Análisis Léxico y Sintáctico con Perl

Casiano R. León [1]

21 de febrero de 2006

[1]DEIOC Universidad de La Laguna

Índice general

Índice de figuras

Índice de cuadros

Comenzando

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 Damian Conway, Tom Christiansen, Francois Desarmenien, Jeffrey E.F. Friedl, Joseph N. Hall, Randal L. Schwartz, Michael Schwern, Peter Scott, Sriram Srinivasan, Linconl Stein, Nathan Torkington and Larry Wall for their books and/or their modules. Special thanks to Larry Wall for giving us Perl.

Erratas

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

Como Obtener Estos Apuntes

Este no es un curso completo de Perl (Probablemente es imposible escribir un curso completo de Perl). Es mas bien un conjunto de notas sobre como usar esta espléndida herramienta. En este curso se hace especial énfasis en su manejo durante el proceso de traducir desde un lenguaje a otro, desde un formato a otro, desde una forma de representación del conocimiento a otra.

La versión *postcript* de este documento puede encontrarse en

`http://nereida.deioc.ull.es/~pl/perlexamples.ps`

La correspondiente versión HTML en la *web* se encuentra en:

`http://nereida.deioc.ull.es/~pl/perlexamples/`

Prólogo

Perl (Practical Extraction and Report Language) es un lenguaje interpretado que permite la manipulación de ficheros de texto y procesos. Perl propociona funciones muy potentes para la manipulación de textos, y en principio se consideró solo para esta tarea. Sin embargo, ha evolucionado tanto que hoy en día se le considera como una herramienta de programación de internet y de administración de sistemas y redes Unix.

Algunas característas que hacen a Perl popular son:

- Reduce el ciclo de programación. Puesto que es interpretado, no es necesario compilar las aplicaciones y se pueden ejecutar en plataformas diferentes.
- Es portable. Existen intérpretes Perl para una gran variedad de sistemas operativos.
- La sintaxis es muy similar a la de otros lenguajes como Sed, AWK o C. Existen herramientas para traducir código Sed y AWK a Perl de manera automática.
- No cuesta nada, es de libre distribución.
- Es fiable y robusto. Herramientas como CVS están implementadas con Perl.

En contra de Perl se esgrimen varias razones, entre ellas:

- Cualquiera puede ver el código fuente de su aplicación porque es interpretado y no compilado.
- Por ser interpretado y no compilado, su velocidad puede ser sustancialmente inferior a la versión C en algunos casos.

La literatura sobre Perl es extraordinaria. Un libro introductorio es "Learning Perl" [1]. Se puede seguir con el libro "Advanced Perl Programming" de S. Srinivasan [2]. La referencia definitiva sobre Perl es el libro de Wall [3]. Para saber en profundidad sobre expresiones regulares esta el libro de Friedl [4]. El libro de Stein [5] cubre temas de programación en redes. Miles de recetas clasificadas según los problemas pueden encontrarse en el libro de Christiansen [6].

Aunque potente, la dependencia del contexto de Perl le hace un lenguaje lleno de "multiples significados". Un libro que ayuda a sortear las ambiguedades es "Effective Perl Programming", de Hall [7].

Los contenidos del capítulo sobre el conjunto de módulos LWP están inspirados en la obra de Burke [8]. Algunos temas de persistencia y bases de datos deben muchísimo al libro de Alligator Descartes y Tim Bunce [9].

La interfase de Perl con otros lenguajes ha mejorado mucho gracias a la presencia del módulo `Inline` debido a Brian Ingerson [10].

Los capítulos de análisis léxico y sintáctico están inspirados en el clásico libro del dragón [11].

Una nota sobre el uso del inglés y el español en este documento: Resulta muy difícil evitar el uso de terminos en inglés en un documento como éste. He intentado utilizar un *font* distinto para esos casos. En los programas, muchos de los nombres de objetos y los comentarios aparecen en inglés. La experiencia me ha mostrado que, si tu programa es suficientemente bueno, acabará siendo estudiado y leído por programadores que no hablan castellano. Si quieres garantizar una verdadera "portabilidad" de tu código, utiliza el inglés como lenguaje de notación y documentación.

Los apuntes asumen que el lector esta familiarizado con el lenguaje C (no es necesario que sea un experto) y que ha recibido un curso en teoría de autómatas y lenguajes formales.

Parte I

Las Bases

Capítulo 1

Introducción

El fuente Perl lo puede encontrar en la página Web:

http://www.perl.com/pub/language/info/software.html.

Se encuentran disponibles los fuentes y los binarios para distintos sistemas.

Generalmente, lo que hay que hacer es descomprimir la distribución elegida y seguir las instrucciones del manual de instalación.

Comprueba que perl existe en tu ordenador:

```
$perl -v
This is perl, version 5.005_03 built for i386-linux

Copyright 1987-1999, Larry Wall
```

Comprueba en que directorio se guarda el ejecutable:

```
$ which perl
/usr/bin/perl
```

Edita y guarda el fichero hola.pl. Cambia los permisos de ejecución:

```
>vi hola.pl
> ... salimos y salvamos ...
>cat hola.pl
#!/usr/bin/perl -w
print "hola!\n";
>
>chmod a+x hola.pl
```

La primera línea, comenzando por `#!` indica al intérprete de comandos en que lugar se encuentra el interprete para este guión, esto es, donde se encuentra `perl`. La opción `-w` (la `w` viene de *warnings*) le indica al intérprete `perl` que debe advertirnos de cualquier potencial error que detecte. La conducta de Perl es, en general, permisiva con el usuario. Es conveniente hacer uso siempre de esta opción. Ahora ya puedes ejecutar tu primer programa Perl:

```
>./hola.pl
```

También podríamos haber escrito:

```
$ perl -w hola.pl
hola!
```

o bien haber usado el depurador para ejecutar el programa paso a paso. El depurador de Perl se activa usando la opción `-d` del intérprete:

```
1 $ perl -d hola.pl
2
3 Loading DB routines from perl5db.pl version 1.25
4 Editor support available.
5
6 Enter h or `h h' for help, or `man perldebug' for more help.
7
8 main::(hola.pl:2):      print "hola!\n";
9   DB<1> h
10 List/search source lines:               Control script execution:
11   l [ln|sub]  List source code            T           Stack trace
12   - or .      List previous/current line  s [expr]    Single step [in expr]
13   v [line]    View around line            n [expr]    Next, steps over subs
14   f filename  View source in file         <CR/Enter>  Repeat last n or s
15   /pattern/ ?patt?   Search forw/backw    r           Return from subroutine
16   M           Show module versions        c [ln|sub]  Continue until position
17 Debugger controls:                        L           List break/watch/actions
18   o [...]     Set debugger options        t [expr]    Toggle trace [trace expr]
19   <[<]|{[{]|>[>] [cmd] Do pre/post-prompt b [ln|event|sub] [cnd] Set breakpoint
20   ! [N|pat]   Redo a previous command     B ln|*      Delete a/all breakpoints
21   H [-num]    Display last num commands   a [ln] cmd  Do cmd before line
22   = [a val]   Define/list an alias        A ln|*      Delete a/all actions
23   h [db_cmd]  Get help on command         w expr      Add a watch expression
24   h h         Complete help page          W expr|*    Delete a/all watch exprs
25   |[|]db_cmd  Send output to pager        ![!] syscmd Run cmd in a subprocess
26   q or ^D     Quit                        R           Attempt a restart
27 Data Examination:     expr     Execute perl code, also see: s,n,t expr
28   x|m expr       Evals expr in list context, dumps the result or lists methods.
29   p expr         Print expression (uses script's current package).
30   S [[!]pat]     List subroutine names [not] matching pattern
31   V [Pk [Vars]]  List Variables in Package.  Vars can be ~pattern or !pattern.
32   X [Vars]       Same as "V current_package [Vars]".  i class inheritance tree.
33   y [n [Vars]]   List lexicals in higher scope <n>.  Vars same as V.
34 For more help, type h cmd_letter, or run man perldebug for all docs.
35   DB<1> l
36 2==>    print "hola!\n";
37   DB<1> n
38 hola!
39 Debugged program terminated.  Use q to quit or R to restart,
40   use O inhibit_exit to avoid stopping after program termination,
41   h q, h R or h O to get additional info.
42   DB<1> q
43 $
```

El comando `h` (línea 9) permite obtener ayuda. Con `l` (línea 35) podemos listar el programa. La orden `n` (línea 37) nos permite ejecutar paso a paso. A diferencia de la orden `s`, en el caso de que la instrucción sea una llamada a subrutina, no se entra en la subrutina. Por último, la orden `q` nos permite salir del depurador.

La opción `-e` del intérprete `perl` nos permite ejecutar un programa dado desde la línea de comandos, en vez de usar un fichero:

```
$ perl -e 'print 4**2,"\n"'
```

Obsérvese como se han utilizado comillas simples para proteger el programa Perl de su posible itnerpretación por el intérprete de comandos del sistema (de la *shell*).

1.1. Escalares

En español se puede distinguir entre singular y plural. En Perl el singular esta representado por las expresiones escalares y el plural por las expresiones de tipo lista. Una variable escalar puede almacenar un número, una cadena de caracteres o una referencia. El valor de una variable escalar puede accederse a través de su nombre precedido de un `$`. Las listas, por el contrario, se prefijan de `@`. Esto contrasta con el español, que usa el sufijo `s` para indicar pluralidad. Ejemplos de escalares son:

```
$days   = 5;
$unit = "meters";
$height = 1.50;
```

Es posible usar el depurador para ver la conducta de Perl con estas sentencias:

```
$ perl -de 0
Loading DB routines from perl5db.pl version 1.25
Editor support available.
Enter h or ‘h h’ for help, or ‘man perldebug’ for more help.
main::(-e:1):   0
  DB<1> $days   = 5;
  DB<2> x $days
0  5
```

1.1.1. Números

Los números se representan en punto flotante (doble precisión). Asi pues, en Perl no se distingue entre enteros y flotantes. Perl además, admite el uso de subrayados en las constantes numericas, para hacerlas mas legibles:

```
$b = 123_456_000_000_000_001;
```

Aunque en otros lenguajes de programación números y cadenas son cosas bien distintas, en Perl ambos objetos son escalares.

Perl permite trabajar con números en bases octal (comenzando por `0`), hexadecimal (`0x`) y binario (`0b`). Por ejemplo:

```
0377  # 377 octal = 256 decimal
0xff  # FF hex = 255 decimal
0b1111_1111 # 255 decimal
```

Además de los operadores habituales, Perl admite el operador de módulo o resto entero, denotado por `%` y el operador de exponenciación denotado por `**`.

Sigue un ejemplo que hace uso del depurador:

```
$ perl -de 0
main::(-e:1):   0
  DB<1> p 234_512.0*2
469024
  DB<2> p 012
10
  DB<3> p 0b111
```

```
7
  DB<4> p 5%2
1
  DB<5> p 5**2
25
  DB<6> p 0x1F
31
 DB<7> p 2/3
0.666666666666667
  DB<8> p (-1)**0.5
nan
  DB<9> p 1/0
Illegal division by zero at (eval 12)...
```

Ejercicio 1.1.1. *¿Que significa la respuesta* `nan` *que da el depurador al cálculo de la raíz cuadrada?*

1.1.2. Cadenas

Las cadenas son secuencias de caracteres y pueden contener cualqueir combinación de caracteres.

Perl proporciona múltiples formas de entrecomillar cadenas. Las mas comunes implican el uso de las *cadenas de comillas simples* y las *cadenas de comillas dobles*. Por ejemplo:

```
print 'El \'Pescailla\' y la "Faraona"';    # El 'Pescailla' y la "Faraona"
```

La función `length` devuelve la longitud en caracteres de la expresión que se le pasa como argumento. Si se omite la expresión usará la variable `$_`.

Comillas Simples

Mientras que en las comillas dobles ciertas secuencias como `\n` son interpretadas, en el caso de las cadenas de *comillas simples*, cualquier carácter entre las comillas que no sea la comilla simple o el carácter de escape \ (lo que incluye al retorno de carro si la cadena esta hecha de varias líneas) no recibe interpretación de ningún tipo. Para obtener una comilla simple dentro de la cadena, la precedemos de un \. Para obtener una \ hay que poner dos sucesivas \\. Por ejemplo:

```
> cat singlequote.pl
#!/usr/bin/perl -w
$a = 'hola,
chicos';
print "$a\n";
$a = 'Le llaman \'speedy\' por que es muy rápido';
print "$a\n";
$a = 'El último carácter en esta cadena es un escape \\';
print "$a\n";
```

Cuando se ejecuta da como resultado:

```
> singlequote.pl
hola,
chicos
Le llaman 'speedy' por que es muy rápido
El último carácter en esta cadena es un escape \
>
```

Ejercicio 1.1.2. *¿Que salida resulta de ejecutar el siguiente programa?*

```
$ cat comillassimples.pl
#!/usr/bin/perl -w
print 'hola\n';
print "\n";
print 'hola\\n';
print "\n";
print 'hola\\';
print "\n";
```

Comillas Dobles

En las *comillas dobles* el carácter de escape permite especificar caracteres de control.

Existe un buen número de secuencias de escape. He aqui algunas:

Código	Significado
`\a`	Alarma (beep!)
`\b`	Retroceso
`\n`	Nueva línea
`\r`	Retorno de carro
`\t`	Tabulador
`\f`	Formfeed
`\e`	Escape
`\007`	Cualquier carácter ASCII en octal
`\x7f`	Cualquier carácter ASCII en hexadecimal
`\cC`	Cualquier carácter de control. Aqui CTRL-C

Existen también secuencias de escape como `\u \U \l \L \E` que cambian de mayúsculas a minúsculas y viceversa:

```
$p = "pescailla"; $f = "faraona";
print "El \U$p\E  y la \u$f\n";    # El PESCAILLA y la Faraona
```

Los operadores `\u` (*uppercase*) y `\l` (*lowercase*) cambian el siguiente carácter a mayúscula y minúscula respectivamente. Los operadores `\U` y `\L` cambian toda la secuencia hasta la aparición del operador `\E`.

Es posible usar los operadores `q` y `qq` para entrecomillar cadenas. El operador `q` convierte su argumento en una cadena de comillas simples. El operador `qq` convierte su argumento en una cadena de comillas dobles.

Así:

```
print q*El 'morao' de \*estrellas\** # El 'morao' de *estrellas*
```

Es posible usar delimitadores "pareja" como `{}`, `[]`, `()`, etc. En tal caso, Perl lleva la cuenta del anidamiento de la pareja:

```
$a = 4;
print qq<No me <<cites>> mas de $a veces!>; #No me <<cites>> mas de 4 veces!
```

Las cadenas con comillas dobles permiten la *interpolación* de variables. Esto quiere decir que si en la cadena escribes el nombre de una variable escalar, se sustituirá por su valor.

```
$a = 1;
print "$a"; # imprime un 1
print '$a'; # imprime $a
```

El nombre de la variable es el identificador mas largo que sigue al `$`. Esto puede ser un problema si lo que quieres es imprimir el valor de la variable seguido de letras. La solución es envolver el identificador de la variable entre llaves.

```
$que = "chuleta";
$n = 3;
print "Antonio comió $n $ques.\n"; #la variable a sustituir es $ques
print "Antonio comió $n ${que}s.\n";
print "Antonio comió $n $que"."s.\n"; # otra forma de hacerlo
```

Operadores de Cadena

El punto concatena y el operador `x` permite repetir una cadena:

```
"hola"."chicos" # es lo mismo que "holachicos"
"HLP"x2 # es HLPHLP
```

El operador de asignación `.=` permite asignar a una variable cotneniendo una cadena el resultado de concatenarla con la cadena en el lado derecho.

```
$s .= " un poco mas";  # Lo mismo que $s = $s . " un poco mas";
```

1.1.3. Contexos Numérico y de Cadena: Conversiones

Perl convierte automáticamente los valores de su representación numérica a su representación como cadena. Por ejemplo, si una cadena aparece al lado de un operador numérico como '+', Perl convierte la cadena a un número, antes de proceder a la evaluación de la expresión aritmética. Los contextos en los que se espera una cadena se denominan *contextos de cadena*. Aquellos en los que se esperan números se denominan *contextos numéricos*.

La función utilizada internamente por Perl para convertir implícitamente números a cadenas es la función de la librería estandard C `sprintf`. Análogamente, la función para convertir cadenas a números es `atof`. Los espacios en blanco iniciales se ignoran. La conversión utiliza la parte inicial que parezca un número y el resto se ignora.

```
$n = 0 + "123"; # contexto numérico $n == 123
$n = 0 + "123xyz";  # contexto numérico $n == 123
$n = 0 + "\n123xyz"; # contexto numérico $n == 123
$n = 0 + "x123xyz"; # no hay número al comienzo $n == 0
```

El proceso de conversión no reconoce números octales no hexadecimales. Para ello deberá usarse explícitamente el operador `oct`:

```
$n = 0 + "x123"; # no hay número al comienzo $n == 0
$n = 0 + oct("x123"); # $n == 291
```

La llamada `oct expresión` devuelve el valor decimal de `expresion`, interpretado como una cadena octal. Si la `expresion` comienza con `0x`, la interpreta como una cadena hexadecimal.

Los *operadores de bit* son *sensibles al contexto*. Si es un *contexto de cadena*, utilizan como unidad el carácter, mientras que si es un *contexto numérico*, la unidad es el bit. Veamos un ejemplo. Para ello usaremos el depurador de Perl que se activa usando la opción `-d`:

```
~/perl/src> perl -de 0
```

Es necesario facilitarle al depurador un programa. Para evitar la molestia de escribir uno, le indicamos que le pasamos el programa en la línea de comandos. Esto se hace con la opción `-e`, que debe ir seguida del programa Perl. En el ejemplo que sigue hemos elegido pasarle como programa uno bastante trivial: una expresión que es la constante cero:

```
~/perl/src> perl -de 0
Loading DB routines from perl5db.pl version 1.19
Editor support available.
Enter h or ‘h h’ for help, or ‘man perldebug’ for more help.
```

```
main::(-e:1):   0
DB<1> $a = 123
DB<2> $b = 234
DB<3> $c = $a & $b
DB<4> p $c
106
DB<5> $a = "$a"
DB<6> $b = "$b"
DB<7> $d = $a & $b
DB<8> p $d
020
```

1.1.4. Variables Mágicas

Otro ejemplo de sensibilidad a este tipo de contexto lo da la variable escalar especial `$!`. En Perl existen *variables "mágicas"* o especiales que tienen este tipo de extraños nombres. Son mágicas porque, en la mayoría de los casos, se actualizan como consecuencia de algún efecto lateral. En un contexto numérico, la variable escalar `$!` retorna el valor numérico de la variable del sistema `errno`, la cual contiene el último error encontrado durante una llamada al sistema o a ciertas funciones de la biblioteca estándar C. En un contexto de cadena devuelve la cadena producida por la función `perror()`:

```
open FILE, "fichero";
print "$!\n" if $!; # "No such file or directory"
print 0+$!,"\n";    # "2" u otro número
```

La función `open` intenta abrir el fichero con nombre `fichero` para dejar una descripción del mismo en la variable `FILE`. En el supuesto de que tal fichero no exista, se producirán los correspondientes mensajes de error almacenados en `$!`. Observe el uso del condicional `if $!`. Si la cadena `$!` existe y está definida, la condición lógica será evaluada como cierta. En caso contrario, el `print` no se ejecutará. Para mas información sobre el uso de las expresiones lógicas en Perl lea la sección 1.2.

Probablemente la mas importante de estas variables mágicas es `$_`. La variable especial `$_` es el argumento por defecto para un gran número de funciones, operadores y estructuras de control. Asi, por ejemplo:

```
print;
```

es lo mismo que

```
print $_;
```

1.1.5. Lectura de Datos

El operador `<STDIN>` nos permite leer desde la entrada estándar. Cuando se usa en un contexto escalar, perl lee de la entrada estándar hasta (e incluyendo) el siguiente retorno de carro.

Asi en:

```
$line = <STDIN>;
```

La variable `$line` contiene el retorno de carro leído. Es por eso que se usa el operador `chomp` el cual elimina el retorno de carro final:

```
$line = <STDIN>; # "hola\n"
chomp($line);    # $line contiene ahora "hola"
```

Si la línea termina en varios retornos de carro, `chomp` sólo elimina uno, si no hay no hace nada.

1.2. La Lógica de Perl

Las reglas básicas para la evaluación de una expresión lógica son:

1. Cualquier cadena es `true` con la excepción de las cadenas `""` y `"0"`.
2. Cualquier número es `true` excepto el `0`.
3. Cualquier referencia es `true`.
4. Cualquier valor `undefined` es `false`.

El operador `defined` provee el medio para distinguir `undef` de `0` y de la cadena vacía `''`. El operador `defined` trabaja con cualquier valor. En versiones previas de Perl se requería que el argumento fuera un `lvalue`, una expresión que se pueda interpretar como la parte izquierda de una asignación.

```
if (defined(0)) { ... } # TRUE, error en Perl 4

if (defined(()) { ... } # TRUE, error en Perl 4
```

1.2.1. Operadores Lógicos

Perl tiene dos tipos de operadores lógicos: uno que tiene "estilo C" y el otro de "estilo Pascal":

```
  DB<1> $a = 4; $b = "hola"; $c = 0; $d = "";
  DB<2> p $a && $b
hola
  DB<3> p $a and $b
4
  DB<4> p ($a and $b)
hola
  DB<5> p $c
0
  DB<6> p !$c
1
  DB<7> p not $c
1
  DB<8> p not $d
1
  DB<9> p !$d
1
  DB<10> p $a || $b
4
  DB<11> p $a or $b
4
```

Toda la evaluación se hace en circuito corto. Así `$a and $b` es `$a` si `$a` es falso, si no da como resultado `$b`. Lo mismo es cierto para `$a && $b`.

Ejercicio 1.2.1. *¿Porqué* `p $a and $b` *produce como salida 4? ¿Es ese realmente el resultado de la evaluación de la expresión? Antes de contestar estudie este experimento:*

```
  DB<1> $a = 4; $b = "hola";
  DB<2> $d = (print $a and $b)
4
  DB<3> p $d
hola
```

```
 DB<4> $e = print $a and $b
4
  DB<5> p $e
1
  DB<6> $d = (print $a && $b)
hola
```

¿Porqué `$d` *contiene* `"hola"` *y sin embargo se imprime un 4 en la línea 2? ¿como interpreta Perl la expresión* `print $a and $b`*? ¿Como* `(print $a) and $b`*? o bien como ¿* `print ($a and $b)`*? ¿Que tiene mas prioridad, el operador* `and` *o la llamada a* `print`*? ¿Ocurre lo mismo con el operador* `&&`*?*

1.2.2. Operadores de Comparación

Perl tiene dos conjuntos de operadores de comparación: uno para los números y otros para las cadenas. Los operadores de cadenas se componen de letras (estilo FORTRAN) mientras que los numéricos siguen el estilo C.

Comparación	Numerico	Cadena
Igual	==	eq
distinto	!=	ne
Menor que	<	lt
mayor que	>	gt
Menor o igual que	<=	le
Mayor o igual que	>=	ge
Comparación	<=>	cmp

Cuadro 1.1: Operadores de Comparación

La comparación entre cadenas sigue el orden lexicográfico.

```
'a' lt 'z' # TRUE
0 < 5 # TRUE
"H" cmp "He" # -1 operador de comparacion (0 si =, 1 si $a>$b, -1 si $a<$b)
10 <=> 8.5   #  1 el operador "guerra de las galaxias"
             #  <=> Darth Vader's fighter!
```

Los operadores de comparación de cadenas no deberían usarse para números, salvo que sea eso precisamente lo que se quiere (esto es, 10 va antes de 2). Analogamente, los operadores de comparación de números no deberían usarse para cadenas:

```
"a" == "b" # TRUE, ambas cadenas como números son 0
```

1.3. Algunas Sentencias de Control

Los condicionales y los bucles son como en C. La única diferencia es que requieren que la sentencia bajo control sea un bloque, esto es, este escrita entre llaves.

1.3.1. La estructura de control `unless`

Si se quiere ejecutar un bloque de código cuando el condicional es falso se puede usar *unless*:

```
unless ($a < $b) {
  print "$a >= $b\n";
}
```

que es equivalente a:

```
if ($a < $b)  { }
else {
  print "$a >= $b\n";
}
```

Por supuesto, siempre cabe la posibilidad de negar la condición:

```
if (!($a < $b)) {
  print "$a >= $b\n";
}
```

1.3.2. La estructura de control `until`

Existe asimismo un bucle `until`:

```
until ($j > $i) { $j *= 2; }
```

que se ejecutará hasta que la condición pasa a ser cierta.

1.3.3. Modificadores de Expresiones

Una expresión puede ir seguida de una sentencia de control que determina su evaluación. Por ejemplo:

```
print "$n < 0 \n" if $n < 0;
```

Existen otros modificadores:

```
$i = 1 unless ($i < 0);
$i *= 2 until $i > $j;
print " ", ($n += 2) while $n < 10;
```

1.3.4. La estructura de control `last`

Como el `break` del lenguaje C, el operador `last` nos permite salir de un bucle.

1.4. Depuración de errores

Siga los siguientes consejos en todos sus programas:

- Para detectar errores sintácticos puede usar la opción `-c`, la cual le indica a Perl que debe compilar pero no ejecutar el programa.
- Use en todos los programas la opción `-w`
- Puede producir diagnósticos mas detallados poniendo:

  ```
  use diagnostics;
  ```

  al comienzo de su programa. Esto ya activa el modo `-w`.
- Añada:

  ```
  use strict;
  ```

  al comienzo de su programa.
- Si tiene errores en tiempo de ejecución utilice el depurador, para ello use la opción `-d` de Perl.

1.5. Una Breve Introducción a las Expresiones Regulares

El siguiente programa combina algunas de las características introducidas en las secciones anteriores, además de hacer uso de las expresiones regulares.

```
1 while (<STDIN>) {
2  if (/__END__/) { last; }
3  elsif (/LHP/) { print; }
4 }
```

En la línea 1 almacenamos en la variable mágica `$_` una línea que es leída desde la entrada estándar. El bucle se ejecuta mientras la condición sea cierta: mientras haya líneas en el fichero. Cuando se alcanza el final del fichero, se devuelve `undef` y por tanto se termina el bucle.

Antes de seguir, responda a las siguientes preguntas:

Ejercicio 1.5.1.

- *Si una línea del fichero de entrada contiene* `0`*: ¿Se terminará el bucle?*
- *¿Y si es una línea vacía?*

Ayuda: recuerde que en Perl el retorno de carro que termina una línea leida queda almacenado en la variable ...

Una expresión entre barras como `/__END__/` (línea 2) o `/LHP/` (línea 3) es una expresión regular en Perl.

Las Expresiones Regulares se pueden encontrar en muchos editores como vi, emacs, en programas como grep, egrep y en lenguajes de programación como awk y sed. Las Expresiones Regulares se usan para busquedas avanzadas dependientes del contexto. Las Expresiones Regulares describen de manera unívoca un lenguaje. La figura 1.2 muestra algunos de los metasímbolos usados en las expresiones regulares. Por ejemplo, la expresión regular `/[a-z]+/` define el lenguaje formado por las cadenas que consisten en repeticiones de una o mas letras minúsculas.

`*`	El elemento precedente debe aparecer 0 o más veces.
`+`	El elemento precedente debe aparecer 1 o más veces.
`.`	Un carácter cualquiera excepto salto de linea.
`?`	Operador unario. El elemento precedente es opcional
`{n}`	que coincida exactamente n veces
`{n,}`	que coincida al menos n veces
`{n,m}`	que coincida al menos n veces y no mas de m
`|`	O uno u otro.
`^`	Comienzo de linea
`$`	Fin de linea
`.`	Cualquier carácter que no sea el retorno de carro
`[...]`	Conjunto de caracteres admitidos.
`[^...]`	Conjunto de caracteres no admitidos.
`-`	Operador de rango
`(...)`	Agrupación.
`\`	Escape
`\n`	Representación del carácter fin de linea.
`\t`	Representación del carácter de tabulación.

Cuadro 1.2: Algunos metasímbolos usados en las expresiones regulares

Para saber más sobre expresiones regulares, estudie el capítulo 4. Por ahora nos basta saber que una expresión regular define un lenguaje. Asi la expresión regular `/__END__/` define el lenguaje cuya única palabra es `__END__` y la expresión regular `/LHP/` define el lenguaje cuya única palabra es `LHP`.

Normalmente, con las expresiones regulares se usa El operador de *binding* `=~`, el cuál nos permite "asociar" la variable con la operación de casamiento:

```
if ($d =~ /esto/) { print "la palabra 'esto' aparece en: $d\n" }
```

Si se omiten el operador `=~` y la variable, como se hace en la línea 2 entonces se usa la variable por defecto `$_`.

A menudo resulta práctico extraer las palabras que contienen una cifra, una vocal, o caracteres de control particulares. El modelo así definido no se indica por un carácter particular sino por un clase de caracteres mediante el operador `[ ]`. He aquí algunas posibles construcciones:

```
[aeiou]        # Cualquier vocal
[0-9]          # Cualquier número del 0 al 9.
[0123456789]   # Igual [0-9]
[0-9a-z]       # Cualquier letra o cualquier numéro
[\~\@;:\^_]    # Cualquiera de los caracteres(~,@,;,:^,_)
```

Generalmente de puede definir una clase de caracteres valiéndose de la complementaria. Ésta se especifíca mediante la sintaxis `[^ ]`. Siendo el símbolo `^` el que representa la negación de los caracteres o clase de caracteres consecutivos:

```
[^0-9]         # Carácter que no sea un dígito
```

Perl introduce algunas abreviaturas usadas para algunas de las clases mas comunes:

Código	Significado
`\d`	`[0-9]` dígitos del 0 al 9
`\D`	`[^0-9]` carácter que no sea un dígito
`\w`	`[a-zA-Z0-9_]` carácter alfanumérico
`\W`	`[^a-zA-Z0-9_]` carácter no alfanumérico
`\s`	`[ \t\n\r\f]` espacio en blanco
`\S`	`[^ \t\n\r\f]` carácter que no es un espacio en blanco

Cuadro 1.3: Abreviaturas para clases expreg

La condición `if (/__END__/)` ... en la línea 2 es cierta si la variable por defecto `$_` "casa" con la expresión regular `/__END__/` o, lo que es lo mismo, pertenece al lenguaje descrito por la expresión regular `/__END__/`. *Casar* en Perl significa que la cadena en cuestión, en este caso `$_`, contiene en algúna posición una subcadena que pertenece al lenguaje descrito por la expresión regular. Si se quisiera que solo hubiera *casamiento* cuando `$_` sea exactamente `__END__` deberíamos usar *anclas*. Un ancla es un metasímbolo que casa con una posición. Por ejemplo, el circunflejo `^` es un metasímbolo que casa con el comienzo de la cadena y el dolar `$` casa con el final. Así pues, si la expresión regular fuera `/^__END__$/` estaríamos forzando a que casar sea equivalente a que la cadena sea exactamente igual a `__END__`.

Obsérvese como también la entrada `<STDIN>` desde `STDIN` no es asignada explícitamente a una variable y por tanto la línea leída es guardada en la variable por defecto `$_`. El constructo *elsif* nos permite abreviar un `else` seguido de un `if`.

Este es un ejemplo típico de programa Perl "moderadamente críptico": no aparece explícitamente ninguna variable en el texto del programa.

1.6. Práctica: Radio de una circunferencia

Escriba un programa que solicite de la entrada un radio de una circunferencia e imprima el área de la misma. El proceso debe repetirse hasta que el radio es cero. Compruebe usando expresiones regulares que la entrada efectivamente contiene un número. Compruebe además que el radio no es negativo.

1.7. Arrays y Listas

Una variable `array` puede almacenar una secuencia de valores escalares, los cuales se identifican unívocamente a través de un índice. Los contenidos de un `array` pueden ser colectivamente accedidos usando el nombre del `array` precedido del símbolo `@`.

```
@days  = (1,2,3,4,5,6,7);
@days[3,4,5]        # same as @days[3..5]
```

El operador `..` toma dos numeros x_1 y x_2 y devuelve una lista con los enteros entre esos dos números:

```
  DB<1> @b = 4..8
  DB<2> p @b
45678
  DB<3> x @b
0  4
1  5
2  6
3  7
4  8
  DB<4> @a = 2.1 .. 4.1
  DB<5> x @a
0  2
1  3
2  4
```

Este operador devuelve una lista vacía cuando $x_2 < x_1$.

```
  DB<6> @c = 5..4
  DB<7> p @c

  DB<8> x @c
  empty array
```

Los valores individuales de un `array` se acceden precediendo de un `$` el nombre del `array`:

```
foreach $i (0..5) {
  print $days[$i]," es un día laboral/n";
}
```

El índice del primer elemento es cero.

Si se omite la variable que se usa para indexar en el bucle, se usará la *variable mágica por defecto* `$_`. Así el bucle anterior es equivalente a:

```
foreach  (0..5) {
  print $days[$_]," es un día laboral/n";
}
```

Una asignación a un elemento que no existe lo crea:

```
$days[9] = 10;
```

Los elementos extra `$days[6]`, `$days[7]` and `$days[8]` son asi mismo creados y se inicializan al valor indefinido `undef`.

La variable `$#days` almacena el índice del último elemento del `array`. En Perl, los *índices negativos* cuentan desde el final del `array` (-1 es el último elemento).

```
DB<1> @a = 1..10
DB<2> $a[-1] = 1
DB<3> p "@a"
1 2 3 4 5 6 7 8 9 1
DB<4> $a[-20] = 0
Modification of non-creatable array value attempted,
subscript -20 at (eval 18)
[/usr/share/perl/5.6.1/perl5db.pl:1521] line 2.
```

Si el array completo es accedido en un contexto en el que se espera un escalar numérico, (*scalar context*) el resultado es un escalar conteniendo el número de elementos del `array`.

```
$i = 0;
while ($i < @days) {
  print $days[$i++],"\n";
}
```

Los `arrays` estan muy relacionados, pero no son lo mismo que las listas. Una lista en Perl es una secuencia de valores separados por comas, normalmente entre parentesis. Un `array` es un contenedor para una lista. Las listas pueden usarse para extraer valores de los `arrays`. Por ejemplo:

```
($d1, $d2) = @days;
```

Si hay mas variables en la lista que elementos en el `array`, las variables extra quedan indefinidas (`undef`). Si por el contrario hay menos variables que elementos en el `array`, los elementos extra son ignorados.

Se puede asignar listas de valores a listas de variables:

```
($a, $b) = ($b, $a);
```

que intercambia los valores de `$a` y `$b`.

Una abreviación muy cómoda para crear listas de cadenas la proporciona el operador `qw`. Una cadena formada por palabras separadas por espacios en blanco dentro de `qw` se descompone en la lista de sus palabras.

```
@a = qw(uno dos tres); # @a == ("uno","dos", "tres")
```

Observa que no se ponen comas entre las palabras. Si por error escribieras:

```
@a = qw(uno, dos, tres); # @a == ("uno,","dos,", "tres")
```

obtendrías comas extra en los elementos de la lista.

