -m "Nueva version"
  ```

- Actualizar el proyecto

  ```
  svn update
  ```

- Ver el estado de los ficheros

  ```
  svn status -vu
  ```

- Crear un tag

  ```
  svn copy svn+ssh://banot/home/logname/repository/PL-Tutu/trunk \
           svn+ssh://banot/home/casiano/repository/PL-Tutu/tags/practica-entregada \
   -m 'Distribución como fué entregada en la UDV'
  ```

Referencias

Consulte http://svnbook.red-bean.com/ . Vea la página de la ETSII http://www.etsii.ull.es/svn
.

En KDE puede instalar el cliente gráfico KDEsvn.

Autentificación Automática

Para evitar la solicitud de claves cada vez que se comunica con el repositorio establezca autentificación SSH automática. Para ver como hacerlo puede consultar las instrucciones en:

http://search.cpan.org/ casiano/GRID-Machine/lib/GRID/Machine.pod#INSTALLATION

Consulte también las páginas del manual Unix de `ssh`, `ssh-key-gen`, `ssh_config`, `scp`, `ssh-agent`, `ssh-add`, `sshd`

Índice alfabético

Bibliografía

[1] Jeffrey E.F. Friedl. *Mastering Regular Expressions.* O'Reilly, USA, 1997. ISBN 1-56592-257-3.

[2] Casiano Rodriguez Leon. *Perl: Fundamentos.* `http://nereida.deioc.ull.es/~lhp/perlexamples/`, 2001.

[3] Peter Scott. *Perl Medic: Maintaining Inherited Code.* Addison Wesley, USA, 2004. ISBN 0201795264.

[4] Ian Langworth and chromatic. *Perl Testing: A Developer's Notebook.* O'Reilly Media, Inc., 2005.

[5] Damian Conway. *Perl Best Practices.* O'Reilly Media, Inc., 2005.

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 1986.

[7] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[8] Henry F. Ledgard. Ten mini-languages in need of formal definition. *SIGPLAN Not.*, 5(4-5):14–37, 1970.