Otro ejemplo:

```
@week = qw(monday tuesday wednesday thursday friday saturday sunday);
```

Los escalares no inicializados tienen el valor `undef`. Sin embargo, las variables `array` no inicializadas tienen como valor la lista vacía `()`. Si se asigna `undef` a una variable de tipo `array` lo que se obtiene es una lista cuyo único elemento es `undef`:

```
@a = undef; # Equivalente a @a = ( undef );
if (@a) { ... } # Por tanto TRUE
```

El modo mas simple de evitar este tipo de problemas cuando se quiere limpiar una variable de tipo `array` es asignar explícitamente la lista vacía `()`:

```
@a = ();
if (@a) { ...}; # scalar(@a) == FALSE
```

También se puede usar el operador `undef`:

```
undef @a;
if (defined(@a)) { ... }; # FALSE
```

Es posible asignar `undef` a los elementos de un `array`:

```
$a[3] = undef;
```

O bien usando la lista vacía, como se hace en la siguiente sesión con el depurador:

```
  DB<1>  @a = 0..9
  DB<2> @a[1,5,7] = ()
  DB<3> p "@a"
0  2 3 4  6  8 9
  DB<4> p $#a
9
  DB<5> p scalar(@a)
10
  DB<6> $" = ","
  DB<7> p "@a"
0,,2,3,4,,6,,8,9
  DB<8> @a[1..7] = ()
  DB<9> p "@a"
0,,,,,,,,8,9
```

1.7.1. Iterpolación de `arrays` en cadenas

Al igual que los escalares, los valores de un `array` son interpolados en las cadenas de comillas dobles. Los elementos son automáticamente separados mediante el *separador de elementos de un array* que es el valor guardado en la variable especial `$"`. Esta variable contiene un espacio por defecto.

```
  DB<1> @a = 0..9
  DB<2> print "@a 10\n"
0 1 2 3 4 5 6 7 8 9 10
  DB<3> $email = "casiano@ull.es"
  DB<4> print $email
casiano.es
  DB<5> $email = 'casiano@ull.es'
  DB<6> print $email
casiano@ull.es
  DB<7> $email = "casiano\@ull.es"
  DB<8> print $email
casiano@ull.es
```

Un único elemento de un `array` es reemplazado por su valor. La expresión que aparece como índice es evaluada como una expresión normal, como si estuviera fuera de una cadena:

```
  DB<1> @a = 0..9
  DB<2> $i = 2
  DB<3> $x = "$a[1]"
  DB<4> p $x
1
  DB<5> $x = "$a[$i-1]"
  DB<6> p $x
1
```

```
  DB<7> $i = "2*4"
  DB<8> $x = "$a[$i-1]"
  DB<9> p $x
1
```

Ejercicio 1.7.1. *¿Cuál es la explicación para la salida de* `p $x` *en el último ejemplo?*

Ejercicio 1.7.2. *Dado la siguiente sesión interactiva:*

```
nereida:~/perl/src> perl -de 0
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

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

main::(-e:1):   0
DB<1> @a = 1..5
DB<2> ($a[0], $a[1]) = undef
DB<3>  p "@a"
```

¿Cuál es la salida de la impresión que aparece al final de la secuencia de comandos?.

1.7.2. Bucles sobre `arrays`

Una diferencia del bucle `for` de Perl con el de C es que el índice del bucle constituye un "alias" del correspondiente elemento del `array`, de modo que su modificación conlleva la modificación del elemento del `array`. Consideremos el siguiente código:

```
#!/usr/bin/perl -w

my @list = 1..10;
foreach $n (@list) {
 $n *= 2;
}
print "@list\n";
```

su ejecución produce la siguiente salida:

```
~/perl/src> foreach.pl
2 4 6 8 10 12 14 16 18 20
```

1.7.3. Las declaraciones `my` y `our`

La declaración `my @list` en el ejemplo de la sección 1.7.2 indica que `@list` es una *variable local*. Las variables declaradas con `my` tienen por ámbito el texto desde la línea en que son declaradas hasta el final del bloque.

Perl 5.6.0 introdujo la declaración `our`. Esto nos permite declarar variables de paquete. La sintáxis es la misma que `my()`. Se trata de una declaración similar a `use vars`. Esto es, las variables no son controladas para su especificación completa por el módulo `strict`. Su ámbito (a diferencia de `use vars` es léxico. Esto es, las variables solo estan libres de comprobación estricta hasta el final del bloque actual.

```
use strict 'vars';
{
  our($x);
```

```
  $x = "hola"; # Uso correcto de $x
}
$x = "mundo";  # Esto producirá un error.
```

Asi pues, mientras que `use vars '$x'` declara que es correcto utilizar `$x` en cualquier parte, `our($x)` nos permite especificar en que partes de nuestro programa se puede escribir `$x` y en que partes es objeto de error.

1.7.4. El operador `grep`

Utilice el operador `grep` para seleccionar una sublista de una lista dada. El operador tiene dos formas de uso. El siguiente ejemplo, muestra la primera forma de uso, crea la lista `@withnumbers` a partir de una lista `@lines` formada por aquellos elementos que contienen uno o más dígitos.

```
@withnumbers =  grep /\d+/, @lines;
```

`grep` recibe como primer argumento una expresión regular y como segundo una lista. En Perl las expresiones regulares se escriben entre barras de división. La notación `\d` casa con cualquier dígito decimal. Así pues el cierre positivo casará con cualquier número entero sin signo. Para saber más sobre expresiones regulares, estudie el capítulo 4.

El siguiente código guarda en `$n_undef` el número de items `undef` en el `array v`

```
$n_undef = grep { !defined($_)} @v;
```

Obsérvese que en esta segunda forma de uso no hay coma de separación entre el bloque y el array. En general, el primer argumento de `grep` es un bloque en el cual se utiliza `$_` como referente a cada elemento de la lista, y se devuelve un valor lógico. Los restantes argumentos constituyen la lista de items sobre la que hay que buscar. El operador `grep` evalúa la expresión una vez para cada item en la lista, como si se tratara de un bucle `foreach`. Cuando el bloque esta constituido por una única expresión se puede escribir simplemente la expresión, separándola mediante una coma de la lista.

Ejercicio 1.7.3. *Explique la interpretación que hace Perl de los siguientes fragmentos de código. Consulte 1.7.4 y el manual de Perl para entender el uso de la función* `grep`*:*

```
@b = grep {not $_ % 5} @s;
```

1.7.5. El operador `map`

Si lo que se quiere es construir un `array` transformado del `array` inicial, se debe usar `map`:

```
@sizes = map {-s $_ } @files;
```

Este código genera un `array` conteniendo los tamaños de los ficheros especificados en `@files`. La forma de uso anterior utiliza la sintáxis `map bloque array`.

1.7.6. Allanamiento de las listas

En contra de lo que cabría esperar, una lista que contiene a otra lista:

```
@virtues = ("faith", "hope", ("love", "charity"));
```

no produce una jerarquía de listas, sino que la lista es aplanada y es lo mismo que si nunca se hubieran puesto los paréntesis. Esto es, es equivalente a:

```
@virtues = ("faith", "hope", "love", "charity");
```

1.7.7. Pilas y Colas: `push`, `pop` y `splice`

Perl proporciona las funciones *push*, *pop*, *shift* y *unshift* que permiten trabajar el `array` o lista como si de una pila o cola se tratase.

La función `push` tiene la sintáxis:

```
push(ARRAY,LIST)
```

empuja el valor de `LIST` en el `ARRAY`. La longitud del `ARRAY` se incrementa en la longitud de `LIST`. Es lo mismo que hacer:

```
for $value (LIST) {
        $ARRAY[++$#ARRAY] = $value;
}
```

La función `pop` tiene la sintáxis:

```
pop(ARRAY)
pop ARRAY
```

`pop ARRAY` tiene el mismo efecto que:

```
$tmp = $ARRAY[$#ARRAY]; $#ARRAY--
```

Si no hay elementos en `ARRAY`, devuelve el valor `undef`.

Las funciones `shift` y `unshift` actuán de manera similar a `push` y `pop` pero utilizando el comienzo de la lista en vez del final de la misma.

Las funciones `push`, `pop`, `shift` y `unshift` son un caso particular de la función `splice`, la cual cambia los elementos de un `ARRAY`. La función `splice` toma 4 argumentos: el `ARRAY` a modificar, el índice `OFFSET` en el cual es modificado, el número de elementos a suprimir `LENGTH` y la lista de elementos extra a insertar.

```
splice(ARRAY,OFFSET,LENGTH,LIST)
splice(ARRAY,OFFSET,LENGTH)
splice(ARRAY,OFFSET)
```

La función `splice` devuelve los elementos suprimidos del `ARRAY`. Si se omite `LENGTH` se suprime todo desde `OFFSET` hacia adelante. Se cumplen las siguientes equivalencias

```
push(@a,$x)          splice(@a,$#a+1,0,$x)
pop(@a)              splice(@a,-1)
shift(@a)            splice(@a,0,1)
unshift(@a,$x)       splice(@a,0,0,$x)
$a[$x] = $y          splice(@a,$x,1,$y);
```

No se puede *acortar un* `array` asignando `undef` a sus elementos del final. Para acortarlo se debe asignar `$#a` o utilizar un operador como `pop` o `splice`.

```
@a = 1 .. 10;
$a[9] = undef;  # @a = (1 ..9, undef)
$x = pop @a;    # @a = (1 ..9)
splice @a, -2;  # @a = (1 ..7) OFFSET = -2. Como se ha suprimido
                # LENGTH, se suprime desde el penúltimo elemento
                #  hacia adelante
$#a = 4;        # @a = (1 ..5)
```

1.7.8. `reverse`

El operador `reverse` toma los elementos de una lista y devuelve la lista en orden inverso:

```
  DB<1> @a = 1..10
  DB<2> @a = reverse @a
  DB<3> p "@a"
10 9 8 7 6 5 4 3 2 1
```

en un contexto escalar, `reverse` devuelve la cadena resultante de concatenar los elementos de la lista en orden inverso:

```
  DB<6> @a = 1..10
  DB<7> $b = reverse @a
  DB<8> p $b
01987654321
```

1.7.9. Ejercicio: Contextos

- ¿En que contexto (escalar o lista) se evalúa `algo` en las siguientes expresiones?

```
$f = algo;
@p = algo;
($w, $b) = algo;
($d) = algo;
$f[3] = algo;
123 + algo;
push @f, algo;
algo + 456;
foreach $f (algo) { ... }
if (algo) { ... }
sort algo
while (algo) { ... }
reverse algo;
$f[algo] = algo;
```

- ¿Que imprime el siguiente código?

```
@w = undef;
$a = @w;
print "$a\n";
```

1.7.10. La función `scalar`

La función `scalar` fuerza un contexto escalar:

```
  DB<1> @a = 0..9
  DB<2> p "@a\n"
0 1 2 3 4 5 6 7 8 9

  DB<3> p "\@a = ",@a,"\n"
@a = 0123456789

  DB<4> p "\@a = ",scalar @a,"\n"
@a = 10
```

1.7.11. Troceado de arrays

Un `slice` de un `array` es un trozo del mismo:

```
@days              # ($days[0], $days[1],... $days[n])
@days[3,4,5]       # @days[3..5]
@tragedy[3,4,5] = ("Mcburger", "King Leer", "Hamlet, A Pig in the City")
@sqrt[1,49,9,16,4] = (1, 7, 3, 4, 2) # índices desordenados
@sqr[1..4] = (1, 4, 9, 16)
@inverse[@sqr] = (1, 0.25, 0.111, 0.0625); # indexado en un array
@list[5..9] = reverse @list[5..9]; # invierte los elementos del 5 al 9
                                   # en list
@a[$n, $m] = @a[$m, $n]; # intercambiamos $a[m] y $a[n]
```

1.7.12. Ejercicio: Elemento o Trozo

¿Que es `@a[1]`? ¿Un elemento de un `array` o un trozo (*slice*) de un `array`?

1.7.13. Ejercicio: Asignaciones, Trozos y Contextos

¿Que efecto tiene el siguiente código?:

```
@text = (1, 2, 3, 4, 5);
@text[@text] = 0;
```

¿En que contexto se interpreta el `array @text` que aparece dentro del corchete? ¿Escalar o lista?

Respuesta

El `array text` entre corchetes es interpretado en un contexto de lista, pues no dice `$text[@text]`, dice `@text[@text]`. El resultado es un trozo con 5 elementos. Por tanto dice: `@text[(1, 2, 3, 4, 5)] = 0`. Asi pues, `text[0]` permanece no modificado y es 1, `text[1]` es igualado a 0 y los restantes miembros quedan `undef` ya que la parte derecha es una lista que contiene un sólo elemento.

1.7.14. join

La función `join` convierte un arreglo en un escalar

```
@a = ('a'..'e');
$a = join ":", @a  # $a queda con "a:b:c:d:e";
```

1.7.15. split

La función `split` convierte un escalar en un arreglo. Es muy útil para separar los campos en un registro de un archivo texto:

```
$a = 'a:b:c:d:e';
@a = split /:/, $a  # @a queda con ('a', 'b', 'c', 'd', 'e')
```

aqui el primer parámetro es una expresión regular.

1.7.16. sort

El operador `sort` toma una lista de valores y los ordena según el alfabeto ASCII. Por ejemplo:

```
@rocks = qw/ bedrocks slate rubble granite /;
@sorted = sort (@rocks); # bedrocks granite rubble slate
```

El operador `sort` de Perl utiliza los operadores de cadenas por defecto.

```
:~/perl/src> perl -de 0

Loading DB routines from perl5db.pl version 1.19
Editor support available.

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

main::(-e:1):   0
  DB<1> print "a" == "b"
1
  DB<2> @x = sort (10, 2);
  DB<3> p "@x"
10 2
```

Se le puede pasar como primer argumento un bloque que determina la función de comparación. dicho bloque depende de dos variables "muertas" `a` y `b`. El valor retornado por el bloque (que deberá ser -1, 0 ó 1) es utilizado como elemento de comparación. Por ejemplo:

```
> cat sortbynumber.pl
#!/usr/bin/perl -w

@a = (4, 7, 9, 12, -1);
@a = sort { $a <=> $b, } @a;

print "\@a = @a\n";
nereida:~/perl/src> sortbynumber.pl
@a = -1 4 7 9 12
```

El siguiente ejemplo ordena primero la lista `0..$#name` de acuerdo con el `array @uid`. El nuevo conjunto ordenado de índices es utilizado para producir una reordenación del `array name` conteniendo los nombres de usuario:

```
#!/usr/bin/perl -w
my @user =  `cat /etc/passwd`;
my (@name, @uid);
my $x;

for ($i=0; $i < @user; $i++) {
  ($name[$i], $x, $uid[$i]) = split ':', $user[$i];
}

@name = @name[
  sort {$uid[$a] <=> $uid[$b]} 0..$#name
];

print "@name\n";
```

El operador `sort` en un contexto escalar devuelve `undef`.

1.7.17. La Lectura en un Contexto de Lista

En un contexto de lista el operador `<STDIN>` lee todas las líneas restantes en la entrada y las almacena en una lista. Por ejemplo:

```
@lines = <STDIN>;
chomp(@lines);
```

Lee líneas hasta el final de fichero (Control-d en Unix) y almacena las líneas en el *array* `@lines`. La orden `chomp` Elimina todos los retornos de carro de todas las líneas en la lista.

1.7.18. Práctica: Fichero en Orden Inverso

Escriba un programa que lea un fichero e imprima las líneas en orden inverso.

1.7.19. Práctica: En Orden ASCIIbético

Escriba un programa que lea un fichero e imprima las líneas en orden ASCIIbético.

1.7.20. Práctica: Sin Distinguir Case

Escriba un programa que lea un fichero e imprima las líneas en orden ASCIIbético sin importar la caja, esto es si son mayúsculas o minúsculas.

1.7.21. Práctica: Indexación

Escriba un programa que lea una lista de números (entre 1 y 26) hasta el final de fichero e imprima las letras mayúsculas en el orden dado por esos números. Por ejemplo, si los números son `1, 2, 4` y `2` la salida será `A B D B`. NOTA: Use el operador `..` para construir un vector con las letras:

```
@a = 'A'..'Z' # A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

1.7.22. Ejercicio: grep

Explique la interpretación que hace Perl de los siguientes fragmentos de código. Consulte 1.7.4 y el manual de Perl para entender el uso de la función `grep`. Lea el capítulo 4 para saber mas sobre expresiones regulares:

`@a = grep /\bjose\b/, <>;`

En la expresión regular `\b` es un *ancla*: casa con una frontera (*boundary*) de palabra. Asi pues, `josefa` y `quejose` no casan con `/\bjose\b/`.

1.7.23. Práctica: Postfijo

Escriba un programa que lea una línea conteniendo una expresión en postfijo e imprima su valor. Para simplificar, asumiremos que los únicos terminales en la expresión son números enteros sin signo, sumas y productos. Asumiremos también que cualesquiera dos terminales están separados por blancos.

Por ejemplo, ante la entrada:

```
4 5  2 + *
```

El programa deberá imprimir `28`.

Sugerencias:

- Lea la cadena y conviertala en una lista usando `split`. Recuerde que la expresion regular `\s` casa con cualquier blanco Puede consultar la tabla en la sección 4.6.
- Utilice una lista auxiliar como pila para la evaluación. Los números se empujan, los operadores binarios extraen dos elementos y empujan el resultado.
- Tenga cuidado de reconocer los números con la expresión regular apropiada. Use el operador `=~` si es necesario.
- Recuerde que las expresiones lógicas se evalúan en circuito corto (véase la sección 1.2.1). La función `die` puede utilizarse para abortar la ejecución de un programa. Observe estos dos ejemplos de uso:

  ```
  $op1 = pop @stack or die("Faltan argumentos\n");
  ($x != 0) or die "No se puede dividir por cero!\n";
  ```

1.8. Hashes

Un `hash` puede verse como una tabla con dos columnas, donde la de la izquierda almacena las *claves* y la de la derecha los *valores*. Una variable `hash` se prefija con el símbolo `%`.

```
%a = ( 'x', 5, 'y', 3); # llena 2 elementos del hash
```

Aqui `x` e `y` son las claves del hash y `5` y `3` los respectivos valores. El elemento indexado en `x` de `%a` es `5` y el elemento indexado en `y` de `%a` es `3`.

Las claves deben ser únicas, aunque los valores pueden estar duplicados. Los valores de un hash deben ser escalares, pero las claves han de ser cadenas. Si no lo fueran se producirá la conversión pertinente.

1.8.1. Acceso a los elementos de un hash

Los elementos individuales de un `hash` se acceden de manera similar a como se hace en un `array`. La diferencia esta en que se usan llaves en vez de corchetes:

```
%a = ( 'x', 5, 'y', 3);
$a{x} = 7;
print  $a{x}; #  imprime: 7
print $a{'x'};  # imprime: 7
print $a{'y'};  # imprime: 3
```

En el interior de las llaves, si la clave es un identificador (incluyendo el guión bajo) se pueden omitir las comillas:

```
$a{x} es lo mismo que $a{'x'}
```

Sin embargo, deberemos escribir:

```
$sound{"homo sapiens"} = "have a nice day";
```

ya que la clave es una cadena conteniendo espacios en blanco.

1.8.2. Asignación de Hashes

Es posible asignar un hash a otro:

```
%n = %a;
```

En un contexto de lista, el valor de un hash es la lista formada por sus pares clave-valor:

```
%a = ( 'x', 5, 'y', 3);
@b = %a;
```

Esta última asignación se llama *desenrollar* (*unwinding*) el hash.

Por supuesto, aunque los pares clave-valor aparecen seguidos, el orden en que figuran en el array es el orden en el que internamente Perl accede al hash:

```
 DB<1> %a = (one=>1, two=>2, three=>3)
 DB<2> @b = %a
 DB<3> p "@b\n"
 one 1 three 3 two 2
```

Lo mas común es transformar el hash inicial de alguna forma:

```
%r = reverse %h;
```

En el hash `r` las claves de `h` son los valores y viceversa. Si los valores de `h` no fueran únicos, en `r` "triunfaría" como clave algun valor de `h`, si bien cuál, depende de la implementación particular que la versión de Perl haga del hash.

Asignar un número impar de elementos a un `hash` da lugar a un mensaje de advertencia.

1.8.3. El operador flecha grande

Cuando se tiene una lista con un número grande de pares clave/valor, es fácil perder la pista de que elementos son claves y cuales valores. Perl proporciona el operador `=>` que puede ser utilizado en lugar de la coma. Además trata el identificador a su izquierda como si estuviera entrecomillado. Asi pues, es posible escribir:

```
%a = (x => 5, y => 3, z => "woof", t => undef);
```

Existe una pequeña diferencia en funcionalidad entre el operador coma y el operador `=>`. El argumento en el lado izquierdo de `=>` se trata siempre como una cadena. Por ejemplo:

```
@a = (time => 'flies');
print "@a\n"; # "time flies"

@b = (time, flies); # El primer argumento es una llamada a la función time
print "@b\n"; # Algo asi como: "81234578 flies"
```

1.8.4. Las funciones `keys` y `values`

Perl proporciona 3 funciones intrínsecas para iterar sobre un `hash`: `keys`, `values` y `each`.

La funcion `keys` toma un `hash` como argumento y devuelve sus claves (en un orden que depende de la representación interna del `hash`). Del mismo modo, `values` devuelve los valores en el mismo orden que `keys`. Así, para imprimir todos los valores de un `hash` escribiremos:

```
foreach $key (keys %a) {
  print "The key $key has the value $a{$key}\n";
}
```

O si sólo quisieramos los valores escribiremos:

```
foreach $val (values %a) {
  print $val,"\n";
}
```

La función `each` será tratada en la sección 1.8.6.

1.8.5. Hashes y `undef`

Como en el caso de los `arrays`, no es posible asignar `undef` a un `hash`.

```
%a = ();
if (keys %a) { ... } # FALSE

%a = (lunes=>1, martes=>2, miercoles=>3, jueves=>4, viernes=>5, sabado=>6, domingo=>7 );
undef %a;
if (keys %a) { ... } # FALSE
if (defined %a) { ... } # Tambien es FALSE
```

1.8.6. La función `each`

Una alternativa es usar la función `each`. La función `each` toma un `hash` como argumento y devuelve una clave distinta cada vez que es llamada. Cuando todas las claves han sido visitadas, devuelve `undef`. Veamos un ejemplo:

```
while (defined($nextkey = each %a)) {
  print "The key $nextkey has the value $a{$nextkey}\n";
}
```

Si `each` es llamada dentro de un *contexto de lista*, devuelve una lista con dos elementos: la siguiente clave y el siguiente valor. después que todos los elementos del `hash` hayan sido visitados, `each` devuelve una lista vacía. Por ejemplo:

```
while (($nextkey, $nextval) = each %sound) {
  print "The key $key has the value $nextval\n";
}
```

Cada hash tiene su propio *iterador privado*. Esto permite el anidamiento de bucles `each`, siempre que esten iterando sobre hashes diferentes. El iterador es reinicializado automáticamente al usar las funciones `keys` o `values` sobre el hash. También es reinicializado si se almacena una nueva lista de valores en el *hash completo* o si `each` ha iterado sobre todos los elementos del mismo, alcanzando el final del hash. Sin embargo, almacenar una nueva pareja clave-valor durante la iteración no necesariamente reinicializa el iterador.

1.8.7. La función `delete`

Utilice `delete` para suprimir los pares clave-valor de un `hash`:

```
%a = (lunes=>1, martes=>2, miercoles=>3, jueves=>4, viernes=>5, sabado=>6, domingo=>7 );
delete $a{lunes};
if (exists $a{lunes}) { ... } # FALSE
```

También se le puede pasar como argumento un trozo del hash (para saber más sobre trozos de hashes vea la sección 1.8.12):

```
delete @a{martes, miercoles}; # dos dias laborales menos!
```

1.8.8. La función `exists`

Una diferenciación interesante, especialmente cuando se trata de `hashes` es entre existencia y definición:

```
%a = (lunes=>1, martes=>2, miercoles=>3, jueves=>4, viernes=>5, sabado=>6, domingo=>7 );
$a{lunes} = undef;
if (defined($a{lunes})) { ... } # FALSE
if (exists($a{lunes})) { ... }  # TRUE
```

1.8.9. Interpolación de hashes

Como es habitual, un elemento de un hash es interpolado en una cadena de comilla doble, sin embargo los hashes no son interpolados:

```
%a = (one=>1, two=>2, three=>3)
print "%a\n"
%a
print "$a{one}\n"
1
```

1.8.10. Obtener el conjunto de elementos de una lista

Si se quiere obtener un array `@out` con los elementos no repetidos de un array `@in` se puede utilizar `grep` y un `hash` auxiliar `%saw` para los elementos ya vistos:

```
my %saw = ();
@out = grep(!$saw{$_}++, @in);
```

otra forma de hacerlo (Uno de los lemas de Perl es *TIMTOWTDI*: *There is more than one way to do it!*)

```
undef %saw;
@saw{@in} = ();
@out = keys %saw;
```

1.8.11. Hashes DBM

La función `dbmopen` nos permite crear hashes persistentes. Veamos el siguiente ejemplo `dbm.pl` que utiliza un hash DBM para almacenar una gramática:

```
> cat dbm.pl
#!/usr/bin/perl -w

dbmopen(%DATA, "my_gramma", 0644) or die "can not create my_gramma $!";
$DATA{TOKENS} = "+|-|/|*|(|)|num";
$DATA{START} = "E";
$DATA{E} = "E+T|E-T";
$DATA{T} = "T*F|T/F";
$DATA{F} = "(E)|num";
```

`$DATA{TOKENS}` contiene los terminales de la gramática, `$DATA{START}` el símbolo de arranque y `$DATA{A}` contiene las reglas de producción de la variable sintáctica `A`.

Antes de la ejecución de `dbm.pl` tenemos dos programas Perl `dbm.pl` y `dbm2.pl` en nuestro directorio:

```
>ls -l
-rwxrwx--x    1 pl       users         194 Mar 10 08:44 dbm.pl
-rwxrwx--x    1 pl       users         300 Mar 10 09:54 dbm2.pl
```

Después de la ejecución de `dbm.pl` aparecen dos nuevos ficheros que contienen las claves y valores del hash persistente:

```
>ls -l
-rwxrwx--x    1 pl       users         194 Mar 10 08:44 dbm.pl
-rwxrwx--x    1 pl       users         300 Mar 10 09:54 dbm2.pl
-rw-r-----    2 pl       users       12333 Mar 10 10:54 my_gramma.pag
-rw-r-----    2 pl       users       12333 Mar 10 10:54 my_gramma.dir
```

Veamos ahora otro programa que usa el hash persistente:

```
> cat dbm2.pl
#!/usr/bin/perl -w

dbmopen(%G, "my_gramma", 0644) or die "can not create my_gramma $!";

$tokens = $G{TOKENS};
@tokens = split('\|',$tokens);

print "TOKENS: ";
{
  my $old = $";
  $" = ',';
  print "@tokens\n";
  $" = $old;
```

```
}

$start = $G{START};
print "$start -> $G{$start}\n";

for my $k (sort keys %G) {
  print "$k -> $G{$k}\n" if ($k !~ /(START)|(TOKENS)|($start)/);
}
```

Al ejecutarlo obtenemos esta salida:

```
> dbm2.pl
TOKENS: +,-,/,*,(,),num
E -> E+T|E-T
F -> (E)|num
T -> T*F|T/F
```

1.8.12. Troceado de un `hash`

Es posible seleccionar un sub-`hash` utilizando la notación que aparece en este ejemplo:

```
%a = (x => 5, y => 3, z => "woof", t => undef);
print @a{"y", "z"}; # 3woof
print @a{qw(y z) }; # 3woof
```

Dos observaciones:

1. Puesto que la lista de claves no es un único identificador se hace necesario el uso de las dobles comillas en este caso.
2. El prefijo del trozo de un hash es `@`. El símbolo `%` se reserva para el hash completo. Un trozo de un hash es una lista, no un escalar o un `hash`.

Observe también el símbolo al comienzo de la variable (dolar o arroba) determina el contexto (escalar o lista) en el que se evalúa la expresión que indexa.

Al usar un trozo de un hash en el lado izquierdo se obtiene el efecto esperado:

```
@a{"y", "z"} = ("andale, andale!", 7.2);
@a = %a;
print "@a\n";
```

La asignación `@a{"y", "z"} = ("andale, andale!", 7.2)` es equivalente a la asignación `($a{y}, $a{z}) = ("andale, andale!", 7.2)`.

Algunos otros ejemplos:

```
@char_num{'A'..'Z'} = 1..26; # crea un hash con claves 'A' .. 'Z'
@old{keys %new} = values %new; # Escribe/extiende los contenidos de %new sobre %old
delete @name{keys %invalid}; # Suprime los elementos en %invalid de %name
```

Reflexione: ¿Porqué `@name` en el último ejemplo?

1.8.13. Práctica: Ordenar por Calificaciones

Se tiene un fichero de entrada con calificaciones de alumnos como el que sigue:

```
Ferrer Pérez, Eduardo & 9'6\\
García  García, Laura & 7'5 \\
García Medina, Anai & 6'5\\
García Rodríguez, Pedro & 7'2\\
```

```
González del Castillo, Jorge & 5'0\\
Hernández Pérez, Daniel & 5'0\\
Marrero Díaz, Jose Antonio & 8'0\\
Marrero Piñero, Sergio & 7'2\\
Padrón Padrón, Adolfo & 5'0\\
Pedrín Ruiz, Ana & 9'6\\
Pedrínez Pérez, Ignacio & 7'5\\
Piñero Piñero, Antonio & 5'3\\
Pérez García, Adolfo & 9'5\\
Pérez González, Germán & 5'0\\
Pérez Pedrínez, Maria & 5'0\\
Ramos Ramos, Ignacio & 5'0\\
Rodríguez González, Andrés & 10'0\\
Rodríguez Rodríguez, Jorge & 9'2\\
```

Lea el fichero (para saber como leer un fichero cuyo nombre ha sido pasado en la línea de argumentos léa la sección 2.1) de manera que los elementos queden almacenados en un hash indexado en los nombres de los alumnos. Ordene el hash en orden decreciente de calificaciones. Observe que las números se han escrito según la convención española de usar apóstrofes en vez del punto decimal. Una posibilidad es que utilice el operador de sustitución para convertirlo a punto. Por ejemplo, `$a =~ s/'/./` sustituirá el primer apóstrofe en `$a` por un punto. También le puede ayudar el uso de paréntesis con memoria en expresiones regulares. Veamos un ejemplo:

```
$ perl -de 0 notas.txt
  DB<1> $a = <>
  DB<2> p $a
Ferrer Pérez, Eduardo & 9'6\\
  DB<3> @b = $a=~/(.*)\s+\&\s+(\d+'\d*).*/
  DB<4> p "@b"
Ferrer Pérez, Eduardo 9'6
  DB<5> x @b
0  'Ferrer Pérez, Eduardo'
1  '9\'6'
```

El uso de la expresión regular en la línea 3 en un contexto de lista hace que queden en `@b` las cadenas que casaron con las expresiones regulares en el primer y segundo paréntesis.

Puede que le convenga leer las secciones 4.1, 4.2 y 4.4 y en particular el uso de paréntesis y el operador de sustitución en las expresiones regulares.

Para la ordenación repase la sección 1.7.16 y en particular el ejemplo en el que los usuarios del sistema se ordenan según su `uid`.

1.9. Subrutinas

1.9.1. Definición de subrutinas

Las subrutinas se definen mediante la palabra clave `sub` seguidas del cuerpo entre llaves. Las definiciones de subrutina se pueden poner en cualquier lugar del texto pero son siempre definiciones globales. para invocar a una subrutina se escribe su nombre precedido de un `&`.

```
 cat nestedsubs.pl
#!/usr/bin/perl -w

sub marine {

  sub submarine {
```

```
    $n +=1;
    print "sub submarine $n!\n";
  }

  $n +=1;
  print "sub marine $n!\n";
}

&marine;
&submarine;
```

Observe como la subrutina `submarine` es accesible desde el programa principal:

```
nereida:~/perl/src> nestedsubs.pl
sub marine 1!
sub submarine 2!
```

1.9.2. Argumentos y valores de retorno

Los argumentos pasados a una subrutina están disponibles dentro del bloque via el `array` especial `@_`. El valor retornado puede hacerse explícito utilizando el operador `return <expresion>` el cual termina la subrutina. Si no se usa, el valor retornado por la subrutina es el valor de la última expresión computada.

Veamos un ejemplo:

```
> cat return.pl
#!/usr/bin/perl -w

sub dictionary_order {
  @ordered = sort @_;
  return @ordered;
}

@a = &dictionary_order(keys %ENV);

foreach (@a) {
  print "$_ $ENV{$_}\n";
}
```

El hash `%ENV` contiene las variables de entorno.

La función interna `return` finaliza inmediatamente la subrutina y hace que el valor especificado después del `return` sea devuelto como resultado. Si no hay sentencia `return` el valor devuelto es el asociado con la última sentencia ejecutada.

Observe como los argumentos en la llamada se han puesto entre paréntesis.

Al igual que las variables, las subrutinas tiene un símbolo de prefijo que indica que se trata de funciones. El nombre "formal" de una subrutina tiene el prefijo `&`, el cual puede usarse en la llamada:

```
@sorted = &dictionary_order("eat", "at", "Joes");
```

sin embargo, no debe usarse cuando se define la subrutina.

El prefijo `&` puede omitirse si por el contexto es claro que se trata de una llamada a función:

```
@sorted = dictionary_order ("eat", "at", "Joes");
@sorted = dictionary_order (@unsorted);
@sorted = dictionary_order (@sheep, @goats, "shepherd", $goathered);
```

Si una subrutina no requiere argumentos, puede ser llamada con una lista vacía de argumentos. La lista puede ser completamente omitida siempre que Perl conozca de antemano que se trata de una función. Asi tenemos:

```
sub get_next { return <>; }

prompt();  # correcto
$next = get_next(); #correcto

prompt;   # error; prompt no ha sido definido aún
$next = get_next; # correcto: get_next fue definido arriba
sub prompt { print "next> "; }
```

1.9.3. Otros modos de llamar a una subrutina

Las subrutinas que han sido definidas en una parte anterior del programa pueden ser llamadas sin usar paréntesis alrededor de la lista de argumentos:

```
sub fun1 {
  $a  = shift;   # shift asume el arreglo @_

  $y = 2  * $a;
  return ( $y );
}
# mas tarde ...
$x = fun1 4;
```

Otra forma de llamar una subrutina es usar `&` pero sin lista de argumentos. En tal caso, los contenidos actuales del `array` especial `@_` se pasan como argumentos a la subrutina. Ejemplo:

```
sub checked_inverse {
  die "can't invert 0" if $_[0] == 0;
  $inv  = &inverse; # llama a &inverse con los argumentos actuales
  die "inversion failed" unless $inv*$_[0] == 1;
  return $inv;
}
```

Observe que esto significa que hay una sutil diferencia entre:

```
&checked_inverse # significa checked_inverse(@_);
```

y

```
checked_inverse; # significa checked_inverse();
```

Una última variación utiliza el prefijo `&` pero invoca a la subrutina a tarvés de un `goto`:

```
goto &inverse;
```

Que llama a la función y le pasa los argumentos en `@_`. Pero no se vuelve a la siguiente sentencia después de la llamada. En vez de eso, `goto &inverse` reemplaza la llamada a la subrutina actual por una llamada a `inverse`. Esta forma especial de llamada se usa principalmente en las subrutinas de carga automática.

1.9.4. Tipo de objeto y ámbito

Existe siete tipos de "objetos con nombre" en Perl: variables escalares, variables `array`, variables `hash`, nombres de subrutinas, formatos, ficheros y directorios. Cada uno de estos objetos tiene su propio espacio de nombres. Un cambio en el valor de uno de estos tipos de objetos no afecta en absoluto el valor de otro objeto que tenga el mismo nombre. Por ejemplo, una modificación de la variable escalar `$a` no afecta a la variable `array` con el mismo nombre `@a`. No siempre puede considerarse que es mal estilo de programación el utilizar esta independencia en el espacio de nombres para dar a dos variables diferentes el mismo nombre. Obsérvese el siguiente ejemplo:

```
#!/usr/bin/perl -w
my $user = shift;
   $user = '.*' unless defined($user);
my @user = grep {/$user/i } `cat /etc/passwd`;
my @field;

foreach $user (@user) {
  @field = split ':', $user;
  print "$field[0]\n";
}
```

1.9.5. Variables privadas

Una *variable privada* o *variable léxica* se declara usando el operador `my`:

```
sub min {
  my $a = shift; # igual que shift @_
  my $b = shift;
  ($a < $b)? $a : $b
}
```

Estas variables tienen como ámbito el bloque que las rodea. Cualesquiera otras variables `$a` y `$b` no se ven afectadas por los cambios que les hagamos a estas.

Observe que se ha suprimido el punto y coma de la última sentencia. Al igual que en Pascal, en perl es legal hacerlo asi.

Las variables léxicas pueden declararse en cualquier bloque, no es obligatorio que sea el bloque de una subrutina.

Si no esta dentro de ningún bloque, el ámbito de la variable será el fichero actual.

El operador `my` no implica ningún contexto en particular. Por ejemplo:

```
my ($num) = @_; # contexto de lista
my $num = @_; # contexto escalar
```

¿Que valores quedan almacenados en `$num` en cada caso?

1.9.6. Variables Dinámicas

Las variables dinámicas se manejan mediante la palabra reservada `local`. En contra de lo que pudiera parecer, no son variables locales. Son variables globales (para ser mas precisos variables de paquete o '*package variables*" declaradas mediante `our` o simplemente no declaradas) cuyo valor es salvado y recuperado al finalizar el bloque en el que se las "localiza" mediante `local`. Las variables localizadas mediante `local` son globales y por tanto siguen siendo accesibles desde cualquier subrutina. La localidad se refiere al ambito temporal de su valor, pero no a su visibilidad, que es global. Así, la palabra `local` debe entenderse en el sentido de valor local (`local value`).

Una aproximación a lo que ocurre cuando se declara una variable `local` es la equivalencia que se expresa en la siguiente tabla:

<table>
<tr><th>DECLARACIÓN DE <code>local</code></th><th>SIGNIFICADO</th></tr>
<tr><td><pre>
{
  local($SomeVar);
  $SomeVar = 'My Value';
       ...
}
</pre></td><td><pre>
{
  my $TempCopy = $SomeVar;
  $SomeVar = undef;
  $SomeVar = 'My Value';
       ...
  $SomeVar = $TempCopy;
}
</pre></td></tr>
</table>

La diferencia entre variables dinámicas y léxicas debería quedar mas clara observando el siguiente ejemplo ...

```
> cat local.pl
#!/usr/bin/perl -w

sub pr { print "$a\n"; }
sub titi { my $a = "titi"; pr(); }
sub toto { local $a = "toto"; &pr(); &titi(); }

$a = "global";
&pr();
&toto();
&titi();
```

... y su ejecución:

```
> local.pl
global
toto
toto
global
```

Ciertas variables son automáticamente declaradas como dinámicas en la entrada a cada bloque, salvándose así su valor. Es como si, el compilador pusiera una declaración `local` automáticamente. Por ejemplo, esto se hace con la variable índice de un bucle. Esto es lo que permite el anidamiento de bucles como muestra el siguiente ejemplo:

```
> cat nestedfors.pl
#!/usr/bin/perl -w

for (1..3) {
  for (0..4) {
    print "$_ ";
  }
  print ": $_\n";
}

print "-----\n";

for $i (1..3) {
  for $i (0..4) {
```

```
    print "$i ";
  }
  print ": $i\n";
}

> nestedfors.pl
0 1 2 3 4 : 1
0 1 2 3 4 : 2
0 1 2 3 4 : 3
-----
0 1 2 3 4 : 1
0 1 2 3 4 : 2
0 1 2 3 4 : 3
```

Las variables especiales como `$_` pueden ser calificadas “local” pero no pueden ser calificadas con `my`.

1.9.7. El pragma `use strict`

Un *pragma* es una anotación en el código que afecta al proceso de compilación. El pragma `use strict` le indica al compilador de 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.)

En general, se recomienda que todo programa que ocupe mas de una pantalla incluya el pragma `use strict`.

1.9.8. Argumentos con Nombre

Perl soporta *argumentos con nombre* a traves de un truco “astuto”. El truco es pretender que los argumentos de la subrutina forman un `hash` en vez de una lista. Por ejemplo, supongamos que queremos escribir una subrutina con una funcionalidad equivalente a `ls` y que toma un numeroso grupo de argumentos especificando las diferentes casuisticas. La ventaja de usar `hash`es es que no hay que especificar todos los argumentos en una llamada y no hay que preocuparse por el orden de los mismos:

```
listdir(cols=>4, page=>1, hidden=>1, sep_dirs=>1);
```

Dentro de la subrutina, simplemente inicializamos un `hash` con los contenidos del `array @_` resultante. De este modo podemos acceder a los argumentos a través de su nombre, utilizando cada nombre como clave de entrada en el `hash`. Por ejemplo:

```
sub listdir {
  %arg = @_; # convierte la lista de argumentos en un hash
  # Utilizamos valores por defecto para los argumentos desaparecidos
  $arg{match} = "*" unless exists $arg{match};
  $arg{cols}  =  1  unless exists $arg{cols};
  # etc.

  # Utilizamos los argumentos para controlar la conducta ...
  @files = get_files($arg{match});
  push @files, get_hidden_files() if $arg{hidden};
  # etc.
}
```

Otra ventaja es que, si tenemos varias llamadas que requieren el mismo conjunto de argumentos, podemos almacenarlos en un `hash` separado y reutilizarlo:

```
%std_listing = (cols=>2, page=>1, sort_by=>"date");
listdir(file=>"*.txt", %std_listing);
listdir(file=>"*.log", %std_listing);
listdir(file=>"*.dat", %std_listing);
```

Esta idea de un conjunto de argumento estándar, sobreescrito por argumentos especificados explícitamente puede ser utilizando dentro de una subrutina para simplificar el manejo de los valores por defecto. Por ejemplo:

```
sub listdir {
  %defaults = (match =>"*", cols=>1, sort_by=>"name");
  %arg = (%defaults, @_);

  #etc.
}
```

Una posible optimización es mover el `hash defaults` fuera de la subrutina `listdir`, de manera que su inicialización se haga una sola vez:

```
%defaults = (match =>"*", cols=>1, sort_by=>"name");

sub listdir {
  %arg = (%defaults, @_);
  #etc.
}
```

1.9.9. `Aliasing` de los parámetros

Los parámetros de una función se pasan por "referencia". Si se modifica `$_[1]` se alteraría el segundo parámetro usado en la expresión llamadora.

```
sub fun1 {
  $_[0]=7;
  # altera el 1er parámetro en el llamador
}

$a = 5;
&fun1($a);
print  $a;  #  imprime:  7
```

1.9.10. Contexto de la llamada

Cuando se llama a una subrutina, es posible detectar si se esperaba un valor escalar, una lista o nada. Estas posibilidades definen tres contextos en los cuales una subrutina puede ser llamada. Por ejemplo:

```
listdir(@files); # contexto void
$listed = listdir(@files); # contexto escalar
@missing = listdir(@files); # contexto de lista
($f1, $f2) = listdir(@files); # contexto de lista
print (listdir(@files)); # contexto de lista
```

Esta información puede obtenerse mediante la función `wantarray`. Esta función devuelve:

- `undef` si no se esperaba valor,
- `""` si se trata de un escalar,

- 1 si se trata de una lista.

Podemos utilizar esta información para seleccionar la información apropiada a utilizar en la sentencia `return`. El siguiente programa suprime los espacios en blanco al comienzo y al final de la variable:

```
#!/usr/bin/perl -w
my @many = (" one ", "  two  ", "   three   ");
my $string = "\n\nstring\n\n";
print "string = $string\n";
print "many = (@many)\n";
$string = trim($string);
@many = trim(@many);
print "string = $string\n";
print "many = (@many)\n";

sub trim {
  my @out = @_;

  for (@out) {
    s/^\s+//;
    s/\s+$//;
  }
  return wantarray? @out : $out[0];
}

```

Ejemplo de ejecución:

```
> trim.pl
string =

string


many = ( one    two      three   )
string = string
many = (one two three)
```

1.9.11. ¿Quién llamó a esta rutina?

Existe un módulo `Carp` que nos permite localizar quién llamo a la rutina actual. La función intrínseca `caller` devuelve una lista de valores indicando:

1. El "package" desde el cuál fue llamada la subrutina
2. El nombre del fichero conteniendo el código desde el que fue llamada
3. La línea en el fichero desde el cual fué llamada

Así la típica llamada es:

```
($package, $filename, $line) = caller;
```

Cuando `caller` se llama en un contexto escalar sólo devuelve el nombre del paquete.

Se le puede pasar un argumento (`caller expr`) en cuyo caso `expr` indica el número de contextos de pila que se retroceden a partir de este. En ese caso la información devuelta es aún mas rica:

```
($package, $filename, $line, $subr, $has_args, $wantarray) = caller($i);
```

Por ejemplo, el siguiente código nos da el nombre de la función actual:

```
  $this_function = (caller(0))[3];
```

1.9.12. Práctica: Máximo

Escriba una función que calcule el máximo de los elementos de una lista. El primer argumento de la función es una cadena conteniendo el operador de comparación a usar. Utilice la función `eval`. Esta función recibe una cadena y evalúa la expresión Perl que representa.

1.9.13. Práctica: Polares a Cartesianas

Escriba una función `p2c` que permita pasar de coordenadas Polares a Cartesianas y de Cartesianas a Polares. Las fórmulas de conversión de Polares a Cartesianas son:

```
$x = $r*cos($angle); $y = $r*sin($angle)
```

y de Cartesianas a Polares son

```
$r = sqrt($x*$x+$y*$y); $angle = $atan($y/$x)
```

Vea `perldoc perlfunc` o `man perlfunc` para mas información sobre las funciones disponibles. Utilice el método de los argumentos con las claves (`x`, `y`, `r`, `angle`) para determinar el tipo de conversión requerida. La función retorna a su vez un hash con el mismo convenio de claves. Así, si se llama a la función con otra combinación de argumentos como: `p2c(x=>1, r=>1)` devolverá un hash con los valores de `y` y `angle`.

Es un error llamar a la función con menos de dos argumentos.

La función `atan2(Y,X)` devuelve la arcotangente de `Y/X` en radianes en el rango $[-\pi, \pi]$. Para calcular la tangente puede usar la función `Math::Trig::tan` o simplemente `sub tan { sin($_[0]) / cos($_[0]) }`. Por ejemplo:

```
DB<1> p (atan2 1, 0)*2
3.14159265358979
DB<2> Math::Trig::tan(3.14159265358979/4)
Undefined subroutine &Math::Trig::tan called at (eval 9)[/usr/share/perl/5.8/perl5db.pl:619] line 2.
DB<3> use Math::Trig
DB<4> p Math::Trig::tan(3.14159265358979/4)
0.999999999999998
```

1.9.14. Práctica: Postfijo y Subrutina

Reescriba la práctica de la calculadora en posfijo realizada en la sección 1.7.23 para que sea una subrutina. Escriba el programa usando dos subrutinas: una que haga el cálculo y otra que haga el análisis léxico. La subrutina que hace el análisis léxico recibe como entrada una cadena de caracteres y produce la lista de terminales. Hay tres tipos de terminales: números (`\d+`) y los símbols `+` y `*`. No asuma que los terminales están separados por blancos; esto es, una entrada legal podría ser:

```
4 5 3+*
```

Una posibilidad es usar el operador de sustitución `s/regexp/subst/g` con la opción global (`g`) para insertar los blancos que faltan. Vea las secciones 4.2 y 4.12. Una alternativa es el operador `splice` (sección 1.7.7).

La subrutina que hace el cálculo recibe como entrada la lista con los terminales y devuelve el resultado de la operación.

1.9.15. Ejercicio: Prioridad de Operaciones

Observe la evaluación de las siguientes expresiones:

```
> perl -de0
Default die handler restored.
Loading DB routines from perl5db.pl version 1.07
Editor support available.
Enter h or ‘h h’ for help, or ‘man perldebug’ for more help.
main::(-e:1):   0
DB<1> print (2+3)*5
5
DB<2>print 5*(2+3)
25
DB<3>
```

¿Podría explicar los resultados?. La función `print` devuelve 1 o 0 dependiendo de si pudo realizar la impresión o no. Observe esta otra prueba:

```
$a = print (2+2)*5  # imprime 4
print $a  # imprime 5
```

Moraleja: ponga paréntesis en todas las llamadas a función en las que pueda aparecer alguna ambiguedad.

1.9.16. Ejercicio: Significados de la Coma

La coma tiene varios significados en Perl: como separador de listas, como operador ... Como separador de listas es su significado habitual en una llamada a subrutina, aunque podría producirse alguna ambiguedad. Como operador, en un contexto escalar se evalúa como en C: evalúa su argumento izquierdo, lo descarta y evalúa el derecho. El operador de coma tiene una prioridad muy baja. Explique los resultados de las siguientes operaciones:

```
$a = 4,5,6; # contexto escalar: $a es 4
$a = (4,5,6); # contexto escalar:
              # la parte derecha es una expresión
              # escalar. $a es 6
@a = (4,5,6); # contexto de lista
$a = @a;  # $a es 3
$a = print (4,3),7
43
print $a; # 1
```

¿Quién tiene mas prioridad, el operador de asignación `=` o la coma `,`?

Capítulo 2

Entrada /Salida

2.1. El operador diamante

El operador diamante, denotado por `<>`, es un operador de entrada. Toma la entrada desde los ficheros cuyos nombres han sido pasados al programa en la línea de comandos:

```
>./miprog datos.dat file.txt x.log
```

el operador diamante, cuando sea llamado desde `miprog` tomara la entrada desde `datos.dat`, `file.txt` y `x.log`. Si el programa se llama sin argumentos leera desde la entrada estándar.

Se puede usar el guión corto como sinónimo de leer desde la entrada estándar. Veamos el siguiente programa:

```
$ cat diamond.pl
#!/usr/bin/perl -w
my $n;
while (<>) {
  printf("%2d %s", $n++, $_);
}
```

Lo ejecutamos:

```
> diamond.pl tan.pl - useswitch.pl
 0 #!/usr/bin/perl -w
 1 #use strict;
 2
 3 print tan(3.1415);
Esto fue escrito por el usuario
 4 Esto fue escrito por el usuario
 5 #!/usr/local/bin/perl5.8.0 -w
 6
 7 use Switch;
 8
 9 $val = shift;
10
11 switch ($val) {
12   case 1 { print "number 1\n" }
13   case "hello" { print "string hello\n" }
14   else { print " none of these two\n" }
15 }
```

Después de imprimir el fichero `tan.pl` la ejecución se detiene para esperar entrada desde `STDIN`. La línea anterior a la 4 fué escrita por el usuario. Después se muestran los contenidos del fichero `useswitch.pl`.

El operador diamante utiliza el array `@ARGV` el cual contiene la lista de argumentos pasados al programa en la línea de comandos. Cuandose usa el operador `shift` fuera de una subrutina, actúa por defecto sobre este array. Si al comienzo de nuestro programa modificamos el array `@ARGV` podemos alterar la conducta del operador diamante. El ejemplo que vemos a continuación modifica `@ARGV` de manera que el programa hará caso omiso de los argumentos pasados por el usuario.

```
> cat diamond2.pl
#!/usr/local/bin/perl5.8.0 -w

@ARGV = qw#min.pl jump.pl#;
my $n = 0;
while (<>) {
  print $n++." $_";
}
```

Veamos un ejemplo de ejecución:

```
> diamond2.pl tan.pl - useswitch.pl
0 #!/usr/bin/perl -w
1
2 sub min {
3   my $a = shift; # igual que shift @_
4   my $b = shift;
5   ($a < $b)? $a : $b
6 }
7
8 print min(4,9)."\n";
9 print min(5,2)."\n";
10 #!/usr/bin/perl -w -I. -IModexample
11 use Modexample::Hops qw(one two);
12
13 $row = Modexample::Hops::hop_along 1, 5, 1;
14 while ($row->($r)) {
15   $col = Modexample::Hops::hop_along 1, 5, 1;
16   while ($col->($c)) {
17     print("($r, $c)\t");
18   }
19   print "\n";
20 }
21
22 Modexample::Hops::noexisto("pepe", "eowyn", "eomer", "maria");
```

2.1.1. Práctica: Enumerar Ficheros

Escriba un programa que produzca como salida los contenidos de los ficheros pasados como argumentos, numerando las líneas. El primer argumento del programa es un número, que indica el número de columnas utilizado para sangrar el número. El número deberá quedar separado de la línea por un espacio en blanco. Para ello use la función `printf` la cual funciona como su homóloga C. Por ejemplo:

```
printf "%8d $_",$n;
```

2.1.2. Ejercicio: Salida con Formato

Indique como saldrá formateada la salida del siguiente código:

```
my @items = qw(un dos tres);
my $format = "Los items son:\n".("%10s\n"x @items);
printf $format, @items;
```

2.1.3. Ejercicio: Contextos y E/S

El operador `<>` nos permite leer desde el flujo de entrada estandar. Así

```
$x = <>;
```

lee una línea y almacena su contenido (incluyendo el retorno de carro final) en `$x`. Este operador es también sensible al contexto. Así, en un contexto de lista

```
@x = <>;
```

Se lee todo el fichero (en entrada estadar unix, se leerán todas las líneas hasta que se pulse CTRL-D) y las diferentes líneas consituyen los elementos del `array x`.

¿En que contexto interpreta Perl el siguiente fragmento de código?

```
print sort <>;
```

2.2. Filehandles

En Perl un `filehandle` es el nombre una conexión de entrada/salida que conecta nuestro proceso Perl con el mundo exterior. Esto es, se trata del nombre de una conexión, no necesariamente del nombre de un fichero. Existen seis `filehandles` que son especiales: `STDIN`, `STDOUT`, `STDERR`, `DATA`, `ARGV` y `ARGVOUT`.

La aparición del terminal `__END__` en un programa indica el arranque del manipulador de ficheros especial `DATA`. Este fichero especial se refiere a todo el texto que sigue al *token* `__END__` en el fichero que contiene el guión Perl. Así una línea como `$line = <DATA>` leerá desde el manipulador de fichero formado por las líneas posteriores a aquella en la que figura `__END__`.

2.2.1. Abrir un fichero

Para abrir una conexión se utiliza el operador `open`. Por ejemplo:

```
if (open FILE, "alus.txt"   ) { ... # Para lectura   }
if (open FILE, "<alus.txt"  ) { ... # Para lectura   }
if (open FILE, ">alus.txt"  ) { ... # Para escritura }
if (open FILE, ">>alus.txt" ) { ... # Para añadir    }
```

El operador `open` devuelve verdadero o falso, dependiendo de si la operación pudo realizarse con éxito o no.

Se puede usar una expresión escalar en lugar del especificador de fichero. Por ejemplo:

```
my  $outfile = "alu.txt";
open FILE, "> $outfile";
```

Observe el espacio después del "mayor que". Asi se evitan extrañas conductas, si por ejemplo `$outfile` es algo asi como `>alu.txt`, se podriá producir un *append* (esto es `>>`) en vez de una escritura (`>`).

2.2.2. Cerrar un fichero

Para cerrar el fichero use el operador `close`:

```
close FILE;
```

Perl cierra automáticamente un fichero cuando se reabre o bien cuando termina la ejecución del programa.

2.2.3. Errores al abrir un fichero

La típica frase Perl para abrir un fichero es:

```
open FILE, ">>logfile" or die "No se puede crear el fichero: $!";
```

Esta expresión aprovecha la evaluación en circuito corto. La función `die` imprime el mensaje a `STDERR` y hace que nuestro programa termine con un estatus distinto de cero. Además añade al mensaje el nombre del programa y el número de línea en caso de que el mensaje de error no termine en un retorno de carro (`\n`). La variable especial `$!` contiene el último mensaje de error del sistema operativo.

2.2.4. Lectura desde un fichero

Una vez que un fichero esta abierto para lectura, podemos leer las líneas de la misma forma que lo hacemos desde `STDIN` usando el operador de lectura `<FILEHANDLE>`. Veamos un ejemplo:

```
my $user = shift;
open PASSWD, "/etc/passwd" or die "Se esperaba un sistema Unix. $!";
while (<PASSWD>) {
  chomp;
  if (/^$user:/) { print "$_\n"; }
}
```

2.2.5. El separador de lectura

La variable especial `$/` contiene el separador de lectura, que por defecto es un `\n`. Asi, si le asignamos `$/ = '.';` en la siguiente lectura se leerá hasta el próximo punto o hasta el final del fichero si no lo hubiera. Asignarle a `$/` la cadena vacía `""` hace que se lea hasta el siguiente párrafo (esto es, hasta la siguiente aparición de dos o más líneas en blanco). Cuando tiene el valor `undef` se leerá todo el resto del fichero.

```
undef $/;
$x = <FILE>; # Ahora $x contiene todo el fichero
```

2.2.6. Ejercicio: Muerte Prematura

Considere el siguiente código:

```
open FILEHANDLE, shift;
while (<FILEHANDLE>) {
  print;
}
```

¿Existe el riesgo de "muerte" prematura debido a que una línea contenga sólamente `"0"`? Pruebe con varios posibles ficheros de entrada. Observe que el operador de lectura `<FILEHANDLE>` incluye el retorno de carro `\n` en `\$_`.

2.2.7. El operador `select`

El operador `select` modifica la salida por defecto. En vez de `STDOUT` el fichero especificado será utilizado por defecto. La variable especial `$|`, cuando vale 1, hace que los *buffers* de salida se vacíen inmediatamente a la salida por defecto.

```
> cat select.pl
#!/usr/bin/perl -w

my $user = shift;
open LOG, ">/tmp/log.file" or die "Se esperaba un sistema Unix";
```

```
select LOG;
$| = 1;
print "Esto es una prueba\n";
select STDOUT;
print "Esto es otra prueba\n"
```

Veamos una ejecución:

```
nereida:~/perl/src> select.pl
Esto es otra prueba
nereida:~/perl/src> cat /tmp/log.file
Esto es una prueba
```

2.2.8. El separador de campos de salida

La variable `$,` contiene el separador para el operador `print`. Normalmente no hay separación entre los argumentos de `print`. Esto puede modificarse como muestra el ejemplo:

```
print 1,2,3 # Salida: 123
$, = ',';
print 1,2,3 # Salida: 1,2,3
$, = ';';
print 1,2,3 # Salida: 1;2;3
```

2.2.9. El separador de registros de salida

Perl normalmente no separa dos salidas realizadas en dos llamadas consecutivas a la sentencia `print`. Para ello se puede usar la variable `$\`. Vea el ejemplo:

```
$\ = "\n***\n";
print "uno"; print "dos";
```

La salida será:

```
uno
***
dos
***
```

2.2.10. Reapertura de uno de los ficheros estandar

Si reabrimos un fichero, el viejo será cerrado automáticamente. Si se trata de uno de los tres ficheros estándar y falla la reapertura se restaura el original. Así para redirigir los errores escribiríamos:

```
open STDERR, ">>/home/casiano/.error.log"
  or die "No se pudo abrir fichero de errores: $!";
```

2.2.11. Tests sobre ficheros

Perl utiliza `-e $filevar` para comprobar la existencia de un fichero cuyo nombre es el guardado en la variable `$filevar`. Si el fichero existe el resultados es verdadero; en otro caso es falso. Por ejemplo:

```
$name = "index.html";
if (-e $name) {
    print "Ya existe un fichero denominado $name\n";
} else {
    print "No existe un fichero denominado $name\n";
}
```

He aqui otro ejemplo:

```
if (-e "index.html" && -e "index.cgi") {
    print "Encontrados.\n";
}
```

Existen otros operadores. Por ejemplo, `-r $filevar` es cierto si el fichero cuyo nombre se guarda en `$filevar` existe y es de lectura. Análogamente, `-w $filevar` comprueba si es de escritura.

```
print "Donde? ";
$filename = <STDIN>;
chomp $filename;
if (-r $filename && -w $filename) {
        # El fichero existe y es de lectura y escritura
        ...
}
```

La tabla 2.1 contiene algunos de los operadores mas importantes.

Fichero	Significado
-r	El fichero o directorio es legible
-w	El fichero o directorio es de escritura
-e	El fichero o directorio existe
-x	El fichero es ejecutable
-z	El fichero existe y tiene tamaño cero (Los directorios nunca tiene tamaño cero)
-s	El fichero o directorio existe y tiene tamaño no nulo (el tamaño en *bytes*
-f	La entrada es un fichero
-d	La entrada es un directorio
-t	`isatty` sobre el fichero es cierto (esto es, es un dispositivo de caracteres)
-T	Fichero de texto
-B	Fichero binario
-M	Edad de modificación en días
-A	Tiempo de acceso en días
-C	Edad de modificación del Inode en días

Cuadro 2.1: Operadores de fichero y su significado

Estos operadores pueden usarse indistintamente sobre `filehandles` o nombres de fichero. Por ejemplo:

```
if (-x SOMEFILE) {
        # SOMEFILE es ejecutable
}
```

Si no se especifica el nombre del fichero, el operador por defecto es la variable `$_`. Por ejemplo:

```
foreach (@some_list_of_filenames) {
        print "$_ es de lectura\n" if -r; # Lo mismo que -r $_
}
```

2.2.12. Práctica: Ficheros Grandes y Viejos

Escriba una función que recibe como argumentos un tamaño `tam`, un número de días `days` y una lista de nombres de ficheros y devuelve una sublista de la anterior con los nombres de los ficheros cuyo tamaño (operador `-s`) es superior a `tam` y que no hayan sido modificado en los últimos `days` días (operador `-M`). Compruebe que su programa funciona sin errores ni advertencias bajo el pragma `use strict`.

2.2.13. La función `stat`

Si se quiere obtener información más detallada como el número de enlaces a un fichero o el `uid` del propietario de un fichero es necesario recurrir a la función `stat`. El operando de `stat` es un fichero o un nombre de fichero. La función `stat` devuelve bien una lista vacía (en caso de error) o una lista de 13 elementos (véase la tabla 2.2).

0	dev	device
1	ino	inode
2	mode	permisos
3	nlink	numero de hard links
4	uid	user ID del propietario
5	gid	group ID del propietario
6	rdev	tipo de dispositivo (ficheros especiales)
7	size	tamaño total, en bytes
8	atime	Tiempo del último acceso
9	mtime	Tiempo de la última modificación
10	ctime	Tiempo del último cambio del inodo
11	blksize	blocksize para filesystem I/O
12	blocks	número de bloques asignados

Cuadro 2.2: Valores devueltos por `stat`

Los argumentos `atime`, `mtime` y `ctime` indican cuantos segundos han pasado desde el comienzo de 1970 MUT (*Midnigh Universal Time*).

En el siguiente ejemplo, en la línea 1 el uso del operador `glob` nos permite la expansión de los comodines tipo *shell*:

```
  DB<1> @f = glob('xml*')
  DB<2> p "@f"
xml xmlparser.pl xmlparserinput.xml
  DB<3> @a = stat "xmlparser.pl"
  DB<4> x @a
0  2051
1  2171300
2  33261
3  1
4  1007
5  1007
6  0
7  191
8  1112287076
9  1087853581
10  1099385099
11  4096
12  8
```

La función `localtime` permite convertir números en formato MUT en una cadena tipo fecha:

```
  DB<1> @a = stat "xmlparser.pl"
  DB<2> use constant mtime => 9
  DB<3> x $a[mtime]
0  1087853581
  DB<4> p scalar(localtime $a[mtime])
Mon Jun 21 22:33:01 2004
```

Cuando se llama a `stat` sobre un enlace simbólico, `stat` devuelve información sobre el fichero apuntado, no sobre el enlace. Si se trata de un enlace en vez de un fichero, use la función `lstat`. Si el operando no es un enlace simbólico `lstat` devuelve la misma información que `stat`. Cuando no se indica operando, ambos `stat` y `lstat` usan la variable por defecto `$_`.

2.2.14. La función `localtime`

Cuando `localtime` es llamada en un contexto de lista devuelve una lista como sigue:

```
#  0    1    2     3     4    5     6     7     8
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime(time);
```

Vea el siguiente ejemplo de uso:

```
  DB<1> @f = `ls`
  DB<2> chomp(@f) # eliminamos retornos de carro finales
  DB<3> x @f[0..5] # veamos los nombres de los 6 primeros ficheros
0  'a2pdf'
1  'Abstract.pm'
2  'adressbook.pl'
3  'advanced_perl_programming'
4  'amatch.pl'
5  'A.pl'
  DB<4> @s = stat $f[0]
  DB<5> x @s
0  2051
1  2171079
2  33261
3  1
4  1007
5  1007
6  0
7  339
8  1111133369
9  1092043518
10  1096562651
11  4096
12  8
  DB<6> use constant mtime => 9 # definimos la constante mtime como 9
  DB<7> p $s[mtime] # formato Midnight Universal Time
1092043518
  DB<8> @t = localtime $s[mtime] # contexto de lista
  DB<9> x @t
0  18  # segundos
1  25  # minutos
2  10  # horas
3  9   # día del mes
4  7   # mes: Agosto. Enero es el 0.
5  104 # año desde 1900: 1900+104 = 2004
6  1   # dia de la semana: Lunes. Domingo es el 0.
7  221 # dia del año
8  1   # daylight savings time (cambio de horario) es cierto
  DB<10> $t = localtime $s[mtime] # contexto escalar
  DB<11> x $t
0  'Mon Aug  9 10:25:18 2004'
```

Cuando no se le da argumento a `localtime` devuelve el correspondiente al MUT devuelto por la función `time`:

```
  DB<1> $d = localtime
  DB<2> p $d
Thu Mar 31 18:46:47 2005
```

La función `gmtime` es similar a `localtime` sólo que devuelve el tiempo Greenwich:

```
  DB<3> $g = gmtime
  DB<4> p $g
Thu Mar 31 17:47:13 2005
  DB<5> $c = time
  DB<6> p $c
1112291278
  DB<7> p localtime $c
5847183121054891
  DB<8> p "".localtime $c
Thu Mar 31 18:47:58 2005
  DB<9> p scalar(localtime $c)
Thu Mar 31 18:47:58 2005
  DB<10> p scalar localtime(time-(24*60*60)), "\n";
Wed Mar 30 19:14:33 2005
```

Ejercicio 2.2.1. *Explique los resultados en los pasos 7, 8,9 y 10 del ejemplo anterior.*

Ejercicio 2.2.2. *Usando el depurador de Perl calcule que día del calendario fué hace 500 días.*

Si se quiere tener mayor precisión en la medida del tiempo, use el módulo `Time::HiRes`. Sigue un ejemplo de uso:

```
  DB<1> use Time::HiRes
  DB<2> p time
1112339548
  DB<3> p Time::HiRes::time
1112339556.72756
```

Puede verse como se usa un número flotante incluyendo no sólo los segundos sino los microsegundos MUT. Si queremos sustituir la función `time` por la que provee `Time::HiRes` deberemos importarla nombrándola explícitamente en el pragma `use`:

```
  DB<4> use Time::HiRes "time"
  DB<5> p time
1112339573.30229
```

2.3. Directorios

2.3.1. Acceso mediante Operadores

El operador `chdir` nos permite cambiar de directorio dentro de la jerarquía de directorios. Estudie el siguiente ejemplo:

```
> cat chdir.pl
#!/usr/bin/perl -w

my $directory = shift;
chdir $directory or die "no encuentro $directory: $!";
```

```
my @files = `ls -a`;
my $i = 1;
for (@files) {
  chop;
  printf "$_\t";
  print "\n" if ($i++ % 5 == 0);
}
print "\n";

> chdir.pl '/tmp/'
.       ..      .301.2112f1     .ICE-unix       .X0-lock
.X11-unix       .fam_socket     .famwe4uVt      .famztg4KC      .font-unix
jtxttst.pdf     kde-casiano     ksocket-casiano magicbM4Uda     mcop-casiano
node26.html     node26_files    node27.html     node27_files    plhtml
session_mm_apache0.sem  ssh-XXPTlSpX    webeqman.pdf
```

Otra forma de manipular un directorio es mediante el uso de un *directory handle* o *manipulador de directorio*. Podemos abrirlo con `opendir`, leer del mismo el siguiente nombre de fichero mediante `readdir` y cerrarlo con `closedir`. Véase el ejemplo:

```
> cat dirhandles.pl
#!/usr/bin/perl -w

my $directory = shift;
opendir DH, $directory or die "No encuentro $directory: $!";
$i = 1;
foreach $file (readdir DH) {
  printf "$file\t";
  print "\n" if ($i++ % 5 == 0);
}
closedir DH;
print "\n";

> dirhandles.pl /tmp
.       ..      .font-unix      session_mm_apache0.sem  .X11-unix
ssh-XXPTlSpX    .X0-lock        kde-casiano     ksocket-casiano .ICE-unix
.fam_socket     .famwe4uVt      mcop-casiano    .famztg4KC      .301.2112f1
node26_files    node26.html     webeqman.pdf    jtxttst.pdf     node27_files
node27.html     plhtml  magicbM4Uda
```

Observe como en la salida producida los ficheros no parecen seguir un orden especial. Los nombres devueltos por `readdir` no contienen el camino, sólo el nombre.

La tabla 2.3 recoge de manera sucinta algunos otros operadores para la manipulación de ficheros y directorios.

Para saber mas, lee con detalle la documentación de los módulos `File::Basename` y `File::Spec`. Usa para ello el comando `man File::Basename` o bien `perldoc File::Basename`.

```
  DB<1> use File::Basename
  DB<2> @a = fileparse("/home/casiano/Lperl/src/dbm.pl")
  DB<3> x @a
0  'dbm.pl'
1  '/home/casiano/Lperl/src/'
2  undef
```

`rename "/tmp/old", "/tmp/new"`	Renombra un fichero existente
`unlink "one", "two", "three"`	Similar a `rm one two three`
`link "one", "two"`	Similar a `ln one two`
`symlink "one", "two"`	Similar a `ln -s one two`
`mkdir "one"`	Equivalente a `mkdir one`
`rmdir "one"`	Suprime el directorio `one`
`chmod 0755, "one", "two"`	Cambia los permisos
`chown $uid, $gid, glob "*.o"`	Cambia propietario y grupo
`getpwnam "casiano"`	devuelve el `uid` del usuario `casiano`
`getgrnam "users"`	devuelve el `gid` del grupo `users`

Cuadro 2.3: Comandos para el manejo de directorios

2.3.2. Acceso Mediante `glob`

El operador `glob` permite la expansión de los comodines de la *shell*. Asi en una expresión como:

```
my @all_files = glob "*";
```

la variable `all_files` guarda ordenados alfabéticamente la lista de los nombres de los ficheros en el directorio actual.

Es posible, como indica el siguiente ejemplo utilizar varios patrones separados por espacios.

```
my @including_hidden = ".* *";
```

Usar `glob` es mas eficiente y mas portable que enviar el requerimiento a la shell del sistema usando backticks (como por ejemplo en `` `ls .*` ``).

Cuando en el operador de lectura `<>` el argumento es una cadena *glob* de descripción de ficheros con comodines, el *globbing* se produce automáticamente:

```
my $dir = "/home/casiano/";
my $dir_files = <$dir/* $dir/.*>;
```

También puede hacerse en un contexto de lista:

```
DB<1> @modules = <*.pm>
DB<2> p "@modules"
A.pm Abstract.pm B.pm C.pm DandC.pm Frac.pm Fraction.pm
```

En general, si el identificador entre ángulos es un `filehandler`, Perl hace un acceso a través de él, en caso contrario interpreta que se trata de una operación de `globbing`. Esto es asi, incluso si la lectura hace uso de un manipulador indirecto que use variables intermedias. Véase el siguiente ejemplo:

```
  DB<1> $f = 'yapp'
  DB<2> @f = <$f/*.yp>
  DB<3> p "@f"
yapp/aSb.yp yapp/Autoaction1.yp yapp/Calc.yp yapp/Calc2.yp
  DB<4> open F, 'logic.pl'
  DB<5> $x = F
  DB<6> @x = <$x>
  DB<7> p "@x"
#!/usr/bin/perl -w
 $a = 4; $b = "hola"; $c = 0; $d = "";
 print $a && $b,"\n";
 print $a and $b,"\n";
 print ($a and $b),"\n";
```

Obsérve que en un bucle `for (list)` el argumento `list` se evalúa en un contexto de lista.

```
my $bytes = 0;
for (<*.c>) { $bytes += -s }
```

En este código, la lectura del directorio `<*.c>` se produce en un contexto de lista. Por ello el directorio completo es leído y produce una lista con todos los nombres de ficheros `*.c` existentes en el directorio actual. El subsecuente proceso iterativo acumula en `$bytes` el tamaño de los ficheros `*.c`.

En ocasiones es importante distinguir entre falsedad e indefinición, como en este ejemplo:

```
while ($file = <*>) {
  do_something($file);
}
```

En este código, en cada pasada del bucle el operador `<*>` produce un nuevo nombre de fichero del directorio actual. El nombre es asignado a `$file`. ¿Que ocurre si el nombre del fichero es `"0"`?. En tal caso el bucle tendrá una "muerte" prematura. La codificación correcta sería:

```
while (defined($file = <*>) {
   do_something($file);
}
```

Sin embargo, siguiendo la filosofía de "ayudar" al programador, las últimas versiones de Perl tiene el comportamiento "esperado". Esto es, detectan el uso en un `while` del `glob` e interpretan que `"0"` como nombre de fichero no es falso. Sin embargo en otro contexto, `"0"` si se interpreta como falso. Observe el siguiente ejemplo:

```
$ cat muerte_prematura2.pl
#!/usr/bin/perl -w

my $file;
my @a = ("uno","0","dos");

while ($file = shift @a) {
  do_something($file);
}

sub do_something {
  print "$file\n";
}
```

Al ejecutar obtenemos:

```
$ ./muerte_prematura2.pl
uno
```

2.3.3. Práctica: Viejos y Grandes Recursivo

Lea la documentación de `File::Find`. Este módulo permite un recorrido recursivo de los directorios. He aqui un ejemplo de uso:

```
  DB<1> use File::Find
  DB<2> sub Wanted { /\.yp$/ or return; print $_,"\n" }
  DB<3> find(\&Wanted, '.')
Autoaction1.yp
Calc.yp
Calc2.yp
Infixtree.yp
Yacc.yp
```

La función `find` espera como primer argumento una referencia a una subrutina. El operador Perl `\` actúa de manera similar al *ampersand* en C: nos devuelve una referencia al objeto al que se le aplica. Por ejemplo:

```
$ra = \$a;  # referencia a escalar
$rb = \@b;  # referencia a arreglo
$rc = \%c;  # referencia a hash
$rf = \&f;  # referencia a subrutina
$rx = \$rb; # referencia a referencia
```

Usando `File::Find` escriba una función que muestre los ficheros en el directorio actual mayores que un cierto tamaño dado.

2.4. Operaciones con ficheros, links y directorios

2.4.1. `unlink`

Use el operador `unlink` para suprimir ficheros:

```
unlink "one", "two", "three", glob "*.o";
```

Devuelve el número de ficheros suprimidos.

2.4.2. `symlink`

La función `symlink` crea un link simbólico:

```
cat ln.pl
#!/usr/bin/perl -w

symlink "/home/pl/public_html/regexpr/perl/doc", "doc"
  or warn "No se pudo construir enlace simbólico: $!";
```

2.4.3. `mkdir`

Para crear un directorio use `mkdir`:

```
mkdir "tmp", 0755 or warn"No se pudo crear directorio: $!";
```

Asegurese de que el argumento con los permisos es evaluado en un contexto escalar numérico y no escalar cadena. Por ejemplo:

```
my ($name, $perm) = @ARGV;
mkdir $name, $perm or die "No se pudo crear $name: $!";
```

No funcionará, incluso si se llama con algo como `prog newdir 0755`. ¿Porqué?

2.4.4. `rename`

La función `rename` nos permite cambiar el nombre de un fichero o moverlo de ubicación:

```
rename "viejo", "nuevo";
rename "/home/casiano/m.txt /tmp/m.txt";
```

La función `rename` tiene un efecto similar al comando `mv` de Unix.

2.4.5. Práctica: Renombrar Tipos de Ficheros

Escriba una función que reciba tres argumentos: un nombre de directorio `dir` y dos cadenas `old` y `new`. La función renombra todos los ficheros en el directorio `dir` con el tipo `old` para que tengan el tipo `new`. Para un fichero dado `file.old`, compruebe antes de hacer el renombrado que no existe un fichero con nombre `file.new`. Use para ello el operador `-e` (sección 2.2.11). El operador `rename`, como `mv` no hace comprobación de ningún tipo. Para resolver la práctica, puede usar *globbing* para leer los nombres de fichero en el directorio (sección 2.3.2). Use el operador de sustitución `s/regexp/string/` para obtener el nuevo nombre del fichero.

2.4.6. `rmdir`

Utilice la función `rmdir` para suprimir directorios:

```
rmdir glob "tmp/*.txt"; # elimina los directorios vacíos bajo tmp
```

2.4.7. `chmod`

Utilice la función `chmod` para cambiar los permisos de un fichero o directorio:

```
chmod 0755, "one", "two";
```

2.4.8. `chown`

Utilice la función `chown` para cambiar el propietario y el grupo de un fichero o directorio:

```
my ($user, $group) = (1004, 100);
chown $user, $group, glob "/tmp/*.c";
```

Utilice la función `getpwnam` para traducir de nombre a número y la función `getgrnam` para traducir de un nombre de grupo a su número:

```
defined(my $user = getpwnam "alu4713") or die "no existe alu4713";
defined(my $group = getgrnam "users") or die "no existe users";
chown $user, $group, glob "/home/alu4713/*";
```

Capítulo 3

Gestión de Procesos

3.1. La función `system`

La forma habitual de lanzar un proceso en Perl es a través el comando `system`:

```
system 'ls -l $PATH'
```

En el ejemplo hemos usado comillas simples para impedir que Perl interpole la variable `$PATH`. Quien debe interpolarla es la shell que ejecutará el comando. También es posible escribir:

```
system "ls -l \$PATH"
```

cuando el comando usado como argumento no implica redirección, canales (pipes), etc. el comando es ejecutado directamente por Perl. En caso contrario Perl llama a la shell para que ejecute el comando. En este segundo caso el comando se convierte en un proceso nieto del programa Perl.

Si se llama al comando `system` con mas de un argumento Perl no llamará a la shell, cualquiera que sea la forma del comando. En ese caso, el primer argumento se interpreta como el nombre del comando y los siguientes como los argumentos para el mismo.

El valor devuelto por `system` es el devuelto por el proceso hijo.

```
$status = system('command');
```

Retorna `-1` si el programa no se pudo arrancar. Con `perldoc perlvar` puede obtener información sobre la variable `$?`, la cuál contiene el estatus retornado por el programa. Un ejemplo de uso, comprobando el estatus es:

```
if ($? == -1) {
    print "No se pudo ejecutar: $!\n";
}
elsif ($? & 127) {
    printf "El proceso hijo a muerto con señal %d, %s coredump\n",
        ($? & 127),  ($? & 128) ? 'con' : 'sin';
}
else {
    printf "Proceso hijo termina con estatus %d\n", $? >> 8;
}
```

Se puede usar una segunda sintáxis con `system`, separando los argumentos mediante comas:

```
$status = system('command', 'arg1','arg2');
```

En este caso, es Perl quien ejecuta el comando y ningún metacarácter shell (redirecciones, pipes, etc.) será interpretado. Por ejemplo:

```
@args = ("command", "arg1", "arg2");
system(@args) == 0 or die "system @args failed: $?"
```

3.2. La función exec

La función `exec` actúa de manera análoga a `system`. La diferencia es que el proceso padre es sustituido por el proceso a ejecutar y no se retorna al programa inicial:

```
exec 'cat -n /etc/passwd';
die "no se pudo ejecutar cat: $!";
```

El nuevo proceso tendrá exactamente el mismo `PID` que el viejo. Al igual que con `system`, si se usa la sintáxis con lista de argumentos los metacaracteres shell no son interpretados.

```
$ cat exec.pl
#!/usr/bin/perl -w
exec '/bin/echo', 'Los argumentos son: ', @ARGV;
lhp@nereida:~/Lperl/src$ ./exec.pl 'uno' '| ls ' '| who > quien'
Los argumentos son:  uno | ls  | who > quien
```

3.3. Variables de entorno

El hash `%ENV` contiene las variables de entorno. Su modificación implica la modificación del entorno del programa.

```
$ENV{PATH} = $ENV{PATH}:"/home/casiano/bin";
delete $ENV{DISPLAY};
system 'myls';
```

En este ejemplo el program `myls` es ejecutado en un entorno en el cual el `PATH` incluye a `/home/casiano/bin` y la variable `DISPLAY` ha sido suprimida.

3.4. Uso de comillas de ejecución

Cuando se quiere capturar la salida de un programa utilizaremos las comillas hacia atrás (*backquotes*):

```
#!/usr/local/bin/perl5.8.0 -w

my $who = shift;
my $options = shift;
my $users = `who $options`;

until ($users =~ m/$who/)
  { $users = `who $options`; }

print $users;
```

Como muestra el ejemplo, la cadena entre *backquotes* se interpola igual que una cadena de comillas dobles.

Si se usan las *backquotes* en un contexto de lista se obtiene una lista cuyos elementos son las líneas de salida el programa. Ya vimos un ejemplo en la sección 1.7.16. Lo repetimos aquí:

```
#!/usr/bin/perl -w
my @user =  `cat /etc/passwd`;
my (@name, @uid);
my $x;

for ($i=0; $i < @user; $i++) {
```

```
  ($name[$i], $x, $uid[$i]) = split ':', $user[$i];
}

@name = @name[
  sort {$uid[$a] <=> $uid[$b]} 0..$#name
];

print "@name\n";
```

3.5. Pipes

Se puede utilizar `open` para lanzar un proceso en *pipe* con el programa actual:

```
open DATE, "date|" or die "Falló la creación del pipe desde date: $!";
open MAIL, "|mail alu1324@csi.ull.es" or die "Falló el pipe hacia mail: $!";
```

La barra al final indica que la salida del proceso lanzado alimenta la entrada a nuestro programa a través del manipulador. Del mismo modo, la barra al principio indica que la entrada del proceso lanzado se alimenta de las salidas de nuestro programa hacia el correspondiente manipulador. Asi podemos escribir:

```
my $now = <DATE>;
```

para leer desde el proceso `date` y

```
print MAIL "Estimado alumno, ha obtenido usted un sobresaliente.\n";
```

Veamos un ejemplo:

```
$ cat rwho.pl
#!/usr/bin/perl -w
use strict;

my $host = shift;

my %who;

open(WHOFH, "rsh $host who |") or die "No se pudo abrir who: $!";

while (<WHOFH>) {
  print $_;
}

close(WHOFH) or die "error al cerrar: $!";
```

Al ejecutarlo obtenemos la salida:

```
$ rwho.pl nereida
lhp      :0           Apr  2 18:19
```

Al cerrar un filehandle:

```
close(MAIL);
die "mail: salida errónea, $?" if $?;
```

nuestro proceso se sincroniza con el proceso lanzado, esperando a que este termine y obtener el valor de salida, el cual queda almacenado en la variable `$?`.

3.6. La función `fork`

Cuando se llama la función `fork`, esta genera un duplicado del proceso actual. El duplicado comparte los valores actuales de todas las variables, ficheros y otras estructuras de datos. La llamada a `fork` retorna al proceso padre el identificador del proceso hijo y retorna un cero al proceso hijo.

Las siglas *PID*, del inglés *Process IDentifier*, se usan para referirse al identificador de proceso.

El hijo puede obtener el identificador del proceso padre llamando a la función `getppid`. La variable especial `$$` contiene el PID del proceso. Es de sólo lectura.

Veamos un ejemplo de uso de estas funciones:

```
#!/usr/local/bin/perl5.8.0 -w

print "PID=$$\n";

my $child = fork();
die "Falló el fork: $!" unless defined $child;

if ($child > 0) { # proceso padre
  print "Aqui proceso padre: PID=$$, hijo=$child\n";
} else { #proceso hijo
  my $ppid = getppid();
  print "Aqui proceso hijo: PID=$$, padre=$ppid\n";
}
```

La variable `$$` contiene el identificador de proceso actual. Veamos un ejemplo de ejecución:

```
> fork.pl
PID=27087
Aqui proceso padre: PID=27087, hijo=27088
Aqui proceso hijo: PID=27088, padre=1
```

Los procesos hijo pueden a su vez generar nuevos hijos, creandose así una jerarquía de procesos. dicha jerarquía recibe el nombre de *grupo de procesos*. Todos los miembros de un grupo comparten los ficheros que estaban abiertos cuando su padre los creó. En particular comparten `STDIN`, `STDOUT` y `STDERR`. Obsérvese la segunda línea de salida: el hijo no alcanza a escribir el PID del proceso padre. El padre ha finalizado primero cerrando STDOUT. Es necesario sincronizar ambos procesos. Para ello es posible usar bien la función `wait` bien la función `waitpid`:

```
$ cat -n forkw.pl
#!/usr/bin/perl -w
use strict;

print "PID=$$\n";

my $child = fork();
die "Falló el fork: $!" unless defined $child;

if ($child > 0) { # proceso padre
  print "Aqui proceso padre: PID=$$, hijo=$child\n";
  waitpid $child, 0;
} else { #proceso hijo
  my $ppid = getppid();
  print "Aqui proceso hijo: PID=$$, padre=$ppid\n";
  exit 0;
}
```

En muchos sistemas Unix, una vez que se ha realizado el `fork`, si el proceso hijo termina antes que lo haga el padre el proceso hijo no desaparece totalmente sino que queda en un estado conocido como *zombie*. El zombie permanece en la tabla de procesos con el único objeto de entregar su estatus de salida al proceso padre por si este pregunta por el usando `wait` o `waitpid`. Este proceso se denomina *reaping* (*segar*). Si el proceso padre se ramifica en un gran número de hijos y no siega es posible que la tabla de procesos se llene de procesos zombis. Por tanto, todo programa que llama a `fork` debe segar sus hijos llamando a `wait` o `waitpid`.

La función `wait` tiene el formato

```
$pid = wait()
```

La función hace que el proceso espere por la finalización de cualquier hijo y retorna el PID del proceso hijo. El proceso se bloquea hasta que exista un hijo que termine. Se puede consultar el código de terminación examinando la variable especial `$?`. Un código 0 indica una salida normal.

La función `waitpid` tiene el formato

```
$pid = waitpid($pid, $flags)
```

En este caso se espera por el hijo especificado en `$pid`. Es posible usar un argumento `$pid` igual a -1 para indicar que se espera por cualquier proceso (a la `wait`). La conducta de `waitpid` pude modificarse mediante el uso del argumento `$flags`. Hay un buen número de constantes definidas en el grupo `:sys_wait_h` del módulo estandar `POSIX`. Estas constantes se utilizan combinandolas con un `OR` binario. La más usada es `WNOHANG` la cual hace que el proceso no se bloquee (útil si se quiere hacer `polling`). La función retorna el PID del proceso o -1 si no existe ninguno disponible (cuando se usa la versión sin bloqueo). Otra constante es `WUNTRACED` la cual le indica a `waitpid` que además de los hijos que hayan terminado también recolecte los hijos parados mediante una señal de `STOP` o `TSTP`.

3.7. La función `pipe`

La función `pipe` abre un par de manejadores de fichero conectados por un pipe. Tiene el formato

```
$result = pipe(READHANDLE,WRITEHANDLE)
```

Se utiliza con `fork`. En el ejemplo el proceso padre usa el de lectura y los procesos hijos comparten el de escritura. El siguiente ejemplo esta tomado del libro de Stein [5]:

```
lhp@nereida:~/Lperl/src/perl_networking/ch2$ cat -n facfib.pl
#!/usr/bin/perl
# file: facfib.pl
# Figure 2.2: Using pipe() to create linked filehandles

use strict;
my $arg = shift || 10;

pipe(READER,WRITER) or die "Can't open pipe: $!\n";

if (fork == 0) { # first child writes to WRITER
  close READER;
  select WRITER; $| = 1;
  factorial($arg);
  exit 0;
}

if (fork == 0) { # second child writes to WRITER
  close READER;
  select WRITER; $| = 1;
```

```
  my $result = fibonacci($arg);
  exit 0;
}

# parent process closes WRITER and reads from READER
close WRITER;
print while <READER>;

sub factorial {
  my $target = shift;
  for (my $result = 1,my $i = 1; $i <= $target; $i++) {
    print "factorial($i) => ",$result *= $i,"\n";
  }
}

sub fibonacci {
  my $target = shift;
  for (my $result = 1,my $i = 1; $i <= $target; $i++) {
    print "fibonacci($i) => ",$result += $i,"\n";
  }
}
```

Al ejecutar se entrelazan las salidas:

```
lhp@nereida:~/Lperl/src/perl_networking/ch2$ ./facfib.pl 18
factorial(1) => 1
factorial(2) => 2
factorial(3) => 6
factorial(4) => 24
factorial(5) => 120
factorial(6) => 720
factorial(7) => 5040
factorial(8) => 40320
factorial(9) => 362880
factorial(10) => 3628800
factorial(11) => 39916800
factorial(12) => 479001600
factorial(13) => 6227020800
factorial(14) => 87178291200
factorial(15) => 1307674368000
fibonacci(1) => 2
factorial(16) => 20922789888000
fibonacci(2) => 4
fibonacci(3) => 7
factorial(17) => 355687428096000
fibonacci(4) => 11
fibonacci(5) => 16
factorial(18) => 6.402373705728e+15
fibonacci(6) => 22
fibonacci(7) => 29
fibonacci(8) => 37
fibonacci(9) => 46
fibonacci(10) => 56
fibonacci(11) => 67
```

```
fibonacci(12) => 79
fibonacci(13) => 92
fibonacci(14) => 106
fibonacci(15) => 121
fibonacci(16) => 137
fibonacci(17) => 154
fibonacci(18) => 172
lhp@nereida:~/Lperl/src/perl_networking/ch2$
```

3.8. La Depuración de Procesos

La ejecución con el depurador de procesos generados mediante `fork` sólo es posible si se usa una `xterm`. Cuando un programa que usa `fork` se ejecuta bajo el depurador es muy posible que se produzca un error debido a la terminal que se abre para cada proceso hijo. El depurador sólo da soporte a `xterm` y a una consola bajo `OS/2`. Observe el mensaje de error que resulta al ejecutar el programa presentado en la sección anterior:

```
lhp@nereida:~/Lperl/src/perl_networking/ch2$ perl -d ./facfib.pl 18
Loading DB routines from perl5db.pl version 1.25
Editor support available.
Enter h or `h h' for help, or `man perldebug' for more help.
main::(./facfib.pl:6):  my $arg = shift || 10;
  DB<1> /fork
10:     if (fork == 0) { # first child writes to WRITER
  DB<2> b 10
  DB<3> L b
./facfib.pl:
 10:    if (fork == 0) { # first child writes to WRITER
   break if (1)
  DB<4> c
main::(./facfib.pl:10): if (fork == 0) { # first child writes to WRITER
  DB<4> n
main::(./facfib.pl:17): if (fork == 0) { # second child writes to WRITER
######### Forked, but do not know how to create a new TTY. #########
  Since two debuggers fight for the same TTY, input is severely entangled.

  I know how to switch the output to a different window in xterms
  and OS/2 consoles only.  For a manual switch, put the name of the created TTY
  in $DB::fork_TTY, or define a function DB::get_fork_TTY() returning this.

  On UNIX-like systems one can get the name of a TTY for the given window
  by typing tty, and disconnect the shell from TTY by sleep 1000000.

main::(./facfib.pl:11):   close READER;
```

En la línea 1 buscamos por la primera aparición de la palabra `fork` en el código. En la línea 2 ponemos un punto de `break` en la línea 10 del fuente. Depués (`DB<3>`) listamos los puntos de ruptura. Al dar la orden de continuar (línea 4) obtenemos el mensaje de error.

Escriba `xterm &` y llame al depurador desde la misma. Ahora será posible depurar los procesos hijo, ya que se abrirá una ventana por proceso. En la jerga del depurador estos procesos-ventana se denomina `daughters`. Es interesante hacer notar que aunque los comandos `p` y `x` usan el manipulador de ficheros `DB::OUT` el cual esta vinculado a la ventana hija, el comando `print` lo sigue haciendo en `STDOUT`, el cual esta asociado con la ventana `xterm` del proceso padre. Deberá terminar cada sesión de depuración manualmente. Puede evitarlo, si lo prefiere, estableciendo la opción `inhibit_exit` a cero:

```
DB<10> o inhibit_exit=0
        inhibit_exit = '0'
```

3.9. Práctica: Un Cálculo Sencillo con Procesos

El área bajo la curva $y = \frac{1}{1+x^2}$ entre 0 y 1 nos proporciona un método para calcular π:

$$\int_0^1 \frac{4}{(1+x^2)}dx = 4\arctan(x)|_0^1 = 4(\frac{\pi}{4} - 0) = \pi$$

Escriba un programa Perl que calcule π usando varios procesos cada uno de los cuales calcula una parte de la suma asociada con la integral anterior. Puede imitar el ejemplo introducido en la sección 3.7. Haga uso del soporte que el depurador da para procesos (véase sección 3.8).

3.10. Señales

Una señal es un mensaje enviado a nuestro programa, habitualmente desde el sistema operativo, indicando que se ha producido una situación excepcional. Puede indicar un error en el programa, por ejemplo una división por cero o un evento como la pulsación por el usuario de la tecla de interrupción (normalmente `CTRL-C`) o la terminación de un proceso lanzado por nuestro programa.

3.10.1. Envío de señales

La función `kill` permite el envío de señales a otros procesos. Su modo de uso es:

```
$count = kill($signal, @processes);
```

que envía la señal `$signal` a los procesos cuyos PID están en la lista `@processes`. El resultado devuelve el número de procesos a los que la señal llegó con éxito. Por supuesto, el proceso que emite la señal debe tener los privilegios suficientes para hacerlo.

Si se usa la señal especial `0`, la función `kill` devuelve el número de procesos que serán señalados, sin que la señal sea realmente entregada. Si se emplea un número negativo, se entregará la señal a todos los procesos con identificador de grupo igual al opuesto de ese número.

3.10.2. Captura de señales

El hash `%SIG` contiene las rutinas de manipulación de las señales. las claves del hash son los nombres de las señales. Así, `$SIG{INT}$` contendrá el nombre de la subrutina que se ejecuta cuando se produce una interrupciónd e teclado (el usuario pulsó `CTRL-C`).

Supongamos que tenemos un programa que crea ciertos ficheros temporales durante su ejecución, los cuales son borrados al final de la misma. Si el usuario pulsa `CTRL-C`, nuestro programa será interrumpido y los ficheros temporales quedarán como restos de la ejecución inacabada. El problema se resuelve usando un manejador de la señal de interrupción:

```
> cat signal.pl
#!/usr/bin/perl -w

use strict;

my $temp_dir = "/tmp/signal.$$";
mkdir $temp_dir, 0700 or die "No se pudo crear $temp_dir: $!";

sub clean_files {
  unlink glob "$temp_dir/*";
```

```
  rmdir $temp_dir;
}

sub int_handler {
  &clean_files;
  die "\nPrograma interrumpido por el usuario\n";
}

$SIG{INT} = 'int_handler';
my $fn = "$temp_dir/file.txt";
open(F, ">$fn");

my $n = shift or 1;
my @files;
my $i = 0;
while ($i < $n) {
  print F "$i\n"; $i++;
  @files = </tmp/* $temp_dir/*>;

  $" = "\n";
  print STDOUT "@files";
}
close(F);

&clean_files;
```

Veamos un ejemplo de ejecución:

```
> signal.pl 100000
/tmp/Acromj4rS5
/tmp/gv_406924cc_1_contratacion.pdf.tmp
/tmp/kde-pp2
/tmp/ksocket-pp2
/tmp/mcop-pp2
/tmp/pico.000835.bak
/tmp/session_mm_apache0.sem
/tmp/signal.7247
/tmp/ssh-XXugMPij
/tmp/v562119
...
<CTRL-C>
Programa interrumpido por el usuario
> ls -ltr /tmp
total 32
drwx------    2 pp2      pp2          4096 Mar 29 08:07 ssh-XXugMPij
drwx------    3 pp2      pp2          4096 Mar 29 08:08 mcop-pp2
drwx------    2 pp2      pp2          4096 Mar 29 08:08 ksocket-pp2
drwx------    2 pp2      pp2          4096 Mar 29 08:11 kde-pp2
drwx------    2 casiano  casiano      4096 Mar 29 13:12 v562291
-rw-------    1 root     root            0 Mar 30 06:29 session_mm_apache0.sem
-rw-r--r--    1 casiano  casiano       705 Mar 30 08:42 gv_406924cc_1_contratacion.pdf.tmp
drwx------    2 casiano  casiano      4096 Mar 30 08:47 v562119
-rw-r--r--    1 casiano  casiano         0 Mar 30 10:00 Acromj4rS5
-rw-------    1 casiano  casiano      2397 Mar 30 11:15 pico.000835.bak
```

3.10.3. Controlando errores en tiempo de ejecución con `eval`

La función `eval` recibe habitualmente una cadena como argumento, evaluándola como si de un programa Perl se tratara. Sin embargo, `eval` admite una segunda forma en la puede ser utilizado para manejar errores en tiempo de ejecución. en esta segunda forma recibe como argumento un bloque de código.

En el siguiente ejemplo atrapamos una excepción de división por cero:

```
> cat eval_excep.pl
#!/usr/bin/perl -w

use strict;

eval {
  my ($a, $b) = (10, 0);
  my $c = $a / $b;
};

print "Ocurrió un error: $@" if $@; # mensaje de error
```

Veamos el resultado de la ejecución:

```
> eval_excep.pl
Ocurrió un error: Illegal division by zero at eval_excep.pl line 7.
```

Para señalar los errores en nuestros programas usaremos `die`. Cuando `die` es ejecutado dentro de un `eval` el intérprete Perl deja la cadena de error en la variable `$@` y sale del bloque. Veamos un ejemplo:

```
> cat eval_die.pl
#!/usr/bin/perl -w

use strict;

sub open_file {
  my $fn = shift;
  open (F, $fn) or die("No se pudo abrir el fichero $fn: $!\n");
}

my $fn = shift;
while (1) {
  eval { open_file($fn); };
  if ($@) { print "En main: $@\n"; }
  else { last; }
  print "Entre el nombre del fichero: ";
  chomp ($fn = <STDIN>);
}

while (<F>) { print; }
```

Veamos el resultado de una ejecución:

```
> eval_die.pl xx.pl
En main: No se pudo abrir el fichero xx.pl: No such file or directory

Entre el nombre del fichero: one.dat
```

```
1
2
3
0
4
0
```

3.10.4. Controlando warnings en tiempo de ejecución

Para controlar los mensajes de *warning* basta con instalar un *manejador de warnings* en `$SIG{__WARN__}`. Todos los mensajes serán redirigidos al manejador y es responsabilidad de este mostrarlos. Así pues, el uso de un manejador `$SIG{__WARN__}` nos permite silenciar selectivamente warnings. Veamos un ejemplo:

```
$ cat -n warnings.pl
#!/usr/bin/perl -w
use strict;

my $ErrStr = "";
my $warnflag = 1;

sub warnhandler {
  $ErrStr .= $_[0] if $warnflag
}

$SIG{'__WARN__'} = \&warnhandler;

warn "Cuidado: Mira detrás de ti!\n";
$warnflag = 0;
warn "Alguien está vigilando!\n";

print "La cadena ErrStr contiene:\n$ErrStr\n";
```

Al ejecutarlo se obtiene la siguiente salida:

```
$ ./warnings.pl
La cadena ErrStr contiene:
Cuidado: Mira detrás de ti!
```

3.10.5. Poniendo límites de tiempo con `eval`

La función `alarm` establece un cronómetro. Cuando la cuenta atrás termina el sistema operativo emite una señal `ALRM`, la cual puede ser interceptada por un manejador de señales. Con un argumento `0` desactiva el cronómetro. Este primer ejemplo pide al usuario un password, controlando si el tiempo de lectura ha excedido un cierto límite:

```
$ cat time_exceeded.pl
#!/usr/bin/perl -w
use strict;

my $deadline = (shift || 3);
my $time_exceeded = 0;
sub set_time_exceeded { $time_exceeded = 1; }
$SIG{ALRM} = "set_time_exceeded";
print "Teclee su password: ";
```

```
alarm($deadline);
my $passwd = <STDIN>;
alarm(0);
print STDERR "Tiempo excedido.\n" if $time_exceeded;
```

El siguiente ejemplo utiliza `eval` junto con las señales para establecer un límite al tiempo de espera en una lectura:

```
> cat eval_alarm.pl
#!/usr/bin/perl -w

use strict;

sub time_out {
  die "Cansados de esperar";
}

my $lim = (shift or 3);
my $buf;

$SIG{ALRM} = "time_out";
eval {
  alarm($lim); # Le indica al SO enviar una señal de ALRM cada $lim s.
  $buf = <>;
  alarm(0); # Cancela la alarma
};
if ($@ =~ /Cansados de esperar/) {
  print "Se acabo el tiempo para la entrada. <Adios muy buenas!\n"
} else {
  print $buf;
}
```

Nótese que si la alarma se activa, `$@` contiene algo parecido a `"Cansados de esperar at eval_alarm.pl line 6"`, asi que no se debe usar `eq` sino una expresión regular.

Ejemplo de ejecución:

```
> eval_alarm.pl 2
Se acabo el tiempo para la entrada. <Adios muy buenas!
> eval_alarm.pl 4
hola
hola
```

En el siguiente ejemplo utilizamos esta estrategia para saber sobre la conectividad de una máquina. El comando `ping` nos permite conocer los tiempos de respuesta. La opción `-c 1` limita el número de paquetes de prueba a uno. Veamos un ejemplo de uso del comando `ping`:

```
$ ping -c 1  etsii
PING etsii (193.145.101.10): 56 data bytes
64 bytes from 193.145.101.10: icmp_seq=0 ttl=63 time=0.7 ms

--- etsii ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.7/0.7/0.7 ms
```

En este otro ejemplo vemos como es una respuesta negativa:

```
$ ping -c 1  miranda
PING miranda (193.145.105.176): 56 data bytes

--- miranda ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
```

El comando `ping` en ocasiones no termina. En el siguiente ejemplo se comprueba la conectividad de la máquina a través del mismo, limitando el tiempo de espera por `ping`.

```
$ cat -n myping.pl
#!/usr/bin/perl -w

use strict;

my @machines = @ARGV;
my ($m, $code) = ("", "");

$SIG{ALRM} = 'kill_it';
for $m (@machines) {
  eval {
    alarm(3);
    $code = `ping -c 1 $m`;
    alarm(0);
  };
  if (defined($@) and ($@ =~ /too long/)) {
    print "El acceso a $m toma demasiado tiempo.\n";
    my @ping = `ps -fA`;
    my @pid = map { /(\d+)\s+$$\b.*\bping -c 1 $m\b/; $1 }  @ping;
    @pid = grep { defined($_) } @pid;
    system('ps -fA | grep ping');
    print "Ejecutamos:\nkill 9, @pid\n" if @pid;
    kill 9, @pid if @pid;
    system('ps -fA | grep ping');
  }
  else {
    print "From $m:\ncode = $code\n\n";
  }
}

sub kill_it { die "too long"; }
$ ./myping.pl miranda # máquin apagada

lhp@nereida:~/Lperl/src$ ./myping.pl miranda # máquina apagada
El acceso a miranda toma demasiado tiempo.
lhp       6058  1899  0 16:16 pts/15   00:00:00 /usr/bin/perl -w ./myping.pl miranda
lhp       6059  6058  0 16:16 pts/15   00:00:00 ping -c 1 miranda
lhp       6061  6058  0 16:16 pts/15   00:00:00 sh -c ps -fA | grep ping
lhp       6063  6061  0 16:16 pts/15   00:00:00 grep ping
Ejecutamos:
kill 9, 6059
lhp       6058  1899  0 16:16 pts/15   00:00:00 /usr/bin/perl -w ./myping.pl miranda
lhp       6059  6058  0 16:16 pts/15   00:00:00 [ping] <defunct>
lhp       6064  6058  0 16:16 pts/15   00:00:00 sh -c ps -fA | grep ping
lhp       6066  6064  0 16:16 pts/15   00:00:00 grep ping
```

```
$ ./myping.pl 193.145.105.253 # encendida
From 193.145.105.253:
code = PING 193.145.105.253 (193.145.105.253): 56 data bytes
64 bytes from 193.145.105.253: icmp_seq=0 ttl=64 time=2.0 ms

--- 193.145.105.253 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 2.0/2.0/2.0 ms
```

3.11. Práctica: Gestor de Colas

Esta práctica es mas complicada que las anteriores. Escriba un programa que recibe como entrada un fichero conteniendo direcciones IP o nombres de máquinas `machines.dat` y un fichero conteniendo descripciones de tareas `tasks.dat`. El programa lanzará las tareas descritas en `tasks.dat` en las máquinas enumeradas en `machines.dat` que estén operativas. Se asume que está habilitado el comando `rsh` y que las máquinas comparten el sistema de archivos (están en NFS).

Desarrolle el programa de manera estructurada. Active el pragma `use strict` para evitar errores. Comienze cubriendo progresivamente los siguientes puntos:

1. El programa deberá comprobar que máquinas están operativas haciendo uso del comando `ping` con la opción `-c 1` para limitar el número de paquetes de prueba a uno. El comando `ping` en ocasiones no termina. Repase las estrategias usadas en el capítulo para finalizar la espera por el comando despues de un número de segundos especificado. Cuando no se sobrepasa el tiempo límite, puede hacer uso de expresiones regulares para detectar que la máquina está accesible. Cuando se sobrepasa, la rutina manejadora puede abortar el proceso.

2. Si la máquina esta accesible, compruebe que tiene el intérprete Perl instalado, haciendo uso de `rsh` y del comando `which`.

3. El fichero de tareas es un fichero de texto que contiene una serie de registros describiendo una tarea. Cada registro empieza por una línea con la palabra `TASK` y un nombre lógico que es el nombre de la tarea. Cada tarea se separa de la siguiente por una línea en blanco. Cada campo del registro es una línea que comienza por el nombre del campo, dos puntos (`:`) y el valor del campo. Los campos permitidos son:

   a) `DIRECTORY`. Contiene el directorio en la máquina remota de trabajo de la tarea. El valor es una cadena entre comillas simples.

   b) El campo `ENV` que contiene la lista de parejas variable de entorno y su valor, separadas por blancos. Estas variables de entorno se deberán modificar antes de la ejecución de la tarea. Las variables y los valores son cadenas entre comillas simples.

   c) `STDOUT` que contiene el nombre del fichero al que se desea redirigir la entrada estándar. Análogamente habrán campos `STDIN` y `STDERR`. Si `STDIN` no existe se redefine a `'/dev/null'`. Elija valores por defecto para `STDOUT` y `STDERR`. El valor de estos campos es una cadena entre comillas simples.

   d) El campo `MACHINES` si existe es una expresión regular describiendo las máquinas en las que esta tarea puede ejecutarse. Cada nombre de máquina se delimita entre comillas simples.

   e) El campo `COMMAND` conteniendo el nombre del comando a ejecutar. El valor es una cadena entre comillas simples.

   f) `PARAMETERS` que contiene la lista de llamadas al programa. Cada llamada se separa de la anterior mediante un punto y coma. Cada parámetro va entre comillas simples.

Todos los campos son opcionales salvo el campo `COMMAND`. En una primera fase, asuma que el campo `COMMAND` es el único campo que existe en el fichero de tareas. Vaya añadiendo campos posteriormente.

Sigue un ejemplo de fichero de tareas:

```
$ cat tasks.dat
TASK listar
ENV:'PATH /bin:/usr/bin'
COMMAND:'ls'
DIRECTORY:'~'
STDOUT: 'listar.out.$machine'
PARAMETERS: '-l';'-s' '-t'

TASK ordenar
COMMAND: 'sort'
DIRECTORY: '~/lhp/procesos/practica/'
STDOUT: 'ordenado.out.$machine'
STDIN: 'aordenar'
MACHINES: 'manis' 'timple'
```

y un ejemplo de fichero de máquinas:

```
$ cat machines.dat
timple
millo
manis
fonil
```

3.11.1. Sugerencias

- Para preparar la ejecución del comando construya dinámicamente un guión Perl con todos los requisitos especificados en el registro de tareas y ejecute el guión asi creado en una de las máquinas disponibles usando `rsh`. Utilice `fork` para crear un proceso hijo el cual será el que realmente lance a `rsh`.
- Además de repasar los capítulos 2 y 3 quizá debería estudiar el manejo de paréntesis con memoria en expresiones regulares, cuyo uso se explica en el capítulo 4. La idea es sencilla: Después de un emparejamiento, la cadena que casó con la primera subexpresión parentizada de la expresión regular se guarda en `$1`, la que caso con la segunda subexpresión parentizada se guarda en `$2`, etc. Por ejemplo:

  ```
  1 if ($line =~ /^DIRECTORY:\s*'([^']*)'/) {
  2   my $directory = $1; ... }
  ```

  Después de la ejecución de la línea 1, si hubo emparejamiento, `$1` contiene la cadena que casó con `[^']*`. Así, si `$line` contiene

  `DIRECTORY: '/home/casiano/tmp'`

  la variable `$1` contendrá la cadena

  `/home/casiano/tmp`.
- Para construir el *script* te puede ser muy útil usar un *here document* o *documento aqui*. Veamos un ejemplo de uso de "documento aqui":

  ```
  print <<EOI
  El programa se deberá ejecutar con:

  myprog numfiles directory initialvalue

  suponemos que hay $x valores y que tenemos
  $y blancos.
  EOI
  ```

Esto es, para definir un "documento aqui" se escribe la etiqueta, en este caso `EOI` pero en general una cadena arbitraria, 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 (esto es, se sustituyen las variables). Veamos otro ejemplo:

```
$ cat here_document.pl
#!/usr/bin/perl -w

use strict;

my $b =4;
my $a = <<EOT;
esto es una
prueba.

b = $b
EOT

print $a;
$ ./here_document.pl
esto es una
prueba.

b = 4
```

- ¡Tenga en cuenta que el guión o *script* generado no debe contener errores!. Durante el desarrollo, vigile la sintáxis del guión generado.

Capítulo 4

Expresiones Regulares

4.1. Un ejemplo sencillo

Reescriba el siguiente ejemplo para que sea una función de conversión de temperaturas. Utilize la estrategia del `hash` para tener parámetros con nombres.

```
#!/usr/bin/perl -w
print "Enter a temperature (i.e. 32F, 100C):\n";
$input = <STDIN>;
chop($input);

if ($input !~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i) {
  print "Expecting a temperature, so don't understand \"$input\".\n";
}
else {
  $InputNum = $1;
  $type = $3;
  if ($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;
}

```

4.2. 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.

4.3. 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

Ejemplo:

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

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

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:

```
/Version: (.*)|Revision: (.*)/ && ($rev = $+);
```

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:

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

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:

```
DB<3> if ($z =~ m{a(\d+)(\.(\d+))?}) { print "@+\n"; }
9 6 9 9
```

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

```
DB<4> if ($z =~ m{a(\d+)(\.(\d+))?}) { print "$#+\n"; }
3
DB<5> if ($z =~ m{(a)(\d+)(\.(\d+))?}) { print "$#+\n"; }
4
```

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
```

Para saber más sobre las variables especiales disponibles consulte `perldoc perlvar`.

4.4. El uso de $1 dentro una expresión regular

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:

```
> cat dollar1slash1.pl
#!/usr/bin/perl

 $a = "hola juanito";
 $b = "adios anita";

 $a =~ /(ani)/;
 $b =~ s/(adios) *$1/\1 boni$1/;
 print "$b\n";
> dollar1slash1.pl
adios boniadiosta
```

Observe como el `$1` que aparece en la cadena de reemplazo se refiere a la cadena que caso en este último casamiento (`adios`).

4.5. 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`.

4.6. Expresiones regulares abreviadas

He aqui algunas de las abreviaturas usadas para algunas de las clases mas comunes:

Código	Significado
`\d`	`[0-9]`
`\D`	`[^0-9]`
`\w`	`[a-zA-Z0-9_]`
`\W`	`[^a-zA-Z0-9_]`
`\s`	`[ \t\n\r\f]`
`\S`	`[^ \t\n\r\f]`

4.7. Listas y ExpReg

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 `(1)`.

```
#!/usr/bin/perl -w
$foo = "one two three four five";
($F1, $F2, $Etc) = ($foo =~ /^\s*(\S+)\s+(\S+)\s*(.*)/);
print "List Context: F1 = $F1, F2 = $F2, Etc = $Etc\n";
# This is equivalent to:
($F1, $F2, $Etc) = split(' ',$foo,3);
print "Split: F1 = $F1, F2 = $F2, Etc = $Etc\n";
```

Observa el resultado de la ejecución:

```
> escapes.pl
List Context: F1 = one, F2 = two, Etc = three four five
Split: F1 = one, F2 = two, Etc = three four five
```

4.8. `Map` y las expresiones regulares

El siguiente ejemplo produce una lista formada por las direcciones electrónicas de una carpeta de correo:

```
#!/usr/bin/perl -w
open MAILFILE, shift;
@from = map /^From:\s+(.*)$/, <MAILFILE>;
map { print "$_\n" } @from;
```

Observése el uso de `map` aplicando la expresión regular `/^From:\s+(.*)$/` a cada una de las líneas del fichero de entrada. Las cadenas que se emparejan con el primer paréntesis son devueltas para formar la lista que se almacena en `@from`. El segundo uso de `map` imprime cada uno de los elementos de la lista `@from`. Al ejecutarlo se produce una salida como esta:

```
~/perl/src> from.pl ~/mail/euromicro
Mail System Internal Data <MAILER-DAEMON@nereida.deioc.ull.es>
pdp03 Conference Manager <pdp03cm@iris.ima.ge.cnr.it>
pdp03cm@samba.ima.ge.cnr.it
"Chiquita Snippe-Marlisa" <euromicro@standby.nl>
"C.Leon [TRACS]" <coromoto@epcc.ed.ac.uk>
Javier Miranda <jmiranda@iuma.ulpgc.es>
Javier Miranda <jmiranda@iuma.ulpgc.es>
```

4.9. 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

4.10. La opción /m

Em el modo `m` o multilínea (`^` y `$` casan con los `\n` internos) pero el punto no casa con los retornos de carro. Véase el 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
```

4.11. La opción /s

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

4.12. 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
```

4.13. La opción /x

La opción `/x` 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.

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

```
#!/usr/bin/perl -w
$progname = shift @ARGV;
open(PROGRAM,"<$progname") || die "can't find $progname";
undef($/);
$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:

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

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

4.14. Interpolación en los patrones

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`.

```
#!/usr/bin/perl -w
my $what = shift @ARGV;
while (<>) {
  if (/$what/o) { # compile only once
    print ;
  }
}
```

Sigue un ejemplo de ejecución:

```
> mygrep.pl '\bif\b' *.pl
if ($input !~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i) {
  if ($type eq "C" or $type eq "c") {
  push(@fields, undef) if $text =~ m/,$/;
  next if !s{
  next if !s/\b([a-z]+)((\s|<[^>]+>)+)(\1\b)/\e[7m$1\e[m$2\e[7m$4\e[m/ig;
  if (/$what/) {
```

4.15. RegExp no "Greedy"

Las expresiones 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}?`
- `*?`
- `+?`
- `??`

El siguiente ejemplo muestra las líneas de los ficheros `*.html` que contienen dos "anclas" en la misma línea. Observe como el uso de los operadores no "Greedy" nos permite, por ejemplo, evitar el uso de la negación de una clase:

```
> perl -e 'print "$ARGV\n\t$_" if (/<a href=\"(.*?)\">(.*?)<\/a>.*?<a href/i)' \
                                                           -n *.html
plindex.html
        <li><a href="http://osr5doc.sco.com:1996/tools/CONTENTS.html">
        Programming Tools Guide</a> (<a href="http://osr5doc.sco.com:19
        96/tools/Lex.html">Chapter 12: LEX</a>) (<a href="http://osr5do
        c.sco.com:1996/tools/Yacc.html">Chapter 13: yacc </a>)
plpracticas.html
        <li><a href="http://nereida.deioc.ull.es/html/java.html#html">
        Varios links sobre HTML</a> (<a href="http://www.lcc.uma.es/~
        eat/services/html-js/manu.html" target="_blank">Tutorial</a>)
        (<a href="http://nereida.deioc.ull.es/html/barebone.html" tar
        get="_blank">Guía de Referencia</a>)
```

4.16. Negaciones y operadores no *greedy*

No siempre se puede usar un operador *greedy* en sustitución de una clase negada. En este ejemplo se intentan detectar las cadenas entre comillas dobles que terminan en el signo de exclamación:

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

 $b = $a = 'Ella dijo "Ana" y yo contesté: "Jamás!". Eso fué todo.';
 $a =~ s/".*?!"/-$&-/;
 print "$a\n";
 $b =~ s/"[^"]*!"/-$&-/;
 print "$b\n";

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

4.17. Algunas extensiones

4.17.1. Comentarios

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

4.17.2. Paréntesis de agrupamiento

`(?: ...)` 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
```

4.17.3. Operador de predicción positivo

`(?= ...)` 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 lookahead.pl
#!/usr/bin/perl

 $a = "bugs the rabbit";
 $b = "bugs the frog";
 if ($a =~ m{bugs(?= the cat| the rabbit)}i) { print "$a matches. \$& = $&\n"; }
 else { print "$a does not match\n"; }
 if ($b =~ m{bugs(?= the cat| the rabbit)}i) { print "$b matches. \$& = $&\n"; }
 else { print "$b does not match\n"; }
> lookahead.pl
bugs the rabbit matches. $& = bugs
bugs the frog does not match
>
```

Compare la siguiente salida:

```
> perl -e 'print "$ARGV\n\t$&\n"
         if (m{<a href=\"(.*?)\">(?=(.*?)</a>.*?<a href)}i)'
                                      -wn ~/public_html/*.html
/home/pl/public_html/plindex.html
        <a href="http://osr5doc.sco.com:1996/tools/CONTENTS.html">
/home/pl/public_html/plpracticas.html
        <a href="http://nereida.deioc.ull.es/html/java.html#html">
```

con la obtenida en la sección 4.15.

4.17.4. Operador de predicción negativo

`(?! ...)` Operador de "trailing" o "mirar-adelante" negativo.

Por ejemplo `/foo(?!bar)/` contiene a cualquier ocurrencia de `foo` que no vaya seguida de `bar`. Aparentemente el operador "mirar-adelante" negativo es parecido a usar el operador "mirar-adelante" positivo con la negación de una clase. 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.

Otros dos ejemplos:

- `^(?![A-Z]*$)[a-zA-Z]*$` casa con líneas formadas por secuencias de letras tales que no todas son mayúsculas.
- `^(?=.*?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`.

Nótese que el "mirar-adelante" negativo es diferente del "mirar-atrás". No puede usar este operador para "mirar-atrás": `(?!foo)bar/` no casa con una aparición de `bar` que no ha sido precedida de `foo`. 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`.

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

 my $a = shift;
```

```
 if ($a =~ m{(?!foo)bar}i) { print "$a casa la primera. \$& = $&\n"; }
 else { print "$a no casa la primera\n"; }

 if ($a =~ m{(?!foo)...bar}i) { print "$a casa la segunda. \$& = $&\n"; }
 else { print "$a no casa la segunda\n"; }

> foobar.pl foo
foo no casa la primera
foo no casa la segunda
> foobar.pl foobar
foobar casa la primera. $& = bar
foobar no casa la segunda
```

Si quisieramos conseguir algo parecido tendríamos que escribir algo asi como `/(?!foo)...bar/` que casa con una cadena de tres caracteres que no sea `foo` seguida de bar. En realidad, es mucho mas fácil escribir:

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

Veamos otro ejemplo:

```
#!/usr/bin/perl -w
$s = "foobar";
if ($s =~ /(?!foo)bar/) {
  print "$s matches (?!foo)bar\n";
}
if ($s !~ /(?!foo)...bar/) {
  print "$s does not match (?!foo)...bar\n";
}
if ($s =~ /bar/ and $` !~ /foo$/) {
  print "$s matches /bar/ and \$` !~ /foo\$/\n";
}
else {
  print "$s does not match /bar/ and \$` !~ /foo\$/\n";
}
```

Los resultados de la ejecución de este ejemplo son:

```
> lookbehind.pl
foobar matches (?!foo)bar
foobar does not match (?!foo)...bar
foobar does not match /bar/ and $` !~ /foo$/
```

4.18. Secuencias de números de tamaño fijo

El siguiente problema y sus soluciones se describen en el libro de J.E.F. Friedl [4]. 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. 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":

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

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. Encontrará así números como el último resaltado:

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.

4.19. 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 en la sección 4.18:

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

4.20. 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.

Esta es la solución:

```
#!/usr/bin/perl -w
# one <a>one</a>
# is two three
# three
$/ = ".\n";
while (<>) {
  next if !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
  }
  "\e[7m$1\e[m$2\e[7m$4\e[m"igx;
  s/^([^\e]*\n)+//mg;  # remove lines that don't contain escapes
  s/^/$ARGV: /mg;      # insert filename at the beginning of the lines
  print;
}
```

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.

Sigue un ejemplo de uso:

```
> cat double.in
one <a><b>one</b></a>
is two three


three
.

xxxx
>
> doublee.pl double.in
double.in: one <a><b>one</b></a>
double.in: is two three
double.in: three
```

4.21. 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

  while ($text =~
    m/"(([^"\\]|\\.)*)",? # quoted fields
      |
      ([^,]+),?           # $3 = non quoted fields
      |
      ,                   # allows empty fields
    /gx
    )
  {
    push(@fields, defined($1)? $1:$3); # add the just matched field
  }
  push(@fields, undef) if $text =~ m/,$/; #account for an empty last field
  return @fields;
}

$test = '"earth",1,"a1, a2","moon",9.374';
print "string = \'$test\'\n";
print "Using parse_csv\n:";
@fields = parse_csv($test);
foreach $i (@fields) {
  print "$i\n";
}

print "Using Text::ParseWords\n:";
#  @words = &quotewords($delim, $keep, @lines);
#The $keep argument is a boolean flag.  If true, then the
#tokens are split on the specified delimiter, but all other
#characters (quotes, backslashes, etc.) are kept in the
#tokens.  If $keep is false then the &*quotewords()
#functions remove all quotes and backslashes that are not
#themselves backslash-escaped or inside of single quotes
#(i.e., &quotewords() tries to interpret these characters
#just like the Bourne shell).

@fields = quotewords(',',0,$test);
foreach $i (@fields) {
  print "$i\n";
}
```

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
```

4.22. 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
```

4.23. 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
```

4.24. 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
```

4.25. Expandiendo y comprimiendo tabs

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

```
#!/usr/bin/perl -w
undef($/);
$string = <>;
$width=2;
while ($string =~ s/\t+/' 'x(length($&)*$width-length($`)%$width)/e) {
}
print "$string";
```

Supuesto que los tabs se paran cada `tpw = 8` posiciones se convierte a espacios calculando el correspondiente múltiplo de `$tpw` a partir de la posición actual. La función `length` devuelve la longitud en caracteres de la expresión que se le pasa como argumento. Si se omite la expresión usará la variable `$_`.

Sigue un ejemplo de ejecución:

```
> cat -t tabs.in
one^Itwo^I^Ithrei^I^I^I
four^I^I^I^Ifive^I^I^I^I^I
end
> tabs.pl tabs.in | cat -t
one     two             threi
four                           five
end
```

4.26. Modificación en múltiples ficheros

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`.

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 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] 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/;
```

4.27. 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
```

4.28. 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
```

Capítulo 5

Referencias

5.1. Referencias a variables ya existentes

5.1.1. Referencias y referentes

Perl proporciona un tipo especial de escalar denominado *referencia*. Para crear una referencia existe el operador unario \ el cual toma una variable o valor y retorna una referencia al mismo. La variable original es conocida como el *referente* al que la referencia se refiere.

```
$ra = \$a;  # referencia a escalar
$rb = \@b;  # referencia a arreglo
$rc = \%c;  # referencia a hash
$rf = \&f;  # referencia a subrutina
$rx = \$rb; # referencia a referencia
```

Una vez que se tiene una referencia, podemos volver al original prefijando la referencia (opcionalmente entre llaves) con el símbolo apropiado:

```
${$ra}  # es  el referente de $ra, el valor de $a
@{$rb}  # es  el referente de $rb, el valor de @a
@{$ra}  # es un error porque $ra apunta a un escalar
%{$rc}  # es  el referente de $rc, el valor de %c
&{$rf}(2,5)  # llamada a la función referida por $rf
```

Los elementos de un array o de un hash se referencian como cabría esperar:

```
$$ra[1] # segundo elemento del array @a
$$rc{key1} # elemento del hash %c con clave key1
```

5.1.2. Referencias a constantes

Se pueden crear referencias a constantes:

```
$rc = \10;
$rs = \"hello";
```

Cuando se trata de referencias a constantes el valor referenciado no se puede modificar. Véase la siguiente sesión el depurador:

```
DB<1> $ra = \10
DB<2> p $$ra
10
DB<3> p $ra
SCALAR(0x8109530)
DB<4> $$ra = 20
```

```
Modification of a read-only value attempted at (eval 10)[/usr/lib/perl5/5.6.1/perl5db.pl:1521] line 2.
DB<5> @b = 5..10
DB<6> $rb = \@b
DB<7> print "rb = $rb; rb-> = @{$rb}"
rb = ARRAY(0x8109548); rb-> = 5 6 7 8 9 10
```

Observe la forma en la que se imprime una referencia.

5.1.3. Contextos y referencias

Un aviso sobre los peligros de Perl. Observe la siguiente ejecución con el depurador:

```
DB<1> $ra = \('a', 'b', 'c')
DB<2> p $ra
SCALAR(0x81095a8)
DB<3> p $$ra
c
```

¿Que está pasando? ¿Porqué `$ra` no es una referencia a un array sino a un escalar? Observe que la parte izquierda de la asignación es `$ra`, lo cual establece un contexto escalar.

Compare el resultado anterior con este otro:

```
DB<1> $a = 4; $b = 5; $c = 6
DB<2> @a = \($a, $b, $c)
DB<3> p @a
SCALAR(0x81046d0)SCALAR(0x81046f4)SCALAR(0x81046ac)
DB<4> p \$a, \$b, \$c
SCALAR(0x81046d0)SCALAR(0x81046f4)SCALAR(0x81046ac)
```

en un contexto de lista el operador `\` actúa sobre cada uno de los elementos de la lista.

En el siguiente ejemplo observamos la acción del operador `\` en una posible situación ambigua:

```
DB<1> @a = (\(1, 0, 0), \(0, 1, 0), \(0, 0, 1))
DB<2> p @{$a[1]}
Not an ARRAY reference at (eval 17)[/usr/share/perl/5.6.1/perl5db.pl:1521] line 2.
```

No se trata por tanto que produzca referencias a cada uno de los vectores. En realidad `@a` contiene un vector de referencias a los elementos individuales:

```
DB<3> x @a
0  SCALAR(0x8450d74) -> 1
1  SCALAR(0x81046f4) -> 0
2  SCALAR(0x8450dc8) -> 0
3  SCALAR(0x8461ccc) -> 0
4  SCALAR(0x846cd28) -> 1
5  SCALAR(0x8461c9c) -> 0
6  SCALAR(0x8461a68) -> 0
7  SCALAR(0x84619d8) -> 0
8  SCALAR(0x84619cc) -> 1
```

Las subexpresiones como estas no se evalúan en un contexto escalar y se consideran sublistas y por tanto se aplanan. Vea el ejemplo:

```
DB<1> @a = ((1, 0, 0), (0, 1, 0), (0, 0, 1))
DB<2> x @a
0  1
1  0
2  0
```

```
3  0
4  1
5  0
6  0
7  0
8  1
DB<3> @a = (scalar(1, 0, 0), scalar(0, 1, 0), scalar(0, 0, 1))
DB<4> x @a
0  0
1  0
2  1
```

5.1.4. Precedencias y prefijos

¿Como se debe interpretar la expresión `$$a[1]`? ¿Cómo `${$a[1]}` o bien como `${$a}[1]`?
La siguiente ejecución en el depurador nos muestra la respuesta:

```
  DB<1> $a = 4; $b = 5; $c = 6
  DB<2> @a = \($a, $b, $c)
  DB<3> $a = \@a
  DB<4> p $$a[1]
SCALAR(0x81046f4)
  DB<5> p @a
SCALAR(0x81046d0)SCALAR(0x81046f4)SCALAR(0x81046ac)
  DB<6> p ${$a[1]}
5
  DB<7> p ${$a}[1]
SCALAR(0x81046f4)
```

Cuando Perl evalúa una expresión deja la evaluación de los índices y las claves para el final. El prefijo mas cercáno a la variable es el primero en evaluarse. Es decir, los prefijos se evalúan de derecha a izquierda.

Las llaves significan un bloque de código. Así pues en la expresión `${ ... }[1]` la parte interna a las llaves es un bloque de código que debe, eso si, devolver una referencia a un array. Veamos un ejemplo:

```
sub t {
  return \$a;
}

$a = 10;
$b = ${t()};
print $b; # 10
```

5.1.5. La notación flecha

El acceso a los elementos de un `hash` o `array` a través de una referencia puede resultar incómodo, por eso Perl provee una sintáxis adicional:

```
$arr_ref = \@a;
$hsh_ref = \%h;
$sub_ref = \&s;
$a[0] = $hsh_ref->{"first"} # $a[0] = $h{"first"}
# o bien ...
$arr_ref->[0] = $h{"first"} # $a[0] = $h{"first"}
```

El operador `->` toma una referencia a su izquierda y un índice o clave a su derecha, localiza el `array` o `hash` correspondiente y accede al elemento apropiado.

Analize el siguiente ejemplo:

```
  DB<1> @a = (1, 0, 0); @b = (0, 1, 0); @c = (0, 0, 1)
  DB<2> @m = \(@a, @b, @c)
  DB<3> p @m
ARRAY(0x810461c)ARRAY(0x8104790)ARRAY(0x810479c)
  DB<4> $rm = \@m
  DB<5> p $rm->[2]
ARRAY(0x810479c)
  DB<6> p @$rm->[2] # Lo mismo que: p @{$rm}->[2]
ARRAY(0x810479c)
  DB<7> p @{$rm->[2]}
001
  DB<8> p $rm->[1,2]
ARRAY(0x810479c)
  DB<9> p $rm->[1]->[1]
1
```

¿Puede explicar la conducta observada en `p $rm->[1,2]`? La razón es que el operador de flecha no funciona con trozos y la subexpresión `1,2` es evaluada en un contexto escalar.

El operador flecha puede aplicarse a referencias a subrutinas, de manera que en vez de escribir:

```
&{$sub_ref}($arg1, $arg2, $etc);
```

podamos escribir:

```
$sub_ref->($arg1, $arg2, $etc);
```

5.2. Paso de arrays y hashes a subrutinas

El aplanado al que somete Perl los arrays y hashes cuando son pasados a una subrutina hace que sean indistinguibles desde la perspectiva de la subrutina llamada. Una forma de evitar este problema consiste en pasar como argumentos las referencias a los arrays y los hashes implicados. El siguiente ejemplo muestra una subrutina que recibe una referencia a una subrutina implantando una operación binaria, el elemento neutro de dicha operación y un vector de referencias a arrays. Se asume que el primer array es el de mayor tamaño. La rutina devuelve un vector conteniendo el resultado de operar vectorialmente los diferentes vectores.

```
$ cat -n vect2.pl
#!/usr/bin/perl  -w
use strict;

sub operate {
  my ($rop, $neutral, $rv1) = splice @_,0,3;
  my @res = @$rv1;
  my $i;

  for($i = 0; $i < @res; $i++) {
    for my $v (@_) {
      $res[$i] = $rop->($res[$i], defined($v->[$i])?$v->[$i]:$neutral);
    }
  }
  @res;
```

```
}

sub plus  {  $_[0]+$_[1] }
sub times {  $_[0]*$_[1] }

my @a = (3, 2, 1, 9);
my @b = (5, 6, 7);
my @m = operate( \&plus, 0, \@a, \@b);
print "@m\n";
my @c = (1, 2);
my @m2 = operate( \&times, 1, \@a, \@b, \@c);
print "@m2\n";
```

Sigue el resultado de la ejecución:

```
$ ./vect2.pl
8 8 8 9
15 24 7 9
```

5.2.1. Práctica: Conjuntos a través de Hashes

Un modo natural de representar conjuntos finitos en Perl es a través de un hash:

```
@A = qw(hiena coyote lobo zorro);
@Caninos{@A} = (); # los valores se dejan undef, solo nos importan las claves
@Pares{0, 2, 4, 6, 8} = ();
```

Defina funciones para

- Calcular el cardinal de un conjunto
- Determinar las relaciones pertenencia y subconjunto
- La operación de unión
- La operación de intersección
- Diferencia de conjuntos.

Alternativamente al hash puede usar una lista. Elija la opción que prefiera.

5.3. Identificando un referente `ref`

La función `ref` devuelve un string que indica el tipo del referente:

```
$ra = \$a;  # referencia a escalar
$rb = \@b;  # referencia a arreglo
$rc = \%c;  # referencia a hash
$rx = \$rb; # referencia a referencia
$rf = \&f;  # referencia a función

ref ( $ra ); #  devuelve "SCALAR"
ref ( $rb ); #  devuelve "ARRAY"
ref ( $rc ); #  devuelve "HASH"
ref ( $rx ); #  devuelve "REF"
ref ( $rf ); #  devuelve "CODE"
```

si el operando de `ref` no es una referencia, `ref` devuelve `undef`.

La función `ref` puede ser utilizada para mejorar los mensajes de error:

```
die "Expected scalar reference" unless ref($slr_ref) eq "SCALAR";
```

Si una referencia es utilizada en un contexto donde se espera una cadena, la función `ref` es llamada automáticamente produciendo una representación hexadecimal de la dirección del referente. Eso significa que una impresión como:

```
print $hsh_ref,"\n";
```

produce algo como esto:

```
HASH(0X10027588)
```

5.4. Referencias a almacenamiento anónimo

Una expresión como:

```
[ 'e1', 'e2', 'e3'];
```

devuelve una referencia a un `array` con esos elementos. Así es posible asignarlo a una variable escalar:

```
$rb = [ 'e1', 'e2', 'e3'];
```

Observe el uso de corchetes en vez de paréntesis. Aqui el *array* no tiene nombre, quien tiene nombre es la referencia. Por supuesto, `@$rb` denota al array referido.

Lo mismo puede hacerse con los `hash`es. en vez de corchetes se usan llaves:

```
$association = { cat =>"nap", dog=>"gone", mouse=>"ball"};
print "When I say 'cat', you say ...",$association->{cat};
```

y también con las subrutinas. Basta con suprimir el nombre de la subrutina:

```
$subref = sub { my $x = shift; $x*$x; };
```

Ejercicio 5.4.1.

- *¿Que sucede si en vez de corchetes usamos paréntesis? Si escribimos:*

  ```
  $rb = ( 'e1', 'e2', 'e3');
  ```

  ¿Que contiene `$rb`*?*

- *¿Que sucede si en vez de llaves usamos paréntesis? Si escribimos:*

  ```
  $association = ( cat =>"nap", dog=>"gone", mouse=>"ball");
  print "When I say 'cat', you say ...",$association->{cat};
  ```

  ¿Que contiene `$association`*?*

5.5. Estructuras anidadas

Es posible crear estructuras de datos multidimensionales usando `arrays` anónimos.

```
my $table = [ [1,2,3],[2,4,6],[3,6,9]];
```

Ahora es posible acceder a los elementos de table con expresiones como:

```
print $table->[$x]->[$y];
```

Perl considera opcional escribir la flecha entre dos corchetes:

```
print $table->[$x][$y];
```

Es posible crear `hash`es multinivel anidando referencias a "hashes" anónimos:

```
$behaviour = {
  cat   => { nap => "lap", eat=>"meat"},
  dog   => { prowl => "growl", pool=>"drool"},
  mouse => { nibble=>"kibble"}
};
```

Para acceder a los datos se requiere una cadena de flechas:

```
print "A cat eats ",$behaviour->{cat}->{eat},"\n";
```

Al igual que para los `array`s multidimensionales, las flechas después de la primera pueden ser suprimidas:

```
print "A mouse nibbles ",$behaviour->{mouse}{nibble},"\n";
```

5.6. Creación implícita y asignación de memoria

Las referencias anónimas suelen implicar el alojamiento automático de memoria. Observe el siguiente ejemplo:

```
DB<1> @a = 1..5
DB<2> $b = [ @a ]
DB<3> p $b
ARRAY(0x833f9a4)
DB<4> p @$b
12345
DB<5> @$b = 6..10
DB<6> p @a
12345
DB<7> p @$b
678910
```

Otro ejemplo. Si la primera línea de nuestro guión Perl es algo como:

`$task{ls}->{parameters}->[0] = "l";`

Perl creará un hash de referencias `task` al cual asignará una clave `ls` y cuyo valor es una referencia a una entrada de un hash, que será creado con una clave `parameters` cuyo valor es una referencia a un array cuya entrada 0 es `"l"`. La asignación de memoria es automática.

5.7. Impresión de estructuras anidadas

Para imprimir una estructura de datos compleja existen varias soluciones. La oficial es usar el módulo `Data::Dumper`. Si no esta instalado, existe una solución ”no oficial” pero que es muy efectiva: usar la función `dumpValue`, la cual se encuentra en el fichero `dumpvar.pl` que se distribuye con la librería estandar. Sigue un ejemplo:

```
$ cat mydumpvalue.pl
#!/usr/bin/perl -w

use strict;

require 'dumpvar.pl';

my @a = (1, {A => ['AB', 'empty'], B => ['bB', 'empty']});

dumpValue(\@a); # recibe su argumento como una referencia
```

```
$ ./mydumpvalue.pl
0  1
1  HASH(0x8106af0)
   'A' => ARRAY(0x8103068)
      0  'AB'
      1  'empty'
   'B' => ARRAY(0x8106b68)
      0  'bB'
      1  'empty'
```

El módulo `Data::Dumper` convierte la lista de escalares pasada como parámetro a su función `Dumper` en una cadena conteniendo código Perl que describe la estructura de datos.

```
$ cat datadumper.pl
#!/usr/bin/perl -w

use strict;
use Data::Dumper;

my @a = (1, {A => ['AB', 'empty'], B => ['bB', 'empty']});

my $m = Dumper(\@a);

print $m;


$ ./datadumper.pl
$VAR1 = [
          1,
          {
            'A' => [
                     'AB',
                     'empty'
                   ],
            'B' => [
                     'bB',
                     'empty'
                   ]
          }
        ];
```

5.8. Ejemplo: El Producto de Matrices

El siguiente ejemplo muestra una realización del producto de matrices en Perl. Una matriz es un vector de referencias a vectores que representan las filas. El número de filas (línea 5) es la longitud del vector, mientras que el número de columnas es el número de elementos en el vector referenciado por el primer item (línea 6).

```
1  #!/usr/bin/perl -w
2
3  sub matrixProd {
4    my ($matA, $matB) = @_;
5    my $nRowsA = @{$matA};
6    my $nColsA = @{$matA->[0]};
```

```
  my $nRowsB = @{$matB};
  my $nColsB = @{$matB->[0]};
  my ($matC, $i, $j, $k, $sum);

  if ($nColsA != $nRowsB) {
    die ("Las dimensiones de las dos matrices no son compatibles\n");
  }
  for($i = 0; $i < $nRowsA; $i++)  {
    for($j = 0; $j < $nColsB; $j++) {
      $sum = 0;
      for($k=0; $k < $nColsA; $k++) {
        $sum += $matA->[$i][$k] * $matB->[$k][$j];
      }
      $matC->[$i][$j] = $sum;
    }
  }
  return $matC;
}

sub printMat {
  my $mat = shift;
  my $nRows = @{$mat};
  my $nCols = @{$mat->[0]};
  my $i = 0;

  for($i = 0; $i < $nRows; $i++) {
    for($j = 0; $j < $nCols; $j++) {
      print $mat->[$i][$j], "\t";
    }
    print "\n";
  }
}

#
# ---- Main ----
#

$matA = [[1,2,3],[2,4,6],[3,6,9]];
$matB = [[1,2],[2,4],[3,6]];
$matC = matrixProd($matA,$matB);

print "Matriz A \n"; printMat($matA);
print "\nMatriz B \n"; printMat($matB);
print "\nMatriz C \n"; printMat($matC);
```

Todo queda mucho mas simple usando el módulo *PDL* (PDL significa *Perl Data Language*):

```
#!/usr/bin/perl -w
use PDL;

$a = pdl [[1,2,3],[2,4,6],[3,6,9]];
$b = pdl [[1,2],[2,4],[3,6]];
$c = $a x $b; # x esta sobrecargado

print "a = $a\nb = $b\nc = $c\n";
```

PDL es un módulo diseñado para el cálculo y visualización de datos científicos.

5.9. Ejercicio: Indentificadores entre LLaves

En general, en un contexto en el que se espera un identificador de variable, Perl interpreta un identificador entre llaves como la cadena literal que es. Explique la conducta de Perl ante el siguiente código:

```
1 @puf = (1..10);
2 toto(\@puf);
3
4 sub toto {
5   my @a = @{shift};
6   print "@a\n";
7 }
```

¿Que ocurre si sustituimos la línea 5 por `my @a = @{shift()};`?

5.10. Gestión de la memoria

En Perl, una *variable* representa una ligadura lógica entre un nombre y un valor.

Del mismo modo, un array es una colección de ligaduras lógicas a valores escalares. Véase la figura 5.1

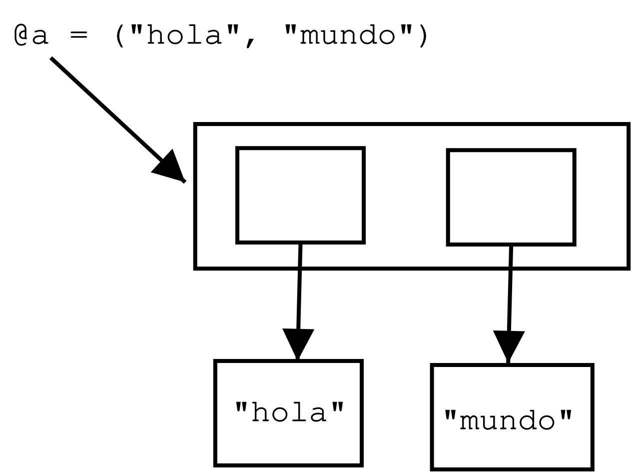

Figura 5.1: Un array es una colección de ligaduras lógicas a valores escalares

Perl mantiene un *contador de referencias* para cada valor existente en el programa. Este contador se incrementa tanto si es usado directamente mediante un nombre de variable como a través de una referencia. En todo momento el contador de referencia del valor asociado con una variable `$a` mantiene el número de referencias existentes a dicha valor. Puesto que hay una ligadura entre la variable `$a`, u valor, este contador es al menos uno. Véase la figura 5.2. Si este contador desciende hasta cero, Perl elimina la memoria asignada a ese valor. Los contadores descienden por diversas razones. Por ejemplo, cada vez que termina un bloque, el contador de los valores asociados con las variables declaradas en ese bloque desciende una unidad. En el caso habitual de que el contador valiera 1, pasará a valer 0 y la memoria asociada con el valor será liberada.

Existe un problema con este algoritmo de recolección de basura: las referencias circulares. En esos casos el programador debe actuar usando, por ejemplo, el operador `delete`.

El siguiente ejemplo muestra como los elementos de un array tienen sus propios contadores de referencias:

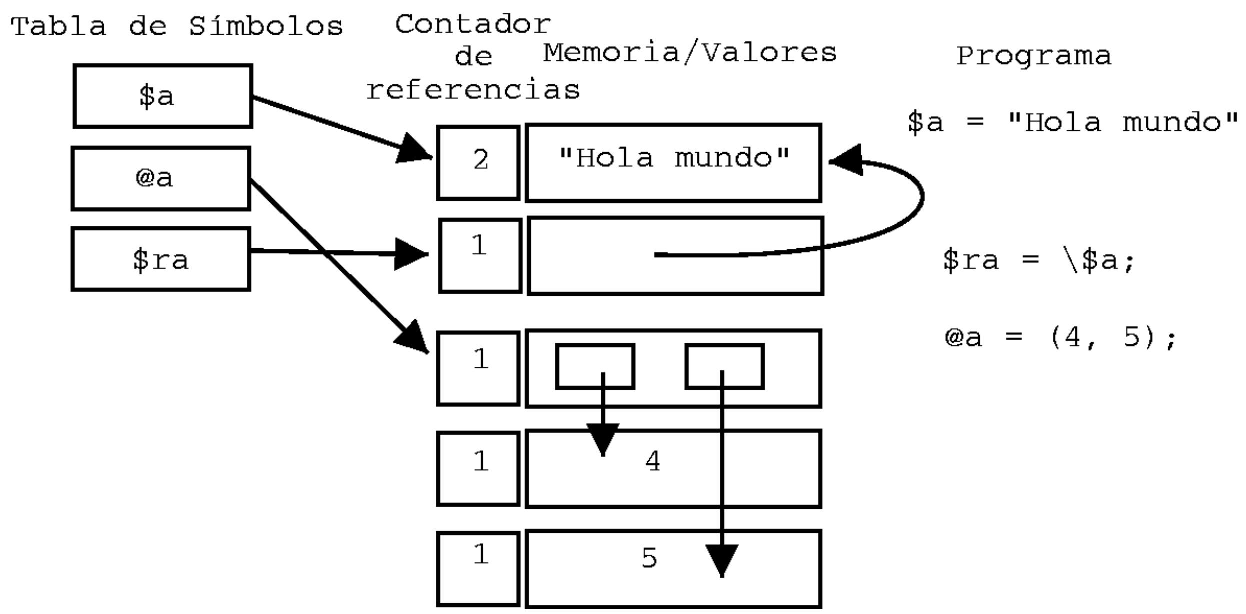

Figura 5.2: Contadores de referencia y asignaciones

```
DB<1> @a = 1..5
DB<2> $ra = \$a[4]
DB<3> pop @a
DB<4> p @a
1234
DB<5> p $$ra
5
```

la orden `pop @a` rompe la ligadura entre `$a[4]` y el valor, esto es, la ligadura-variable `$a[4]` es eliminada del array. Sin embargo el valor asociado no es eliminado de memoria ya que su contador de referencias todavía no es cero. La variable `$ra` aún esta referenciándolo.

5.11. Referencias simbólicas

Lo normal es que una expresión como `$$a` indique que `$a` es una variable del tipo referencia a un escalar. Si no fuera el caso, esto es, si `$a` no fuera una referencia, Perl comprueba si `$a` contiene una cadena (esto es, la variable está en un contexto escalar cadena) y si es el caso usa esa cadena como nombre de la variable. Esto se conoce como referenciado simbolico. Su uso es mas eficiente que la alternativa de usar `eval`. Veamos un ejemplo:

```
DB<1> $user = "juan"
DB<2> $a = "user"
DB<3> $$a = "pedro"
DB<4> p $user
pedro
```

El pragma `strict` impide el uso de referencias simbólicas:

```
$ cat symbol_ref.pl
#!/usr/bin/perl -w

use strict;

my $x = 4;
```

```
my $a = "x";
$$a = 10;

print $x;
$ ./symbol_ref.pl
Can't use string ("x") as a SCALAR ref while "strict refs" in use at ./symbol_ref.pl line 7.
```

Se puede ser mas específico y restringir solo el uso de referencias simbólicas con `use strict 'refs'`:

```
$ cat symbol_ref2.pl
#!/usr/bin/perl -w

use strict 'refs';

$x = 4; # no my
$a = "x"; # $a no declarada
$$a = 10;

print $x;
$ ./symbol_ref2.pl
Can't use string ("x") as a SCALAR ref while "strict refs" in use at ./symbol_ref2.pl line 7.
```

Si, por el contrario, lo que se quiere es permitir el uso de referenciado simbólico en un segmento del programa sin renunciar al control que nos da `use strict`, debemos usar la cláusula `no`:

```
$ cat -n ./symbol_ref3.pl
     1  #!/usr/bin/perl -w
     2
     3  use strict;
     4
     5  my $x = 4;
     6  my $a = "x";
     7  {
     8    no strict 'refs';
     9    $$a = 10;
    10  }
    11
    12  print "$x\n";
$ ./symbol_ref3.pl
4
```

Por tanto:

- Una referencia simbólica es simplemente una cadena de caracteres que contiene el nombre de una variable o subrutina de la tabla de símbolos de un determinado paquete.
- Cuando Perl encuentra una cadena alli donde esperaba encontrar una referencia, busca por una entrada en la tabla de símbolos con ese nombre y reemplaza la referencia de manera apropiada.

Otro ejemplo:

```
#!/usr/bin/perl -w
package main;

sub data {
  print "inside data\n";
}
```

```
$data = 4;
@data = (1,2,3,4);
$name = "data";

print "${$name}\n";
push  @{$name}, @data;
print "@data\n";
&{$name}();
```

La ejecución da como resultado:

```
~/perl/src> symbolic.pl
4
1 2 3 4 1 2 3 4
inside data
```

Si la cadena sigue las reglas de un nombre completo, Perl utilizará la tabla de símbolos adecuada:

```
$name = "General::Specific::data";
print ${$name}; # Lo mismo que: print $General::Specific::data;
```

Las referencias simbólicas tambien pueden usarse a la izquierda del operador `->`

```
$symref = "set";
$symref->{type} = "discrete"; # Lo mismo que: $set->{type} = "discrete";
```

Es importante señalar que, puesto que las referencias simbólicas acceden a la tabla de símbolos, no pueden ser usadas para acceder a las variables léxicas. Por ejemplo:

```
#!/usr/bin/perl
{

  my $grain = "headache";

  ${"grain"} = "rye";

  print "$grain\n";

}

print "$grain\n";
```

Imprime la primera vez `headache` y no `rye`. Es asi porque la variable léxica `$grain` oculta a la variable de paquete `$main::headache` en el ámbito. La salida del programa es:

```
$ symbolex.pl
headache
rye
```

En un programa Perl pueden existir varias tablas de símbolos. Una declaración `package` cambia el "espacio de nombres" hasta que encontremos una nueva declaración `package`, o hasta el final del bloque actual. De hecho, las variables en Perl se clasifican como `package variables` y `lexical variables`. Estas últimas son las declaradas con `my`. Las variables `package` pertenecen, naturalmente, a un `package` (normalmente el actual). El `package` inicial es el `package main`. Cuando sea necesario hacer explícito a que `package` pertenece la variable, puede hacerse prefijando su nombre con el del `package`, separado por `::`.

5.11.1. Práctica: Referenciado Simbólico

Escriba un guión Perl que reciba como entrada en la línea de comandos una opción `-Dvar=value` de manera que inicie una variable interna del guión cuyo nombre es el indicado por `$var` al valor indicado por `value`. Use referenciado simbólico para ello.

5.12. Referencias a subrutinas anónimas

Vimos antes que es posible mediante una referencia apuntar a una función ya existente:

```
1 sub tutu {
2   print "En tutu\n";
3 }
4
5 $refsub = \&tutu;
```

Note que en la línea 5 no se esta llamando a la función `&tutu` de la misma forma que cuando tomamos una referencia a una variable escalar no evaluamos el escalar. La cosa cambia totalmente si la asignación de la línea 5 se cambia por `$refsub = \&tutu()`. ¿Qué ocurre en este caso?

Para crear una referencia a una subrutina anónima, simplemente, omitimos el nombre de la subrutina. He aqui un ejemplo de referencia a subrutina anónima:

```
$sub_ref = sub { print "Hola $_[0]!\n" };
```

y un ejemplo de llamada:

```
$sub_ref->("Ana");
```

o bien:

```
&$sub_ref("Ana");
```

5.13. Funciones de orden superior

Una de las ventajas de disponer de referencias a funciones es que podemos escribir meta-funciones: funciones que reciben funciones como parámetros. Se dice de tales funciones que son *funciones de orden superior*.

El siguiente ejemplo muestra una función `sum` que recibe como parámetros de entrada dos referencias a funciones de una variable y devuelve una referencia a la función suma de aquellas.

```
#!/usr/local/bin/perl5.8.0 -w

use strict;

use Set::Scalar;

sub f {
  my $x = shift;
  2*$x;
}

sub g {
  my $x = shift;
  $x*$x;
}
```

```
sub sum {
  my $rf1 = shift;
  my $rf2 = shift;

  return (sub { my $x = shift;  &$rf1($x)+&$rf2($x); });
}

my $rf = \&f;
my $rg = \&g;

my $s = sum($rf, $rg);


print "Dame un número: ";
my $x;
chomp($x = <>);
print "f($x)+g($x) = ",&$s($x),"\n";

print "Dame una subrutina anónima ";
my $r;
chomp($r = <>);
$r = eval($r);
my $t = sum($r, $s);
print "Tu función($x)+f($x)+g($x) = ",&$t($x),"\n";
```

Sigue un ejemplo de ejecución:

```
$ ./readfunction.pl
Dame un número: 4
f(4)+g(4) = 24
Dame una subrutina anónima sub { my $x = shift; $x -1; }
Tu función(4)+f(4)+g(4) = 27
```

5.13.1. Práctica: Emulación de un Switch

Vamos a realizar una función de orden superior: Emule la sentencia `switch` de C usando un hash de punteros a subrutinas. Escriba una función `switch` que recibe como segundo parámetro un hash con claves las constantes del `case` y como valores las referencias a las subrutinas. Cómo primer parámetro recibe un valor escalar. Según sea el valor deberá ejecutar la subrutina con clave correspondiente. Si no hay ninguna clave con ese valor deberá de ejecutar la subrutina que corresponde a la clave `default`, si tal clave existe en el hash.

En el ejemplo que haga para la llamada a la función `switch` use subrutinas anónimas cuando el código en cuestión lleve menos de una línea.

El módulo `Switch` de Damian Conway proporciona una sentencia `switch` mas completa. Sigue un ejemplo de uso:

```
$ cat -n useswitch.pl
#!/usr/local/bin/perl5.8.0 -w

use Switch;

$val = shift;

switch ($val) {
```

```
  case 1 { print "number 1\n" }
  case "hello" { print "string hello\n" }
  else { print " none of these two\n" }
}
```

5.14. Typeglobs

5.14.1. Introducción

Perl mantiene una tabla de símbolos o espacios de nombres (que internamente es una tabla hash) separados para cada paquete y cada tipo de objeto en un paquete. Asi pues, fijado un paquete tenemos las variables

```
$file, @file, %file, &file
```

que son distintas y que pueden ser usadas simultáneamente. Puesto que una tabla hash no permite claves duplicadas, Perl interpone una estructura que se sitúa entre la entrada de la tabla de símbolos y la memoria para los diferentes tipos de variable para cada prefijo. Podemos decir que esta estructura es, en cierto modo, un *typeglob*.

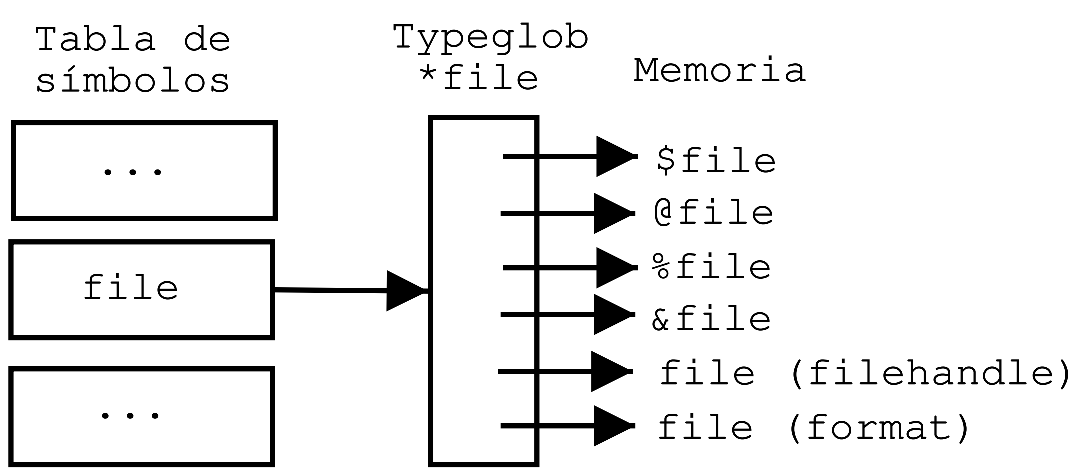

Figura 5.3: Un typeglob nombra a la estructura que se interpone entre la tabla de símbolos y la memoria

Perl provee una sintáxis especial, denominada "typeglob" para referirse conjuntamente a los diferentes tipos de variable: `*file`. (Piensa en `*` como en un comodín).

5.14.2. Variables léxicas y typeglobs

Las variables léxicas (declaradas con `my`) no se introducen en las tablas de símbolos de los paquetes. En estas sólo van las variables globales. Cada bloque y subrutina tienen su propia tabla de símbolos, que se denominan en la jerga Perl *scratchpads*. A cada variable léxica se le asigna una ranura en el *scratchpad*.

5.14.3. Asignación de typeglobs

Una asignación de un "typeglob" a otro, tal como:

```
*file = *source;
```

hace que `file` se convierta en un sinónimo de `source`: La entrada en la tabla de símbolos para `file` es una copia de la entrada en la tabla de símbolos de `source`. Así `$file` referencia a la misma variable que `$source`, `@file` a la misma que `@source`, etc.

Un “typeglob” puede ser visto como un hash o vector de referencias con un elemento por tipo de variable (escalar, lista, “hash”, etc.). Cuando se asigna un “typeglob” a otro se copia dicho vector de referencias.

5.14.4. `local` y typeglobs

No existe forma de eliminar un typeglob. Sin embargo es posible usar `local` sobre un typeglob para salvar temporalmente su valor:

```
$b = 5;
{
  local *b;
  *b = *a;
  $b = 20;
}
print $a; # 20
print $b; #  5
```

Es un error intentar hacer un *typeglob* de una variable privada (`my`) de tipo typeglob:

```
$ cat mytypeglob.pl
#!/usr/bin/perl  -w
use strict;

my *a;
my $b;

*a = *b;

$ ./mytypeglob.pl
syntax error at ./mytypeglob.pl line 4, near "my *a"
Execution of ./mytypeglob.pl aborted due to compilation errors.
```

5.14.5. Paso de filehandles como parámetros a una subrutina

Los ficheros (filehandles) no se pueden asignar o pasar como parámetros, esto es, es ilegal hacer:

```
open(F,"fich1");
open(G,"fich2");
F = G;
```

En los años A.R. (Antes de las Referencias) la alternativa era hacer una asignación de los typeglobs: `*F = *G`

Lo mismo ocurría para el paso de ficheros como parámetros de una función. Vea el siguiente ejemplo:

```
#!/usr/bin/perl -w
sub welcome {
  my $fh = shift;

  print $fh "Welcome ...\n";
}

open(FILE, ">test.txt");
$file = *FILE;
welcome($file);
close($file);
```

Es mas fácil usar el módulo `IO::File`

```
$ cat -n ./fileparameter3.pl
     1  #!/usr/bin/perl -w
     2
     3  use IO::File;
     4
     5  sub welcome {
     6    my $fh = shift;
     7
     8    print $fh "Welcome ...\n";
     9  }
    10
    11  $file = new IO::File "test.txt", "w";
    12  die "No pude abrir el fichero: $!" unless $file;
    13  welcome($file);
    14  $file->close;
```

5.14.6. Typeglobs y eficiencia

Hacer un alias via un typeglob es mas eficiente que usar una referencia. Para verlo usaremos el módulo `Benchmark` el cual proporciona, entre otras muchas, la función `timethese`:

```
$ cat benchtypeglobs.pl
#!/usr/bin/perl
#use strict;
use Benchmark;

local $i;
*b = *i;
$r = \$i;

timethese(4,
  { reference => sub { for ($$r = 0; $$r < 1e6; $$r++) { $$r; } },
    typeglob  => sub { for ($b = 0; $b < 1e6; $b++) { $b; } }
  });
```

El resultado de la ejecución en `nereida` nos da una relación de apróximadamente 1.6 veces mas rápido:

```
$ ./benchtypeglobs.pl
Benchmark: timing 4 iterations of reference, typeglob...
 reference:  6 wallclock secs ( 6.57 usr +  0.00 sys =  6.57 CPU) @  0.61/s (n=4)
  typeglob:  4 wallclock secs ( 4.11 usr +  0.01 sys =  4.12 CPU) @  0.97/s (n=4)
```

5.14.7. Ejercicio: Typeglobs

¿Que se imprimirá al ejecutar el siguiente código?

```
$a = 3.14159;
my $a = 5;
*b = *a;
print $b;
```

5.14.8. Práctica: Includes C

Escriba una subrutina que recibe un fichero conteniendo un programa C y devuelve la lista de ficheros incluidos en dicho programa (esto es aquellos que figuran en una directiva `#include` ...). Si el fichero incluido incluye a su vez otros ficheros, estos deberán figurar en la la lista. Escriba la subrutina de manera que sea recursiva.

No es necesario que controle la presencia de comentarios. Opcionalmente, si desea controlar la presencia de `#include` ... dentro de comentarios, léa la sección 4.13.

5.14.9. Typeglobs selectivos

Los "typeglobs" puede asignarse selectivamente a una referencia:

```
*SOURCE = \$SOURCE1;
*args = \@ARGV;
*do_it = sub { print "do it!\n" };
```

La primera asignación hace que `$SOURCE` sea un sinónimo de `$SOURCE1`, pero `@SOURCE`, `%SOURCE` y `&SOURCE` continúan con sus anteriores valores. Lo mismo se aplica a los otros dos ejemplos.

Es legal tomar una referencia a un "typeglob". Por ejemplo:

```
  DB<1> $a = 4; @a = 1..5
  DB<2> $b = \*a
  DB<3> x $b
0  GLOB(0x8450df8)
   -> *main::a
  DB<4> x ${*$b}
0  4
  DB<5> x @{*$b}
0  1
1  2
2  3
3  4
4  5
```

Asi, el código en la sección 5.14.5 puede ser reescrito:

```
#!/usr/bin/perl -w
...

open(FILE, ">test.txt");
@file = (1,2,3,4);
$file = \*FILE;
welcome($file);
print "@file\n";
close($file);
```

El resultado de la ejecución es:

```
$ filehandle2.pl
1 2 3 4
$ cat test.txt
Welcome ...
```

otro ejemplo:

```
$variable = 'esto es una $variable';
%variable = (v => "ven", a => "a", r=> "rumania");
sub variable { print "esto es una $variable" };

$typeglob_ref = \*variable;
```

Ahora `typeglob_ref` contiene una referencia a la entrada en la tabla de símbolos para `variable`. Podemos acceder a los elementos individuales a través de la referencia:

```
${*$typeglob_ref}; # sinónimo de $variable
%{*$typeglob_ref}; # sinónimo de %variable
&{*$typeglob_ref}(); # sinónimo de &variable()
```

5.14.10. Ejercicio: Asignaciones a Typeglobs

¿Que se imprimirá al ejecutar el siguiente código?

```
#!/usr/local/bin/perl5.8.0 -w
sub mundo { "mundo\n" }

*hola = \&mundo;
$mundo = "hola";
$hola = "bienvenido";
print "$mundo ",&hola();
```

5.14.11. Typeglobs vistos como hashes

Perl dispone de una sintáxis especial para acceder a un "typeglob" como si fuera un "hash" indexado en el tipo de variable:

```
$scalar_ref = *variable{SCALAR} # $scalar_ref = \$variable
$array_ref = *variable{ARRAY} # $scalar_ref = \@variable
$hash_ref = *variable{HASH} # $scalar_ref = \%variable
$sub_ref = *variable{CODE} # $scalar_ref = \&variable
```

Muy interesante: podemos usar esta sintáxis para acceder a la entrada de fichero del "typeglob":

```
$handle_ref = *variable{IO}
```

Veamos un ejemplo con el depurador:

```
  DB<1> $a = 4; @a = 1..5
  DB<2> $rsa = *a{SCALAR}
  DB<3> x $rsa
0  SCALAR(0x8450e04)
   -> 4
  DB<4> open a, "matrixP.pl"
  DB<5> $rfa = *a{IO}
  DB<6> x $rfa
0  IO::Handle=IO(0x847db4c)
  DB<7> $b = <$rfa>
  DB<8> p $b
#!/usr/bin/perl -w
```

Usando esta sintáxis podemos construir una referencia a la componente "hash" partiendo de una referencia a un "typeglob" con una asignación como esta:

```
$hsh_ref = *$typeglob_ref{HASH};
```

5.14.12. Referencias simbólicas y typeglobs

Podemos incluso usar referencias simbólicas para acceder a la tabla de símbolos. Por ejemplo, si tenemos

```
$symbol_name = "data";
```

Entonces podemos acceder a traves del mismo al "typeglob" como `*{$symbol_name}` en vez de `*{data}`.

5.15. Prototipos

Los prototipos en Perl son un mecanismo que permite la activación de ciertas comprobaciones asi como escribir subrutinas cuyo comportamiento disfrute de privilegios similares a los de los operadores internos de Perl.

Como ejemplo, supongamos que queremos escribir una función `shift2` que elimina los dos primeros elementos del principio de un array. Supongamos que queremos usarla como `shift`:

```
@a = 1..10;
($f, $s) = shift2 @a;
```

Si queremos escribirla, no vale hacer:

```
sub shift2 { splice @_, 0, 2 }
```

Ejercicio 5.15.1. *Observe el comportamiento del siguiente programa.*

```
$ cat -n shift2.pl
#!/usr/bin/perl -w
use strict;

sub shift2 { splice @_, 0, 2 }

my @a = 1..5;
my ($f, $g) = shift2 @a;

local $" = ’,’;
print "f = $f, g = $g  \@a = (@a)\n";
$ ./shift2.pl
f = 1, g = 2  @a = (1,2,3,4,5)
```

¿Sabria explicar la salida?

La solución al problema es pasar el array por referencia:

```
sub shift2 { splice @{$_[0]}, 0, 2 }
```

pero entonces la llamada queda distinta, obligándonos a pasar la referencia:

```
@a = 1..10;
($f, $s) = shift2 \@a;
```

Si queremos que la interfase de `shift2` emule el comportamiento de `unshift` podemos lograrlo, haciendo uso de los *prototipos*. Podemos redeclarar la función como:

```
sub shift2(\@) { splice @{$_[0]}, 0, 2 }
```

Aqui el prototipo `\@` indica que si la rutina es llamada con un argumento de tipo array, el argumento será automáticamente convertido a una referencia a un array. Asi las llamadas siguientes producen error:

```
@x = shift2 %a
Type of arg 1 to main::shift2 must be array (not hash dereference)

@x = shift2 \@a
Type of arg 1 to main::shift2 must be array

@x = shift2 @a, @b
Too many arguments for main::shift2
```

Como se ve en el ejemplo, se comprueba que el número y la clase de los parametros coinciden. Para que los prototipos trabajen de esta forma la llamada a la función debe hacerse sin prefijarse de `&`. Vea el siguiente ejemplo:

```
$ cat -n ./shift2.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3
 4  sub shift2(\@@) { my $arr = shift; splice @$arr, 0, 2 }
 5
 6  my @a = 1..6;
 7  my ($f, $g) = shift2 @a;
 8
 9  local $" = ',';
10  print "f = $f, g = $g  \@a = (@a)\n";
11
12  ($f, $g) = shift2(@a);
13  print "f = $f, g = $g  \@a = (@a)\n";
14
15  # dará error
16  ($f, $g) = &shift2(@a);
17  print "f = $f, g = $g  \@a = (@a)\n";
$ ./shift2.pl
f = 1, g = 2  @a = (3,4,5,6)
f = 3, g = 4  @a = (5,6)
Can't use string ("5") as an ARRAY ref while "strict refs" in use at ./shift2.pl line 4.
```

La llamada de la línea 16 produce un error: lo queda de `@a` es enviado como parámetro a la rutina y se produce la protesta.

Los prototipos se construyen a partir de *átomos prototipo* los cuáles estan hechos de un prefijo o de un escape y un prefijo. Así un prototipo como `\$` le indica a Perl que debe pasar a la subrutina la referencia al argumento escalar.

Sin embargo un prototipo de la forma `$` lo que hace es que fuerza un contexto escalar.

Ejercicio 5.15.2. *Observe la conducta del siguiente programa:*

```
$ cat -n ./dollarproto.pl
#!/usr/bin/perl -w
use strict;

sub t ($@) { my $a = shift; my @b = @_; print "a = $a, b = (@b)\n"; }

my ($a, $b, $c) = qw/uno dos tres/;
```

```
t ':',$a, $b, $c;

my @r = 1..5;
t @r;

$ ./dollarproto.pl
a = :, b = (uno dos tres)
a = 5, b = ()
```

¿Podría explicar la salida?

Sin embargo, un prototipo del tipo `@` o del tipo `%` no sólo fuerza un contexto de lista sino que se traga todo el resto de argumentos.

Un prototipo de la forma `&` fuerza una referencia a código. Si se trata del primer argumento, entonces la palabra `sub` es opcional y se puede llamar como los operadores `eval` o `grep`:

```
$ cat -n ampproto.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3
     4  sub t (&$) { my ($f, $x) = @_; print &$f($x),"\n" }
     5
     6  t { $_[0]**3 } 4;
     7
$ ./ampproto.pl
64
```

Observe la ausencia de coma en la llamada de la línea 6.

Si el prototipo es `*` fuerza una referencia a un typeglob. Por último un prototipo de la forma `;` separa los argumentos requeridos de los argumentos opcionales.

Veamos otros ejemplos de declaraciones de prototipos de operadores ya existentes en Perl:

`myreverse (@)`	`myreverse $a,$b,$c`
`myjoin ($@)`	`myjoin ":",$a,$b,$c`
`mypop (\@)`	`mypop @array`
`mysplice (\@$$@)`	`mysplice @array,@array,0,@pushme`
`mykeys (\%)`	`mykeys %{$hashref}`
`myopen (*;$)`	`myopen HANDLE, $name`
`mypipe (**)`	`mypipe READHANDLE, WRITEHANDLE`
`mygrep (&@)`	`mygrep { /foo/ } $a,$b,$c`
`myrand ($)`	`myrand 42`
`mytime ()`	`mytime`

Ejercicio 5.15.3. *Suponga que quiere escribir una función* `mypush` *que actúa exactamente como lo hace* `push`*: La llamada* `push @a, 4, 5, 6` *empuja los elementos de la lista* `(4, 5, 6)` *en* `@a`*. ¿Como debería ser el prototipado de dicha subrutina?*

5.16. Clausuras

El concepto de *clausura* es una noción que viene del mundo de la programación en Lisp. Se refiere a que cuando se define una subrutina en un determinado contexto léxico, se ejecuta usando ese contexto incluso si es llamada fuera de ese contexto.

En Perl una clausura es una subrutina que referencia a una o mas variables léxicas declaradas fuera de la subrutina. Por ejemplo:

```
#!/usr/local/bin/perl5.8.0 -w

local $name = "Red Hat";
{
  my $name = "Debian";

  sub print_my_name {
    print $name, "\n";
  }
}

print $name,"\n";
print_my_name;
```

cuando se ejecuta produce la siguiente salida:

```
$ ./closure.pl
Red Hat
Debian
```

En condiciones normales Perl liberaría la memoria correspondiente a la variable `name` al final de su ámbito. Sin embargo, la subrutina `print_my_name` que la usa continúa visible y, por tanto, Perl conserva la variable `name` mientras la rutina este en uso. Obsérvese que la única forma de acceder a `name` es a través de la subrutina `print_my_name`.

Esta es la característica de una clausura: una subrutina que preserva las variables léxicas que esta usando, incluso si estas resultan invisibles desde cualquier otra parte.

5.16.1. Clausuras y Generación de Funciones Similares

Las clausuras pueden ser un buen mecanismo para generar dinámicamente una familia de funciones que se diferencian en algún parámetro. El siguiente ejemplo está tomado de la documentación de Perl (`perldoc perlref`). Se quieren esribir diversas funciones, una por cada color, de manera que devuelvan el texto HTML que pone en el correspondient color la cadena que se les pase como parámetro:

```
$ cat -n colors.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3
     4  my @colors = qw(red blue green yellow orange purple violet);
     5  for my $name (@colors) {
     6  no strict 'refs';       # permitir la manipulación de la tabla de símbolos
     7  *$name = *{uc $name} = sub { "<FONT COLOR='$name'>@_</FONT>" };
     8  }
     9
    10  print "<Tenga ", red("cuidado"), "con esa ", green("luz!"),"\n";
    11  print "<Tenga ", RED("cuidado"), "con esa ", GREEN("luz!"),"\n";
```

La clausura se forma en la línea 7 con la variable léxica `$name` declarada en la línea 5. Se han usado typeglobs para instalar entradas en la tabla de símbolos a la función tanto en minúsculas como en mayúsculas. Al ejecutar este programa tenemos la salida:

```
$ ./colors.pl
<Tenga <FONT COLOR='red'>cuidado</FONT>con esa <FONT COLOR='green'>luz!</FONT>
<Tenga <FONT COLOR='red'>cuidado</FONT>con esa <FONT COLOR='green'>luz!</FONT>
```

5.16.2. Anidamiento de subrutinas

Ya vimos en la sección 1.9.1 que Perl no anida subrutinas como ocurre en Pascal. Sin embargo, el uso combinado de clausuras, typeglobs y `local` permite emular el efecto:

```
$ cat -n nestedsub.pl
 1  #!/usr/local/bin/perl5.8.0 -w
 2
 3  sub interior {
 4    my $x = shift;
 5    print "sub marine: $x\n";
 6    $x*$x;
 7  }
 8
 9  sub exterior {
10    my $x = $_[0] + 3;
11    local *interior = sub { print "interior anidada: $x\n"; return $x - 1 };
12    return $x + interior();
13  }
14
15
16  print 'exterior(2) = ',exterior(2),"\n";
17  print 'interior(3) = ',interior(3),"\n";
```

En este ejemplo se ha creado en la línea 11 una subrutina denominada `interior` que ensombrece a la subrutina `interior` global definida en la línea 3 y que además tiene acceso al ámbito de `exterior`. Esta subrutina `interior` de la línea 11 es accesible solamente desde la subrutina `exterior`. El resultado de la ejecución es:

```
$ ./nestedsub.pl
Subroutine main::interior redefined at ./nestedsub.pl line 11.
interior anidada: 5
exterior(2) = 9
sub marine: 3
interior(3) = 9
```

5.16.3. Clausuras e Iteradores

El siguiente ejemplo muestra una subrutina que retorna una referencia a una clausura, la cual puede usarse para saltar a través de un array con un paso fijo:

```
sub hop_along {
  my ($from, $to, $step) = @_;
  my $next = $from-$step;  # inicializar contador
  my $closure_ref = sub {
    $next += $step;
    return 0 if $next > $to;
    $_[0] =$next;
    return 1;
  }; # Ponemos ; puesto que es una asignación
  return $closure_ref;
}
```

El generador podria usarse como sigue:

```
$iterator = hop_along 1, 100, 7; # crear la clausura
```

```
while ($iterator->($next)) {     # llamar a la clausura
  print $next;
}
```

Observe que la siguiente llamada a `hop_along` crea un conjunto nuevo de variables léxicas y una nueva subrutina anónima. Es posible simultanear de este modo dos clausuras, cada una con su propio rango y paso:

```
$row = hop_along 1, 1024, 1;
while ($row->($r)) {
  $col = hop_along 1, 768, 2;
  while ($col->($c)) {
    print "$r, $c\n";
  }
}
```

Ejercicio 5.16.1. *La función* `time` *devuelve el número de segundos desde Enero de 1970. Utilizando esta función rellene el código marcado con ... en el programa que figura a continuación:* `timestamp`

```
#!/usr/local/bin/perl5.8.0 -w
sub timestamp {
  my ...
  return sub { ... };
}

$first_exp = timestamp();
sleep(1);
$second_exp = timestamp();
sleep(1);
$f_end = $first_exp->();
$s_end = $second_exp->();
print "First exp $f_end\n";
print "Second exp $s_end\n";
```

La salida del programa deberá ser algo parecido a:

```
$ ./closure_time.pl
First exp 2
Second exp 1
```

5.16.4. Currying

Se denomina *currying* (en honor al lógico americano *Haskell Curry*, 1900-1982) al proceso de representar una función de múltiples argumentos como una función de un sólo argumento. Por ejemplo, la función suma `$x+$y` es una función de dos argumentos `($x, $y)`, pero puede verse también como una función de un argumento `$x`. ¿Cómo?.

La clave esta en pensar en la función suma como una función que cuando recibe el primer argumento `$x` devuelve como resultado una función `sub { $x+$_[0] }` la cual suma a su argumento el primer argumento `$x`.

Ejercicio 5.16.2. *En el siguiente listado se ha suprimido el cuerpo de la función* `curry`*, la cual* currifica *una función de mas de un argumento. Escriba el código que falta en las líneas 6-8. ¿Donde se esta haciendo uso del concepto de clausura aqui? ¿Que salidas producen las líneas 20-23? Teniendo en cuenta que* `$tu9` *es un puntero a una función, ¿no se puede reescribir la línea 18 como* `my $tu9_7 = curry $tu9, 7`*?*

```
$ cat -n currying.pl
#!/usr/bin/perl -w
use strict;

sub curry(&$) {
 # Rellene los contenidos



}

sub times { $_[0]*$_[1] }
sub tutu  {$_[0]+$_[1]*$_[2] }

my $p4 = curry { $_[0]+$_[1] } 4;
my $t5 = curry { $_[0]*$_[1] } 5;
my $t6 = curry \&times, 6;
my $tu9 = curry \&tutu, 9;
my $tu9_7 = curry \&$tu9, 7;

print $p4->(8),"\n";
print $t5->(2),"\n";
print $t6->(3),"\n";
print $tu9_7->(5),"\n";
```

5.16.5. Memoizing

Una función se dice *pura* si no tiene efectos laterales. Dicho de otro modo: el valor retornado sólo depende de sus argumentos de entrada. En ocasiones, es posible acelerar la ejecución de una función pura utilizando una estrategia denominada *memoizing*. La estrategia consiste en hacer una cache, habitualmente un hash, en el que las claves son los argumentos y los valores los resultados. El siguiente ejemplo *memoiza* la función que computa los números de Fibonacci:

```
$ cat -n ./memoize.pl
#!/usr/bin/perl -w
use Benchmark;

sub fib {
  my $n = shift;
  if ($n < 2) { $n }
  else { fib($n-1)+fib($n-2) }
}

{

my @fib = (0, 1);

  sub fibm {
    my $n = shift;

    return $fib[$n] if defined $fib[$n];
    $fib[$n] = fibm($n-1)+fibm($n-2);
  }
}
```

```
21
22 timethese(2, {
23   recursivo => sub { &fib(30) },
24   memoized  => sub { &fibm(30) }
25   }
26 );
```

Este es un ejemplo en el cuál es conveniente ocultar la cache e impedir posibles colisiones de la variable `@fib` con cualesquiera otras variables que pudieran existir en el programa. La solución es hacer una clausura (líneas 10-20).

La ejecución muestra como la técnica mejora claramente el rendimiento:

```
$ time ./memoize.pl
Benchmark: timing 2 iterations of memoized, recursivo...
  memoized:  0 wallclock secs ( 0.00 usr +  0.00 sys =  0.00 CPU)
            (warning: too few iterations for a reliable count)
 recursivo:  5 wallclock secs ( 5.81 usr +  0.00 sys =  5.81 CPU) @  0.34/s (n=2)
            (warning: too few iterations for a reliable count)

real    0m5.894s
user    0m5.880s
sys     0m0.010s
```

Ejercicio 5.16.3.

- Memoize *la función que computa el factorial de un número y llame a la función 30 veces para calcular el factorial de los 30 primeros números. ¿Por qué no se obtiene ninguna ventaja al memoizar la función factorial si sólo se llama una vez para un número dado N?*

5.16.6. Compartición, Persistencia y Privacidad: Un ejemplo con threads

Como se vió en la sección 5.16.5 las clausuras permiten a las subrutinas mantener variables privadas persistentes, algo parecido al comportamiento que se obtiene en C cuando se usa una variable "static". Lo que no se puede expresar tan fácilmente en C es el hecho de que dos o mas subrutinas pueden compartir las mismas variables privadas.

En el ejemplo que sigue se muestran dos *threads* o *hilos* que comparten un recurso: la variable `$locked` la cuál ambas modifican. El ejemplo sirve para ilustrar la ventaja de usar una clausura: La variable `$locked` esta clausurada y compartida por las subrutinas `tutu` y `titi`:

```
$ cat -n shared2.pl
#!/usr/bin/perl -w
use strict;
use threads;
use threads::shared;
{
  my $locked : shared = 0;

  sub tutu { sleep(rand(3)); lock($locked); $locked++; $locked }
  sub titi { sleep(rand(3)); lock($locked); $locked--; $locked }
}

my $t = threads->new(\&tutu); # creamos la thread
my $rm = titi();
my $rs = $t->join(); # sincronizamos las dos threads
print "Maestro: $rm\nEsclavo: $rs\n";
```

El paquete `threads`, usado en la línea 3, nos proporciona las herramientas para la creación de threads. El método `new()` (línea 12) toma una referencia a una subrutina y crea una nueva thread que ejecuta concurrentemente la subrutina referenciada.

Si se hubiera necesitado, es posible pasar parámetros a la subrutina como parte de la fase de arranque:

```
    $thr = threads->new(\&tutu, "Param 1", "Param 2", 4);
```

El método `join()` usado en la línea 28 retorna cuando la thread `$t` termina. Además recolecta y retorna los valores que la thread haya retornado. En general es una lista:

```
@ReturnData = $t->join;
```

Cuando se crea un nuevo hilo, todos los datos asociados con el hilo actual se copian en el nuevo. Por tanto, las variables son, por defecto privadas a la thread. En la mayoría de los casos se pretende que exista alguna forma de comunicación entre los hilos, para lo cual es conveniente disponer de mecanismos para hacer que ciertas variables sean compartidas por los hilos. Esta es la función del módulo `threads::shared` y del atributo `shared` (línea 6). La función `lock` proporcionada por el módulo `threads::shared` nos permite sincronizar el acceso a la variable. El cerrojo se libera al salir del contexto léxico en el que se produjo el `lock`. En el ejemplo, se libera al salir de la correspondiente subrutina. No existe por tanto una función `unlock`.

Veamos una ejecución:

```
$ ./shared2.pl
Maestro: 0
Esclavo: 1
$ ./shared2.pl
Maestro: -1
Esclavo: 0
```

Parte II

Programación Avanzada en Perl

Capítulo 6

Módulos

6.1. Introducción a los packages

Cada módulo suele contener una declaración `package`. Una declaración `package` cambia el "espacio de nombres" hasta que encontremos una nueva declaración `package`, o hasta el final del bloque actual. De hecho, las variables en Perl se clasifican como `package variables` y `lexical variables`. Estas últimas son las declaradas con `my`. Las variables `package` pertenecen, naturalmente, a un `package` (normalmente el actual).

El `package` inicial es el `package main`. Cuando sea necesario hacer explícito a que `package` pertenece la variable, puede hacerse prefijando su nombre con el del `package`, separado por `::`.

```
package C110;
# estamos en el espacio de nombres C110

$a = 5;           # variable del paquete C110
fun1              # función del paquete C110
{
  print  "$a\n";
}

package D110;
# ahora estamos en el espacio de nombres D110
# ...salimos del paquete C110

$a = 7;           # esta $a es del paquete D110
print  $a;        # imprime 7

print  $C110::a;
# imprime 5
# note como accesamos el espacio de nombres C110...
# note el $ y los ::

C110::fun1;       # llama a fun1 de C110...imprime: 5
```

Asi pues, para acceder a un identificador situado en un espacio de nombres diferente del actual debemos prefijar el identificador con el nombre del paquete; esto se denomina *especificación completa del nombre* o *fully qualifying the name*. Si un identificador no está completamente especificado, Perl lo busca en el `package` actual.

Nótese que escribimos `$C110::a` y no `C110::$a`

6.2. Tablas de símbolos y packages

Cada package tiene su propia tabla de símbolos. Así para acceder a una variable es preciso especificar de que tabla se habla (el package) y de que variable se habla (identificador dentro de la tabla). Las variables en el package `main` pueden ser referidas prefijándolas con `::`; así el programa:

```
$ cat package_main.pl
#!/usr/local/bin/perl5.8.0 -w

$a = 4;

package toto;
$a = 8;
$::a = 7;

package main;
print $a, "\n";
print $toto::a, "\n";
```

produce como salida:

```
$ ./package_main.pl
7
8
```

Una excepción a esto la constituye las variables especiales como `$_`, `@ARGV`, `%ENV`, etc. Estas pertenecen siempre al *paquete* `main`.

```
#!/usr/local/bin/perl5.8.0 -w

package toto;
$_ = 7;
package main;
print;
print "\n";
$main::_ = 4;
print;
print "\n";
```

Da como salida:

```
$ ./package_specialvars.pl
7
4
```

6.2.1. Ejercicio: Variables Léxicas

Por supuesto, como las variables léxicas no van en las tablas de símbolos sino en los *scratchpads*, no tiene sentido prefijarlas con un identificador de paquete. Sin embargo el ámbito léxico se refiere al texto mientras que el ámbito de paquete se refiere a la tabla de símbolos. Para entender esto, averigue y explique la salida del siguiente programa:

```
$ cat privacy.pl
#!/usr/local/bin/perl5.8.0 -w

package tutu;
my $a = 10;
```

```
package titi;

if (defined($tutu::a)) { print "$tutu::a\n" }
else { print "No esta definida \$tutu::a\n"};
print "$a\n";
```

6.2.2. Ejercicio: Subrutinas Locales

Tengo una subrutina en un package que no quiero que sea visible desde otros ficheros. ¿Que puedo hacer para conseguirlo? (Recuerde que las subrutinas son siempre globales).

6.3. Paquetes y ficheros

En principio no hay relación directa entre paquetes y ficheros: un mismo paquete puede existir en diversos ficheros y diversos paquetes pueden ser declarados en un único fichero.

Como un paquete generalmente se hace para ser reutilizado muchas veces, se guarda en un archivo *libreria* de extension `.pl` (por ejemplo `cgilib.pl`) o en un archivo *módulo* con el sufijo `.pm`.

Los programas que quieren usar pueden invocar el fichero con `require`

```
require "cgilib.pl";
```

la función `require` lee el archivo `cgilib.pl` si este no ha sido leido antes. El archivo no tiene que necesariamente estar asociado con un `package` pero si debe devolver verdadero, por tanto, es necesario que la última sentencia ejecutada sea:

`return 1;`

Si se omiten las comillas y el sufijo, se asume una extensión `.pm`.

Mientras que `require` carga el paquete en tiempo de ejecución, comprobando que no esta ya cargado, `use` carga los módulos en tiempo de compilación. El siguiente código no funciona:

```
$paquete = "MiPaquete"; # se ejecuta en tiempo de ejecución
use $paquete; # se ejecuta en tiempo de compilación
```

Además `use` requiere que el sufijo del fichero sea `.pm`

6.4. Búsqueda de librerías y módulos

El fichero conteniendo el módulo o librería debe colocarse en uno de varios directorios estándar en los que el compilador busca (la variable de entorno `PERL5LIB` gobierna la búsqueda, también puedes usar la opción `-I` del compilador). Esa lista esta disponible en un programa Perl a través de la variable `@INC`. Así puedes ver el camino de búsqueda escribiendo:

```
$ perl -e 'print "@INC\n";'
/usr/local/lib/perl/5.6.1 /usr/local/share/perl/5.6.1 /usr/lib/perl5
/usr/share/perl5 /usr/lib/perl/5.6.1 /usr/share/perl/5.6.1
/usr/local/lib/site_perl .
```

si tienes mas de una versión de Perl, puede que difieran en sus caminos de búsqueda:

```
$ perl5.8.0 -e 'print "@INC\n";'
/usr/local/lib/perl5/5.8.0/i686-linux /usr/local/lib/perl5/5.8.0
/usr/local/lib/perl5/site_perl/5.8.0/i686-linux
/usr/local/lib/perl5/site_perl/5.8.0 /usr/local/lib/perl5/site_perl .
```

Otra posibilidad es llamar a Perl con la opción `-V` (versión):

```
$ perl5.8.0 -V
Summary of my perl5 (revision 5.0 version 8 subversion 0) configuration:
...
Characteristics of this binary (from libperl):
  Compile-time options: USE_LARGE_FILES
  Built under linux
  Compiled at May 14 2003 16:02:03
  @INC:
    /usr/local/lib/perl5/5.8.0/i686-linux
    /usr/local/lib/perl5/5.8.0
    /usr/local/lib/perl5/site_perl/5.8.0/i686-linux
    /usr/local/lib/perl5/site_perl/5.8.0
    /usr/local/lib/perl5/site_perl
    .
```

El compilador sustituye cada `::` por el separador de caminos. Asi la orden: `use Text::ParseWords;` se traduce por el fichero `Text/ParseWords.pm`. En nereida, por ejemplo, el directorio exacto es `/usr/lib/perl5/5.00503/Text/ParseWords.pm`

El compilador Perl abre el primer fichero que case con `Text/ParseWords.pm` y evalúa el texto en su interior. Si falla, la compilación termina con un mensaje de error. en otro caso, el compilador busca en el módulo por una rutina denominada `import`, y si existe la ejecuta. Cuando esta termina la compilación continúa en el fichero original, justo después de la línea en la que aparece la sentencia `use`.

Si queremos especificar directorios adicionales de búsqueda podemos optar por una de estas opciones:

1. Utilizar la opción `-I` de la línea de comandos. Por ejemplo:

   ```
   perl -I/home/casiano/perl/src/packages/ -I/usr/local/test/perl first.pl
   ```

2. Definir la variable `PERL5LIB` como secuencia de caminos de acceso separados por el símbolo dos puntos (`:`)

3. Modificar la variable `@INC`:

   ```
   unshift(@INC, '/home/casiano/perl/src/packages/');
   require 'mypackage.pl';
   ```

4. Usar el módulo `lib` el cual añade los caminos especificados como argumentos en tiempo de compilación:

   ```
   #!/usr/local/bin/perl5.8.0 -w
   use lib qw(/home/lhp/perl/src /home/lhp/public_html/cgi-bin);
   print "@INC \n";
   ```

   Al ejecutar nos da:

   ```
   bash-2.05b$ ./use_lib.pl
   /home/lhp/perl/src /home/lhp/public_html/cgi-bin
   /usr/local/lib/perl5/5.8.0/i686-linux /usr/local/lib/perl5/5.8.0
   /usr/local/lib/perl5/site_perl/5.8.0/i686-linux
   /usr/local/lib/perl5/site_perl/5.8.0 /usr/local/lib/perl5/site_perl .
   ```

Si se quiere garantizar que el camino en cuestión esta disponible antes de que se ejecute ninguna sentencia se puede usar la opción 4. También podemos hacerlo “a mano” escribiendo un código como:

```
BEGIN {
  unshift @INC, "home/lhp/perl/src";
}
```

Cuando Perl esta en la fase de compilación y encuentra un bloque con nombre `BEGIN` pasa a ejecutarlo y continúa con la compilación. Puede existir mas de un bloque `BEGIN` en un programa, en cuyo caso se van ejecutando durante la fase de compilacion según se van viendo.

6.5. Control de Versiones

La sentencia `use` permite a un programa especificar que el módulo utilizado debe tener un número de versión no menor que un valor dado. Por ejemplo, si sabemos que para trabajar necesitamos versiones posteriores a la 1.5 del módulo `Biblio::Doc`, podríamos escribir:

```
use Biblio::Doc 1.5;
```

Esto hace que en el momento de la carga del módulo `Biblio::Doc` se ejecute automáticamente su subrutina `VERSION` (si existe) con argumento el número de versión. Existe una subrutina `VERSION` por defecto, que es proveída por el módulo `UNIVERSAL` (véase la sección 7.6.1). La rutina por defecto comprueba el valor en la variable `$VERSION` del paquete en cuestión.

6.6. Importación

A menudo queremos *importar* ciertos símbolos de un módulo en nuestro espacio de nombres, para ahorrarnos pulsaciones: queremos escribir `sqrt` y no `math::sqrt`.

Una vez que un módulo ha sido localizado y compilado dentro de un programa Perl como consecuencia de una declaración `use`, el siguiente paso es la ejecución de la subrutina `import` de ese módulo. De hecho, la sentencia `use module List` es equivalente a:

```
BEGIN {require module; import module List; }
```

La conducta por defecto de `import` es vacía, pero podemos cambiar dicha conducta creando en nuestro módulo nuestra propia subrutina `import`. Es decir, el módulo en cuestión tiene que estar preparado para *exportar* esos identificadores al código cliente que los utiliza. El uso de `BEGIN` implica que `require` e `import` se ejecuten en el momento de la compilación.

Cuando es llamada `import` recibe como argumentos cualesquiera argumentos que aparezcan despues de la sentencia `use`. Por ejemplo, si un programa incluye una línea como:

```
use Technique::DandC::FFT ("sample");
```

entonces, una vez localizado y compilado el módulo `Technique::DandC::FFT` se procede a la llamada de `import`:

```
Technique::DandC::FFT::import("Technique::DandC::FFT", "sample");
```

Existe una declaración `no` que puede ser usada para desechar las importaciones realizadas mediante `use`:

```
no integer;
no strict 'refs';
```

La desactivación se mantendrá en el ámbito léxico de la declaración `no`.

Ejercicio 6.6.1. *Queremos que la subrutina* `import` *del módulo* `myimport.pm` *que se define mas abajo exporte su función* `titi` *al espacio de nombres del package llamador, de manera que cuando se ejecute el siguiente programa* `usemyimport.pl`*:*

```
$ cat -n ./usemyimport.pl
#!/usr/bin/perl -w -I.
use strict;
use myimport;

my $titi = 4;
my @titi = (1,2,3);
&titi();

print"$titi\n";
print"@titi\n";
```

de lugar a la salida:

```
$ ./usemyimport.pl
Hola
4
1 2 3
```

para ello, deberás escribir el código que ha sido sustituido por las líneas de puntos en el listado de `myimport.pm`*.*

```
$ cat -n ./myimport.pm
package myimport;
use strict;

sub titi {
  print "Hola\n";
}

sub import {

  my ($caller_package) = ....; # averiguar paquete llamador
  {
    ...............; # desactivar strict
    ...............; # instalar identificador
  }
}

1;
```

SUGERENCIA:

- *La idea es que hay que instalar en la tabla de símbolos del paquete llamador (que no forzosamente es el paquete* `main`*) una entrada* `titi`*.*
- *Para instalar* `titi` *tiene que averiguar el nombre del paquete llamador. Recuerde que el módulo* `Carp` *provee funciones que nos permiten localizar quién llamo a la rutina actual. En concreto, la función* `caller` *devuelve el "package" desde el cuál fue llamada la subrutina.*
- *Para instalar la entrada* `titi` *en la tabla de símbolos del paquete llamador tendrá que usar typeglobs y referenciado simbólico.*
- *Minimize el periodo de desactivación de referenciado simbólico (*`no strict 'refs'`*).*
- *Observe el uso de la opción* `-I.` *en la primera línea del programa cliente* `usemyimport.pl`*: garantiza que el intérprete perl encontrará el módulo* `./myimport.pm`*.*
- *Si es necesario, repase las secciones 1.9.11, 5.14.9 y 5.11.*

6.7. Acceso a la tabla de símbolos

Perl permite acceder a la tabla de símbolos de un package `Toto` mediane un hash denominado `%Toto::`. Por ejemplo las variables del package `main` estan accesibles a través del hash `%main::` o también `%::`. Una estructura de este tipo recibe el nombre de *stash* (por Symbol Table Hash, la palabra stash tiene un significado similar a cache). De este modo, es sencillo mostrar los identificadores usados en un paquete:

```
foreach $id (keys %Toto::) {
  print $id,"\n";
}
```

Cada uno de las claves es una entrada de la tabla de símbolos. Los correspondientes valores son typeglobs, los cuales apuntan a los diferentes tipos: escalar, array, etc.

Consideremos el siguiente código que contiene dos paquetes: `main` y `toto`:

```
$ cat -n package_main.pl
     1  #!/usr/local/bin/perl5.8.0 -w
     2
     3  $a = 4;
     4
     5  package toto;
     6  $a = 8;
     7  $::a = 7;
     8
     9  package main;
    10  print $a, "\n";
    11  print $toto::a, "\n";
```

La ejecución con el depurador muestra la estructura de un stash:

```
$ perl -d package_main.pl
Loading DB routines from perl5db.pl version 1.25
main::(package_main.pl:3):      $a = 4; # a de main
  DB<1> n
toto::(package_main.pl:6):      $a = 8; # a de toto
  DB<1> n
toto::(package_main.pl:7):      $::a = 7;
  DB<1> n
main::(package_main.pl:10):     print $a, "\n";
  DB<1> x %toto::
0  'a'        # clave
1  *toto::a   # valor
  DB<2> x $toto::{a} # indexamos
0  *toto::a
  DB<3> x *{toto::a}{SCALAR} # typeglob usado como hash
0  SCALAR(0x8163978) # referencia a un escalar
   -> 8
  DB<4> x *{$toto::{a}}{SCALAR}
0  SCALAR(0x8163978)
   -> 8
  DB<5> *b = $toto::{a}
  DB<6> p $b
8
```

6.7.1. Práctica: Stash

Escriba una subrutina que vuelque los contenidos de las variables (escalar, array, hash) que están definidas en el paquete cuyo nombre se le pasa como parámetro a dicha subrutina. La capacidad de poder acceder y modificar la tabla de símbolos en tiempo de ejecución se denomina *introspección*. Para imprimir una estructura de datos compleja existen varias soluciones. Puede usar el módulo `Data::Dumper` para volcar las estructuras de datos resultantes o bien la función `dumpValue` de la librería `dumpvar.pl`. Véase la sección 5.7 para un ejemplo e uso.

6.8. Carga Automática

En la mayor parte de los lenguajes de programación, si se llama a una subrutina que no existe se produce inmediatamente un error. Perl proporciona un medio para crear una rutina "captura-llamadas" para cada paquete, la cuál será llamada siempre que la rutina solicitada no exista. Su nombre debe ser `AUTOLOAD`. Los parámetros que se le pasan a dicha subrutina serán los mismos que se pasaron a la subrutina desaparecida. Cuando se invoca a `AUTOLOAD`, la variable (del paquete) `$AUTOLOAD` contiene el nombre de la rutina solicitada. De este modo es posible conocer que rutina intentaba invocar el programa usuario.

El siguiente ejemplo proporciona una interfase funcional a los comandos del sistema operativo:

```
$ cat -n Ejautoload.pm
package Ejautoload;

sub AUTOLOAD {
  our $AUTOLOAD =~ s/.*:://; # suprimir el prefijo con el nombre del paquete
  my @result = `$AUTOLOAD @_`; # ejecutamos la funcion como un comando para
                           # la shell del sistema
  return @result;
}

sub import {
  my $mypackage = shift;

  my ($caller_package) = caller;
  {
    no strict 'refs';
    for my $subroutinename (@_) {
      *{$caller_package."::".$subroutinename} = \&{$subroutinename};
    }
  }
}

1;
```

Veamos el programa cliente:

```
$ cat -n command3.pl
#!/usr/bin/perl -w -I.
use Ejautoload('AUTOLOAD');

@files = ls("-l", "-t", "-r");
print @files[-5..-1];
```

El programa obtiene el listado de ficheros en orden inverso de tiempos y muestra los cinco ficheros mas recientemente modificados:

```
$ ./command3.pl
-rwxr-xr-x   1 lhp lhp    156 2005-05-05 18:12 stash_ex.pl
-rw-r--r--   1 lhp lhp    330 2005-05-06 08:28 command.pl
-rw-r--r--   1 lhp lhp    330 2005-05-06 08:30 command2.pl
-rwxr-xr-x   1 lhp lhp    107 2005-05-06 11:06 command3.pl
-rw-r--r--   1 lhp lhp    475 2005-05-06 11:07 Ejautoload.pm
```

Veamos otro ejemplo:

Ejemplo 6.8.1. *Tenemos en nuestro directorio un programa cliente* `jump.pl` *y un módulo* `Hops` *situado en el subdirectorio* `Modexample`*:*

```
$ tree
.
|-- CVS
|   |-- Entries
|   |-- Repository
|   `-- Root
|-- Modexample
|   `-- Hops.pm
`-- jump.pl
```

Siguen los contenidos del fichero `Modexample/Hops.pm`*:*

```
package Modexample::Hops;

sub hop_along {
  my ($from, $to, $step) = @_;
  my $next = $from-$step;  # incializar contador
  my $closure_ref = sub {
    $next += $step;
    return if $next > $to;
    $_[0] =$next;
    return 1;
  };
  return $closure_ref;
}

sub import {
  print("Modexample::Hops::import: argumentos: ",join(", ", @_),"\n\n");
  return 1;
}

sub AUTOLOAD {
  @_ = map { "\"".$_."\""; } @_;
  print "Has llamado a $AUTOLOAD(",join(", ",@_),") y no existe!\n";
}

1;
```

El programa cliente `jump.pl`*:*

```
#!/usr/bin/perl -w -I.
use Modexample::Hops qw(one two);

$row = Modexample::Hops::hop_along 1, 5, 1;
```

```
while ($row->($r)) {
  $col = Modexample::Hops::hop_along 1, 5, 1;
  while ($col->($c)) {
    print("($r, $c)\t");
  }
  print "\n";
}

Modexample::Hops::noexisto("pepe", "eowyn", "eomer", "maria");
```

Y un ejemplo de ejecución:

```
$ jump.pl
Modexample::Hops::import: argumentos: Modexample::Hops, one, two

(1, 1)  (1, 2)  (1, 3)  (1, 4)  (1, 5)
(2, 1)  (2, 2)  (2, 3)  (2, 4)  (2, 5)
(3, 1)  (3, 2)  (3, 3)  (3, 4)  (3, 5)
(4, 1)  (4, 2)  (4, 3)  (4, 4)  (4, 5)
(5, 1)  (5, 2)  (5, 3)  (5, 4)  (5, 5)
Has llamado a Modexample::Hops::noexisto("pepe", "eowyn", "eomer", "maria") y no existe!
```

6.9. Uso del Módulo de Exportación

Es un poco incómodo tener que prefijar los objetos del módulo con el nombre completo. Esto se puede obviar usando el módulo `Exporter`. De hecho para que un package pueda ser considerado un módulo no basta con que este en un fichero separado con sufijo `.pm`, debe tener (o heredar) un método de exportación y definir una lista de símbolos (que puede ser vacía) que son automáticamente exportados y/o una lista de símbolos (que puede ser vacía) que son exportados a petición.

6.9.1. Ejemplo de uso

Reescribamos el módulo del ejemplo anterior en un nuevo módulo `HopsExport`:

```
package Modexample::HopsExport;
use Exporter;

@ISA = ('Exporter');
@EXPORT = qw(&hop_along);

sub hop_along {
  my ($from, $to, $step) = @_;
  my $next = $from-$step;  # incializar contador
  my $closure_ref = sub {
    $next += $step;
    return if $next > $to;
    $_[0] =$next;
    return 1;
  };
  return $closure_ref;
}

1;
```

El módulo `Exporter` es el encargado de administrar la interfase pública del módulo. La inicialización del vector especial `@ISA` hace que el módulo `Modexample::HopsExport` "herede" del módulo `Exporter` los métodos que nos hacen falta como `import`. Aún cuando no hemos visto objetos, puede dar una ojeada a la sección 7.6 que trata sobre la herencia. El método `import` que proporciona `Exporter` examina la lista en `@EXPORT` para determinar que variables se exportan por defecto. ¿Como hacer si tenemos rutinas que no queremos que forme parte de la exportación por defecto pero que estén disponibles en caso de que se soliciten explícitamente? La alternativa es escribir los nombres de esos objetos en la lista `@EXPORT_OK`.

La línea de asignación a la variable `@EXPORT` hace que se cree un alias para la función `hop_along` en el programa cliente. De este modo no es necesario llamar a la función por su nombre completo `Modexample::HopsExport::hop_along` sino simplemente `hop_along`.

```
#!/usr/bin/perl -w -I.
use Modexample::HopsExport;

$row = hop_along 1, 5, 1;
while ($row->($r)) {
  $col = hop_along 1, 5, 1;
  while ($col->($c)) {
    print("($r, $c)\t");
  }
  print "\n";
}
```

Veamos otro ejemplo. Supongamos el módulo:

```
package Trivial::Tutu;
our @EXPORT = qw(uno dos);
our @EXPORT_OK = qw(tres cuatro cinco);
use Exporter;
our @ISA = qw(Exporter);
```

Los siguientes ejemplos ilustran el modo de uso de la exportación:

- `use Trivial::Tutu;`

  Esto nos exportaría `uno dos`.

- `use Trivial::Tutu qw(uno dos)`

  Lo mismo.

- `use Trivial::Tutu qw(tres cinco)`

  Ahora obtenemos `tres cinco`. No se importan `uno dos`.

- `use Trivial::Tutu();`

  No se importa ningún símbolo.

- `use Trivial::Tutu(siete);`

  Es un error. Todo símbolo importado debe estar bien en la lista `@EXPORT` bien en la lista `@EXPORT_OK`.

Observe que un programa cliente siempre puede acceder a un símbolo del módulo, incluso si no figura como símbolo exportable, sin mas que escribir su nombre completo (por ejemplo `Trivial::Tutu::siete`).

6.9.2. Formato de uso de Exporter

El módulo `Exporter` permite definir de manera precisa la interfase externa de nuestro módulo. Para ello deberemos escribir el siguiente código en `NuestroModulo.pm`

```
package NuestroModulo;
use strict;
use vars qw(@ISA @EXPORT @EXPORT_OK %EXPORT_TAGS $VERSION);

use Exporter;

$VERSION = 1.00;
@ISA = qw(Exporter);

@EXPORT = qw(...); # Símbolos a exportar
@EXPORT_OK = qw(...); # Símbolos a exportar a petición
%EXPORT_TAGS = (
  TAG1 => [...],
  TAG2 => [...],
  ...
);

######################
Nuestro código va aqui
######################

1;
```

En los ficheros desde los que queremos usar nuestro módulo deberemos escribir una de estas líneas:

```
use NuestroModulo;         # Importar los símbolos por defecto
use NuestroModulo qw(...); # Importar los símbolos listados
use NuestroModulo ();      # No importar símbolos
use NuestroModulo qw(:TAG1)# Importar el conjunto del tag
```

Cuando alguien escribe `use NuestroModulo`, ello implica un `require "NuestroModulo.pm"` seguido de una llamada a `NuestroModulo->import()` durante la compilación. El método `import`, que es heredado del módulo `EXPORTER` busca las variables globales que gobiernan la conducta de nuestro paquete. Puesto que son globales, es necesario declararlas con `ours` o bien utilizar el pragma `use vars` para satisfacer el uso de `use strict` sin que se produzcan mensajes de error. Estas variables son:

- `$VERSION`

  Se usa asi:

  ```
  use NuestroModulo 1.5  # Si  $VERSION < 1.5 error
  ```

- `@EXPORT`

  Contiene la lista de funciones y variables que serán exportadas por defecto al espacio de nombres del cliente.

- `EXPORT_OK`

  Este vector contiene los símbolos que serán cargados únicamente si se pregunta específicamente por ellos. Si los vectores se cargan asi:

  ```
  @EXPORT = qw(&F1 &F2 @List);
  @EXPORT_OK = qw(Op_Func %table);
  ```

Y el usuario carga el módulo como sigue:

```
use NuestroModulo qw(Op_Func %Table F1);
```

Entonces importamos las funciones `Op_func` y `F1` y el “hash” `%Table` pero no la función `F2` y el vector `@List`.

Uso de etiquetas: el hash `EXPORT_TAGS`

Si lo que se quiere es obtener todo lo que hay en `@EXPORT` además de los “extras” se deberá usar la etiqueta especial `DEFAULT`. Por ejemplo:

```
use NuestroModulo qw(:DEFAULT %Table)
```

- `%EXPORT_TAGS`

  Este “hash” es usado por módulos grandes como `CGI` o `POSIX` para crear grupos de alto nivel de símbolos relacionados. Por ejemplo:

```
%EXPORT_TAGS = (
  Functions => [ qw(F1 F2 Op_Func) ],
  Variables => [ qw(@List %Table)  ],
);
```

Un símbolo de la lista de importación precedido de dos puntos indica una etiqueta:

```
use NuestroModulo qw(:Functions %Table);
```

Como se ha dicho, el módulo `CGI.pm` funciona con esta filosofía. Véase el siguiente ejemplo que usa `CGI` en el que se carga el grupo `:standard`:

```
$ cat -n cgitaste.pl
 1  #!/usr/bin/perl -w
 2  use CGI qw(:standard);
 3
 4  print header;
 5  print start_html('Un ejemplo Sencillo'),
 6      h1('Un ejemplo Sencillo'),
 7      start_form,
 8      ">Tu nombre? ",textfield('nombre'),
 9      p,
10      ">Matriculado en?",
11      checkbox_group(-name=>'estudios',
12                     -values=>['Sistemas','Gestión','Superior'],
13                     -defaults=>['sistemas']),
14      p,
15      ">Lenguaje favorito? ",
16      popup_menu(-name=>'len',
17                 -values=>['C','C++','Pascal','Java','Lisp','Prolog','Python','Perl']),
18      p,
19      submit(-name=>"Enviar"),
20      end_form,
21      hr;
22
```

```
if (param()) {
    print h1('Tus datos:'),
        p,
        "Nombre: ",em(param('nombre')),
        p,
        "Estudios: ",em(join(", ",param('estudios'))),
        p,
        "Lenguaje favorito: ",em(param('len')),
        hr;
}
print end_html;
```

Puede ejecutar este ejemplo en http://nereida.deioc.ull.es/lhp-cgi/cgitaste.pl.

6.10. Práctica: AUTOLOAD

Escriba un módulo y su programa cliente. El módulo deberá proveer una función `AUTOLOAD` que emule a un conjunto de funciones que devuelvan el código HTML deseado. Así por ejemplo, la llamada:

```
A(href=>"nereida.deioc.ull.es", text=>"Visite nuestra página");
```

debería devolver la cadena:

```
<A href="nereida.deioc.ull.es">Visite nuestra página</A>
```

La llamada:

```
font(color=>red,text=>"Alerta!");
```

devolverá:

```
<font color="red">Alerta!</font>
```

etc.

Utilize la estrategia de paso de parámetros a través de un hash para tener argumentos con nombres (véase la sección 1.9.8)

6.11. Instalación de módulos

La Red de Archivos Perl Completa o Comprehensive Perl Archive network (CPAN, pronunciése ”sipan”) contiene la mayor colección de módulos Perl existente. Visite la página http://www.cpan.org/ o bien http://search.cpan.org/. Esta última esta exclusivamente dedicada a módulos, mientras que la primera contiene información adicional como programas Perl y las distribuciones de las diferentes versiones del compilador. Existen varios ”mirrors” en el mundo, de los cuales estos están en España:

- http://cpan.imasd.elmundo.es
- ftp://ftp.rediris.es/mirror/CPAN/
- ftp.ri.telefonica-data.net
- ftp://ftp.etse.urv.es/pub/perl/

6.11.1. Instalación a mano

Si no ha sido instalado aún el módulo `CPAN` - o no es el administrador del sistema - los pasos para una instalación son los siguientes:

Visite http://search.cpan.org/. Una vez allí, en el menu eliga la opción `distributions` o bien `all` y en la casilla de búsqueda ponga el nombre del módulo o el tópico de búsqueda. Se nos mostrarán todos los módulos para los que la búsqueda tiene éxito. Elija el módulo concreto que le interesa. La documentación POD del módulo es automaticamente convertida a HTML (figura 6.1). Si elige el enlace de la distribución en la cabecera, iremos al listado de ficheros que conforman la distribución. Esto nos permite ver el fichero `README` que va con la distribución así como la puntuación que ha obtenido (http://cpanratings.perl.org). Si se elije la opción `source` iremos al código fuente.

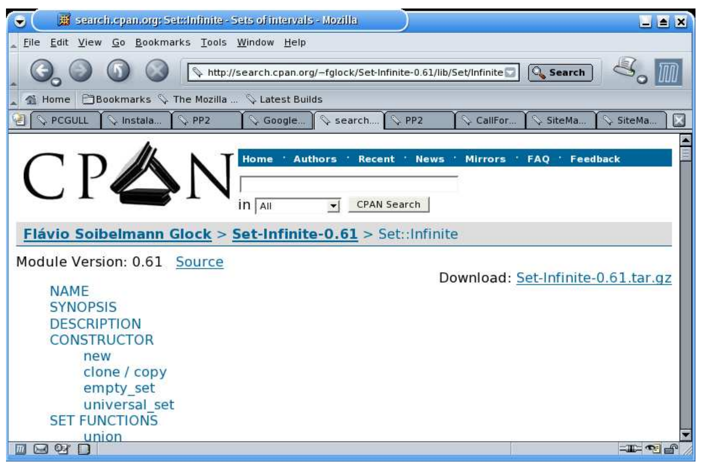

Figura 6.1: Buscando en CPAN

Presione sobre la opción `download` para descárgarlo en el directorio que desea. Después continúe como sigue:

```
> gunzip ese-modulo-6.12.tar.gz
> tar xf ese-modulo-6.12.tar
> cd ese-modulo-6.12
```

Si eres el administrador de tu sistema, puedes hacer:

```
> perl Makefile.PL
```

Si, por el contrario, quieres instalar los módulos en tu propio directorio, deberas escribir algo parecido a esto:

```
> perl Makefile.PL LIB=~/lib
```

Y si tienes tu propia distribución completa en local, algo asi:

```
> perl Makefile.PL PREFIX=~/perl5
```

En este caso asegúrate que el directorio en cuestión (`~/perl5` en el ejemplo) será encontrado por Perl cuando busca el módulo. Despues de esto deberás ejecutar el clásico *mantra de instalación*:

```
> make
> make test
> make install
```

6.11.2. Práctica: Instalar un Módulo

Instale en un directorio local el módulo `Memoize`. Recuerde pasar la opción adecuada cuando ejecute `perl Makefile.PL`. Compruebe que puede usar el módulo escribiendo un programa que *memoize* la función de Fibonacci recursiva tal y como aparece en la sección 5.16.5. Expanda el código en ese ejemplo para comparar los tiempos de la función memoizada a mano, la memoizada usando el módulo `Memoize` y la recursiva. No se olvide de usar la opción `-I` en la llamada al intérprete perl indicando el camino en el que se debe buscar el módulo o bien establecer la variable `PERL5LIB` al valor adecuado.

6.11.3. Usando el módulo `CPAN.pm` como Administrador

Una forma de instalar un módulo desde CPAN es utilizar el módulo `CPAN.pm`. Supuesto que ese módulo está instalado en tu máquina, la orden para usarlo es:

```
%perl -MCPAN -e shell
```

Si conoces el nombre del módulo a instalar, por ejemplo `Text::Balanced`, todo lo que tienes que hacer es escribir:

```
cpan> install Text::Balanced
```

y el modulo CPAN hará el resto.

La primera vez que se ejecuta el módulo `CPAN` se dispara un proceso de configuración que establece los valores por defecto que posibilitan la descarga, desempaquetado construcción y verificación de los modulos cuya instalacion has requerido.

En el siguiente ejemplo arrancamos el módulo `CPAN` como administradores del sistema:

```
# perl -MCPAN -e shell

cpan shell -- CPAN exploration and modules installation (v1.70)
ReadLine support enabled
```

Pedimos ayuda ...

```
cpan> help

Display Information
 command  argument          description
 a,b,d,m  WORD or /REGEXP/  about authors, bundles, distributions, modules
 i        WORD or /REGEXP/  about anything of above
 r        NONE              reinstall recommendations
 ls       AUTHOR            about files in the author's directory

Download, Test, Make, Install...
 get                        download
 make                       make (implies get)
 test      MODULES,         make test (implies make)
 install   DISTS, BUNDLES   make install (implies test)
 clean                      make clean
 look                       open subshell in these dists' directories
 readme                     display these dists' README files

Other
 h,?           display this menu       ! perl-code   eval a perl command
 o conf [opt]  set and query options   q             quit the cpan shell
 reload cpan   load CPAN.pm again      reload index  load newer indices
 autobundle    Snapshot                force cmd     unconditionally do cmd
```

Y pasamos a instalar el módulo `Switch` desarrollado por Conway, el cual nos proporciona una sentencia `switch` similar a la de C:

```
cpan> install Switch
CPAN: Storable loaded ok
Going to read /root/.cpan/Metadata
  Database was generated on Tue, 02 Dec 2003 06:46:46 GMT
CPAN: LWP::UserAgent loaded ok
Fetching with LWP:
  ftp://ftp.rediris.es/mirror/CPAN/authors/01mailrc.txt.gz
Going to read /root/.cpan/sources/authors/01mailrc.txt.gz
Fetching with LWP:
  ftp://ftp.rediris.es/mirror/CPAN/modules/02packages.details.txt.gz
Going to read /root/.cpan/sources/modules/02packages.details.txt.gz
  Database was generated on Thu, 18 Mar 2004 06:50:42 GMT

  There's a new CPAN.pm version (v1.76) available!
  [Current version is v1.70]
  You might want to try
    install Bundle::CPAN
    reload cpan
  without quitting the current session. It should be a seamless upgrade
  while we are running...

Fetching with LWP:
  ftp://ftp.rediris.es/mirror/CPAN/modules/03modlist.data.gz
Going to read /root/.cpan/sources/modules/03modlist.data.gz
Going to write /root/.cpan/Metadata
Running install for module Switch
Running make for R/RG/RGARCIA/Switch-2.10.tar.gz
Fetching with LWP:
  ftp://ftp.rediris.es/mirror/CPAN/authors/id/R/RG/RGARCIA/Switch-2.10.tar.gz
CPAN: Digest::MD5 loaded ok
Fetching with LWP:
  ftp://ftp.rediris.es/mirror/CPAN/authors/id/R/RG/RGARCIA/CHECKSUMS
Checksum for /root/.cpan/sources/authors/id/R/RG/RGARCIA/Switch-2.10.tar.gz ok
Scanning cache /root/.cpan/build for sizes
Deleting from cache: /root/.cpan/build/HTML-Tagset-3.03 (11.0>10.0 MB)
Deleting from cache: /root/.cpan/build/HTML-Parser-3.34 (10.9>10.0 MB)
Deleting from cache: /root/.cpan/build/HTML-Tree-3.18 (10.3>10.0 MB)
Switch-2.10/
Switch-2.10/t/
Switch-2.10/t/given.t
Switch-2.10/t/switch.t
Switch-2.10/t/nested.t
Switch-2.10/MANIFEST
Switch-2.10/META.yml
Switch-2.10/Changes
Switch-2.10/Switch.pm
Switch-2.10/README
Switch-2.10/Makefile.PL

  CPAN.pm: Going to build R/RG/RGARCIA/Switch-2.10.tar.gz
```

```
Checking if your kit is complete...
Looks good
Writing Makefile for Switch
cp Switch.pm blib/lib/Switch.pm
Manifying blib/man3/Switch.3
  /usr/bin/make  -- OK
Running make test
PERL_DL_NONLAZY=1 /usr/local/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/given.....ok
t/nested....ok
t/switch....ok
All tests successful.
Files=3, Tests=590,  5 wallclock secs ( 4.57 cusr +  0.06 csys =  4.63 CPU)
  /usr/bin/make test -- OK
Running make install
cpan> install SwitchInstalling /usr/local/lib/perl5/site_perl/5.8.0/Switch.pm
Installing /usr/local/man/man3/Switch.3
Writing /usr/local/lib/perl5/site_perl/5.8.0/i686-linux/auto/Switch/.packlist
Appending installation info to /usr/local/lib/perl5/5.8.0/i686-linux/perllocal.pod
  /usr/bin/make install  -- OK

cpan> Use of uninitialized value in ord at /usr/local/lib/perl5/site_perl/5.8.0/Term/ReadLine/readline.pm line 1481.
```

Ahora podemos usar `Switch`:

```
> cat useswitch.pl
#!/usr/local/bin/perl5.8.0 -w

use Switch;

$val = shift;

switch ($val) {
  case 1 { print "number 1\n" }
  case "hello" { print "string hello\n" }
  else { print " none of these two\n" }
}
$ useswitch.pl "hello"
string hello
$ useswitch.pl "hola"
 none of these two
```

6.11.4. Opciones de Configuración y Otras Opciones

`urllist`

Es posible consultar las opciones actuales:

```
cpan> o conf urllist
    urllist
        ftp://archive.progeny.com/CPAN/
        ftp://cpan-sj.viaverio.com/pub/CPAN/
        ftp://cpan.calvin.edu/pub/CPAN
        ftp://cpan.cs.utah.edu/pub/CPAN/
        ftp://cpan.digisle.net/pub/CPAN
```

```
        ftp://cpan.erlbaum.net/
        ftp://cpan.llarian.net/pub/CPAN/
```

Y si nos conviene, modificarlas:

```
cpan> o conf urllist unshift ftp://ftp.rediris.es/mirror/CPAN/
cpan> o conf urllist
    urllist
        ftp://ftp.rediris.es/mirror/CPAN/
        ftp://archive.progeny.com/CPAN/
        ftp://cpan-sj.viaverio.com/pub/CPAN/
        ftp://cpan.calvin.edu/pub/CPAN
        ftp://cpan.cs.utah.edu/pub/CPAN/
        ftp://cpan.digisle.net/pub/CPAN
        ftp://cpan.erlbaum.net/
        ftp://cpan.llarian.net/pub/CPAN/
Type 'o conf' to view configuration edit options
```

Ahora:

```
cpan> install Email::Find
Running install for module Email::Find
Running make for M/MI/MIYAGAWA/Email-Find-0.09.tar.gz
CPAN: LWP::UserAgent loaded ok
Fetching with LWP:
  ftp://ftp.rediris.es/mirror/CPAN/authors/id/M/MI/MIYAGAWA/Email-Find-0.09.tar.gz
....
```

`keep_source_where`

El directorio en el que se guardan las distribuciones puede obtenerse asi:

```
cpan> o conf keep_source_where
    keep_source_where  /root/.cpan/sources
```

La estructura del directorio se organiza por autores:

```
~/.cpan/sources# tree
.
|-- MIRRORED.BY
|-- authors
|   |-- 01mailrc.txt.gz
|   |-- 01mailrc.txt.gz.bak
|   `-- id
|       |-- A
|       |   |-- AF
|       |   |   `-- AFERBER
|       |   |       |-- CHECKSUMS
|       |   |       `-- Thread-RWLock-1.02.tar.gz
|       |   |-- AM
|       |   |   `-- AMS
|       |   |       |-- CHECKSUMS
|       |   |       `-- Storable-2.13.tar.gz
|       |   `-- AR
|       |       `-- AREIBENS
|       |           |-- CHECKSUMS
```

```
|       |           `-- PDF-API2-0.3r77.tar.gz
|       |-- B
|       |   |-- BD
|       |   |   `-- BDARRAH
|       |   |       |-- CHECKSUMS
|       |   |       `-- Proc-ParallelLoop-0.5.tgz
........... etc.
|       |-- V
|       |   `-- VI
|       |       `-- VIPUL
|       |           |-- CHECKSUMS
|       |           |-- Class-Loader-2.02.tar.gz
|       |           |-- Convert-ASCII-Armour-1.4.tar.gz
|       |           |-- Crypt-Primes-0.50.tar.gz
|       |           |-- Crypt-RSA-1.50.tar.gz
|       |           |-- Crypt-Random-1.23.tar.gz
|       |           `-- Tie-EncryptedHash-1.21.tar.gz
|       |-- f
|       |   `-- f
|       |       `-- f
|       |           `-- ft
|       |               `-- ftp:
|       |                   `-- sunsite.rediris.es
|       |                       `-- mirror
|       |                           `-- CPAN
|       |                               `-- modules
|       |                                   `-- by-module
|       |                                       `-- Class
|       |                                           `-- Class-MethodMaker-2.04-1.tar
|       `-- t
|           `-- t
|               `-- t
|                   `-- tm
|                       `-- tmp
|                           `-- CPAN
`-- modules
    |-- 02packages.details.txt.gz
    |-- 02packages.details.txt.gz.bak
    |-- 03modlist.data.gz
    `-- 03modlist.data.gz.bak
```

Bundles

Un *bundle* es un tipo de objeto CPAN que nos permite la fácil instalación de un conjunto de módulos relacionados. El modulo CPAN detecta que nos estamos refiriendo a un bundle porque siempre van prefijados por `Bundle::`. Uno de los usos de los bundles es combinarlo con el comando `autobundle`:

```
cpan> autobundle

Package namespace         installed    latest  in CPAN file
AnyDBM_File                    1.00      1.00  N/NW/NWCLARK/perl-5.8.6.tar.gz
Apache                         1.27      1.27  G/GO/GOZER/mod_perl-1.29.tar.gz
Apache::Connection             1.00      1.00  G/GO/GOZER/mod_perl-1.29.tar.gz
Apache::Constants              1.09      1.09  G/GO/GOZER/mod_perl-1.29.tar.gz
```

```
Apache::Constants::Exports     undef     undef  G/GO/GOZER/mod_perl-1.29.tar.gz
Apache::Debug                  1.61      1.61  G/GO/GOZER/mod_perl-1.29.tar.gz
Apache::ExtUtils               1.04      1.04  G/GO/GOZER/mod_perl-1.29.tar.gz
....
Wrote bundle file
    /root/.cpan/Bundle/Snapshot_2005_05_11_00.pm
```

Esto permite la creación automática de un bundle que congela la lista de módulos en nuestro sistema. Si ahora queremos tener una instalación de Perl con los mismos módulos que esta, sólo tendremos que renombrar el bundle (p.ej. `Bundle/mybundle.pm` e instalar el bundle `install Bundle::mybundle.pm`.

6.11.5. CPAN: Si no tenemos los privilegios de administrador

En las versiones mas recientes de Perl que hacen uso de las versiones mas modernas de `CPAN.pm` esto no es un problema. Simplemente escriba `perl -MCPAN -e shell` y responda al interrogatorio para determinar la configuración que necesita.

En el caso de que la versión con la que trabajemos sea antigua y no tengamos los privilegios de administrador del sistema tendremos que trabajar un poco para usar el módulo `CPAN`:

```
 1 home/casiano[23]> uname -a
 2 Linux millo.etsii.ull.es 2.4.22-1.2188.nptl #1 \
    Wed Apr 21 20:36:05 EDT 2004 i686 i686 i386 GNU/Linux
 3 /home/casiano[5]> perl -MCPAN -e shell
 4 Terminal does not support AddHistory.
 5
 6 Your configuration suggests "/root/.cpan" as your
 7 CPAN.pm working directory. I could not create this directory due
 8 to this error: mkdir /root/.cpan: Permiso denegado at \
              /usr/local/lib/perl5/5.8.5/CPAN.pm line 553
 9
10
11 Please make sure the directory exists and is writable.
12
```

Lo primero que obtenemos es un fracaso: Obviamente no podemos modificar el fichero `/root/.cpan`. Por tanto deberemos indicar que este no es el fichero de configuración:

```
13 /home/casiano[6]> mkdir .cpan
14 /home/casiano[7]> mkdir .cpan/CPAN
15 /home/casiano[9]> perl -MCPAN::Config -e 'print $INC{"CPAN/Config.pm"},"\n"'
16 /usr/local/lib/perl5/5.8.5/CPAN/Config.pm
```

En la línea 15 ejecutamos un guión interactivo (`-e`) que carga el módulo `CPAN::Config` (opción `-M`) y que hace uso del hash `%INC`.

```
17 /home/casiano[10]> cp /usr/local/lib/perl5/5.8.5/CPAN/Config.pm .cpan/CPAN/MyConfig.pm
18 /home/casiano[11]> cd .cpan/CPAN/
19 /home/casiano/.cpan/CPAN[12]> perl -ne 'print if /(build_dir|keep_source_where)/' MyConfig.pm
20   'build_dir' => q[/root/.cpan/build],
21   'keep_source_where' => q[/root/.cpan/sources],
```

La opción `-n` en la línea 19 envuelve en un bucle de lectura el guión interactivo. Muestra las líeas del fichero que casan con la expresión regular. Ahora editamos el fichero y cambiamos las líneas que hacen alusión al `root`:

```
22 /home/casiano/.cpan/CPAN[13]> vi MyConfig.pm
    .... # un poco después ...
```

```
23 /home/casiano/.cpan/CPAN[16]> perl -ne 'print if /(casiano)/' MyConfig.pm
24   'build_dir' => q[/scratch/casiano/build],
25   'cpan_home' => q[/scratch/casiano/.cpan],
26   'histfile' => q[/scratch/casiano/.cpan/histfile],
27   'keep_source_where' => q[/scratch/casiano/.cpan/sources],
```

El comando nos muestra que líneas fueron cambiadas en la edición. Ahora estamos en condiciones de ejecutar CPAN, pero tenemos que hacer aún algunos cambios en la configuración:

```
28 /home/casiano/.cpan/CPAN[19]> perl -MCPAN -e shell
29 Terminal does not support AddHistory.
30
31 cpan shell -- CPAN exploration and modules installation (v1.7601)
32 ReadLine support available (try 'install Bundle::CPAN')
33 cpan> o conf makepl_arg PREFIX=/scratch/casiano/perl
34     makepl_arg         PREFIX=/scratch/casiano/perl
35 cpan> o conf commit
36 commit: wrote /home/casiano/.cpan/CPAN/MyConfig.pm
```

En las líneas 33 y 35 le hemos indicado a CPAN donde debe dejar los módulos descargados. Deberemos asegurarnos que los programas que usen esos módulos tengan dicho directorio en su @INC. Ahora podemos instalar un módulo:

```
37 cpan> install Parse::Yapp
38 CPAN: Storable loaded ok
39 Going to read /scratch/casiano/.cpan/Metadata
40 ....
41 Appending installation info to /scratch/casiano/perl/lib/perl5/5.8.5/i686-linux/perllocal.pod
42   /usr/bin/make install  -- OK
```

Veamos la jerarquía creada por la instalacion de Parse::Yapp

```
$ tree /scratch/casiano/perl/
/scratch/casiano/perl/
|-- bin
|   `-- yapp
|-- lib
|   `-- perl5
|       |-- 5.8.5
|       |   `-- i686-linux
|       |       `-- perllocal.pod
|       `-- site_perl
|           `-- 5.8.5
|               |-- Parse
|               |   |-- Yapp
|               |   |   |-- Driver.pm
|               |   |   |-- Grammar.pm
|               |   |   |-- Lalr.pm
|               |   |   |-- Options.pm
|               |   |   |-- Output.pm
|               |   |   `-- Parse.pm
|               |   `-- Yapp.pm
|               `-- i686-linux
|                   `-- auto
|                       `-- Parse
|                           `-- Yapp
```

```
`-- share
    `-- man
        |-- man1
        |   `-- yapp.1
        `-- man3
            `-- Parse::Yapp.3

17 directories, 11 files
```

Aún mas secillo: use el módulo `CPAN::FirstTime`:

```
$ perl -MCPAN::FirstTime -e 'CPAN::FirstTime::init()'
CPAN is the world-wide archive of perl resources. It consists of about
100 sites that all replicate the same contents all around the globe.
Many countries have at least one CPAN site already. The resources
found on CPAN are easily accessible with the CPAN.pm module. If you
want to use CPAN.pm, you have to configure it properly.

If you do not want to enter a dialog now, you can answer 'no' to this
question and I'll try to autoconfigure. (Note: you can revisit this
dialog anytime later by typing 'o conf init' at the cpan prompt.)

Are you ready for manual configuration? [yes] yes


The following questions are intended to help you with the
configuration. The CPAN module needs a directory of its own to cache
important index files and maybe keep a temporary mirror of CPAN files.
This may be a site-wide directory or a personal directory.



I see you already have a  directory
    /root/.cpan
Shall we use it as the general CPAN build and cache directory?

CPAN build and cache directory? [/root/.cpan]
.... # etc.
```

6.11.6. Práctica: CPAN

Reconfigure `CPAN` para trabajar como usuario ordinario.

6.12. Construcción de un Módulo con `h2xs`

Una forma de comenzar a escribir un módulo es usando la herramienta Perl `h2xs`. Supongamos que queremos construir un módulo `Parse::Yard`. La orden es:

```
$ h2xs -XA -n Parse::Yard
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 Parse-Yard/lib/Parse/Yard.pm
Writing Parse-Yard/Makefile.PL
```

```
Writing Parse-Yard/README
Writing Parse-Yard/t/Parse-Yard.t
Writing Parse-Yard/Changes
Writing Parse-Yard/MANIFEST
```

Lo cual crea la siguiente estructura de directorios y ficheros:

```
$ cd Parse-Yard/
Parse-Yard$ tree
.
|-- Changes
|-- MANIFEST
|-- Makefile.PL
|-- README
|-- lib
|   `-- Parse
|       `-- Yard.pm
`-- t
    `-- Parse-Yard.t
```

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.

Después de esto tenemos un módulo "funcional" que no hace nada. Lo podríamos instalar como si lo hubieramos descargado desde CPAN (para saber más sobre el proceso de instalación, véase 6.11.1).

`Makefile.PL` es un programa Perl, que se encarga de crear el `Makefile`:

```
$ cat -n 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              => 'Parse::Yard',
 VERSION_FROM      => 'lib/Parse/Yard.pm', # finds $VERSION
 PREREQ_PM         => {}, # e.g., Module::Name => 1.1
 ($] >= 5.005 ?     ## Add these new keywords supported since 5.005
   (ABSTRACT_FROM  => 'lib/Parse/Yard.pm', # retrieve abstract from module
    AUTHOR         => 'Lenguajes y Herramientas de Programacion <lhp@ull.es>') : ()),
);
```

El programa usa el módulo `ExtUtils::MakeMaker` que proporciona la subrutina `WriteMakefile`, la cual se encarga de construir el `Makefile` de acuerdo a las especificaciones que se le pasan como parámetros:

- `NAME` es el nombre del módulo
- La clave `VERSION_FROM` dice que fichero contiene la variable `$VERSION` que define la versión de esta distribución.
- La entrada `PREREQ_PM` es una referencia a un hash cuyas claves son los nombres de los módulos de los que depende y los valores son los números de versión requeridos. Por ejemplo:

  ```
  PREREQ_PM => { LWP::UserAgent => 1.73, HTTP::Request => 1.27 }
  ```

- `ABSTRACT_FROM` indica el fichero que contiene la descripción del módulo.

- ABSTRACT es una línea describiendo el módulo. Se incluye en el fichero *PPD* (*Perl Package Descriptor*). Los ficheros PPD son ficheros XML.

Por tanto, para crear el Makefile basta ejecutar este programa escribiendo perl Makefile.PL

```
Parse-Yard$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Parse::Yard
Parse-Yard$ ls -ltr
total 44
drwxr-xr-x  2 lhp lhp  4096 2004-12-23 09:47 t
-rw-r--r--  1 lhp lhp  1202 2004-12-23 09:47 README
-rw-r--r--  1 lhp lhp    69 2004-12-23 09:47 MANIFEST
-rw-r--r--  1 lhp lhp   561 2004-12-23 09:47 Makefile.PL
drwxr-xr-x  3 lhp lhp  4096 2004-12-23 09:47 lib
-rw-r--r--  1 lhp lhp   158 2004-12-23 09:47 Changes
-rw-r--r--  1 lhp lhp 19549 2004-12-23 16:34 Makefile
```

En primer lugar se ha comprobado si nuestra aplicación está completa, para ello WriteMakefile mira la lista de ficheros que figura en el fichero MANIFEST y comprueba que todos los ficheros en la lista están presentes. Los contenidos de MANIFEST son:

```
$ cat MANIFEST
Changes
Makefile.PL
MANIFEST
README
t/Parse-Yard.t
lib/Parse/Yard.pm
```

Una de las opciones proporcionadas por MakeMaker es make dist el cual también usa el fichero MANIFEST para determinar que ficheros forman parte de la distribución. Es importante mantener este fichero actualizado. MakeMaker también proporciona un objetivo MANIFEST. Tecleando make MANIFEST se crea un fichero MANIFEST que contiene todos los directorios y subdirectorios del directorio actual. Esta conducta puede modificarse creando un fichero MANIFEST.SKIP el cual especifica mediante expresiones regulares que tipos de ficheros no deben ser incluidos en el MANIFEST. Por ejemplo:

```
$ cat -n /usr/local/src/parrot-0.1.1/MANIFEST.SKIP
     1  \.o$
     2  ^\.cvsignore$
     3  /\.cvsignore$
     4  \.cvsignore$
     5  CVS/[^/]+$
     6  ^include/parrot/config\.h$
     7  ^include/parrot/has_header\.h$
     8  ^include/parrot/platform\.h$
     9  ^Makefile$
    10  /Makefile$
    11  ^lib/Parrot/Config\.pm$
    12  ^platform\.c$
    ..................
```

El programa h2xs creó un esqueleto de módulo para nosotros. Estos son sus contenidos:

```
package Parse::Yard;
use 5.008004;
use strict;
use warnings;
require Exporter;
our @ISA = qw(Exporter);

# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.

# This allows declaration use Parse::Yard ’:all’;
# If you do not need this, moving things directly into @EXPORT or @EXPORT_OK
# will save memory.
our %EXPORT_TAGS = ( ’all’ => [ qw( ) ] );
our @EXPORT_OK = ( @{ $EXPORT_TAGS{’all’} } );
our @EXPORT = qw( );
our $VERSION = ’0.01’;

# Preloaded methods go here.
1;
__END__
# Below is stub documentation for your module. You’d better edit it!

=head1 NAME

Parse::Yard - Perl extension for blah blah blah

=head1 SYNOPSIS

  use Parse::Yard;
  blah blah blah

=head1 DESCRIPTION

Stub documentation for Parse::Yard, created by h2xs. It looks like the
author of the extension was negligent enough to leave the stub
unedited.

Blah blah blah.

=head2 EXPORT

None by default.

=head1 SEE ALSO

Mention other useful documentation such as the documentation of
related modules or operating system documentation (such as man pages
in UNIX), or any relevant external documentation such as RFCs or
standards.

If you have a mailing list set up for your module, mention it here.
```

```

If you have a web site set up for your module, mention it here.

=head1 AUTHOR

Lenguajes y Herramientas de Programacion, E<lt>lhp@E<gt>

=head1 COPYRIGHT AND LICENSE

Copyright (C) 2004 by Lenguajes y Herramientas de Programacion

This library is free software; you can redistribute it and/or modify
it under the same terms as Perl itself, either Perl version 5.8.4 or,
at your option, any later version of Perl 5 you may have available.

=cut
```

Ahora podemos hacer `make`:

```
$ make
cp lib/Parse/Yard.pm blib/lib/Parse/Yard.pm
Manifying blib/man3/Parse::Yard.3pm
```

Como consecuencia la estructura del proyecto ha cambiado:

Antes de `make`

```
$ cd Parse-Yard/
Parse-Yard$ tree
$ tree
.
|-- Changes
|-- MANIFEST
|-- Makefile
|-- Makefile.PL
|-- README
|-- lib
|   `-- Parse
|       `-- Yard.pm
`-- t
    `-- Parse-Yard.t
```

Después de `make`

```
$ tree
.
|-- Changes
|-- MANIFEST
|-- Makefile
|-- Makefile.PL
|-- README
|-- blib
|   |-- arch
|   |   `-- auto
|   |       `-- Parse
|   |           `-- Yard
|   |-- lib
|   |   |-- Parse
|   |   |   `-- Yard.pm
|   |   `-- auto
|   |       `-- Parse
|   |           `-- Yard
|   `-- man3
|       `-- Parse::Yard.3pm
|-- lib
|   `-- Parse
|       `-- Yard.pm
|-- pm_to_blib
`-- t
    `-- Parse-Yard.t
```

Para comprobar que el módulo funciona se hace `make test`:

```
$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" \
    "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/Parse-Yard....ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.04 cusr +  0.01 csys =  0.05 CPU)
```

Una vez escritas las diferentes componentes del módulo, podemos instalarlo (haciendo `make install`. Normalmente esto requeriría privilegios de superusuario). También podemos construir una versión para su distribución mediante los comandos:

```
> perl Makefile.PL # Crea el Makefile
> make dist # o bien: make tardist
```

6.13. La Documentación en Perl

La documentación en Perl se hace en formato `pod` (*plain old documentation*). Esta documentación se inserta en nuestro programa. Se trata de un lenguaje de marcas. Las líneas 25 a 69 en el ejemplo de la sección 6.12 muestran el uso de `pod`. Cuando Perl encuentra una línea que comienza con un signo igual, ignora todas las líneas hasta encontrar una línea que comienze con `=cut`.

Existe un buen número de utilidades que permiten pasar de formato `pod` a casi cualquier formato: `pod2html`, `pod2man`, `pod2latex`, `pod2ps`, `pod2text`, `pod2pdf.pl`, etc. Un traductor `pod-to-XXX` lee un fichero en formato `pod` párrafo por párrafo y lo traslada al formato `XXX`. Por ejemplo, para convertir a LaTeX la documentación de `perlpod` puedes escribir algo como:

`pod2latex -full /usr/share/perl/5.6.1/pod/perlpod.pod`

Si se ha instalado la documentación de Perl en tu sistema, puedes acceder a ella utilizando `info`, `man` o `perldoc`. La orden `man perl` te mostrará las diferentes secciones existentes y sobre que tratan. Asi `man perlboot` te introducirá en la programación orientada a objetos en Perl.

Para saber mas sobre el lenguaje de marcas `pod` escribe `man perlpod`.

Un documento POD consiste de párrafos separados por líneas en blanco. Existen tres tipos de párrafos: verbatim, comandos y texto.

6.13.1. Texto verbatim

Cuando un párrafo esta sangrado (comienza con un espacio o tabulador) se reproduce exactamente como aparece.

6.13.2. Un comando de párrafo

```
=head1 cabecera
=head2 cabecera
=item texto
=over N
=back
=cut
=pod
=for X
=begin X
=end X
```

- `=pod`, `=cut`

  Es útil cuando mezclas código y documentación. Le indica a Perl que lo que sigue es documentación hasta que se alcance el siguiente `=cut`.

- =head1, =head2

  Primer y segundo nivel de encabezamiento. El texto cabecera debe estar en el mismo párrafo que la directiva.

- =over, =back, =item

  =over comienza una lista, =back la termina y los elementos de la lista se especifican con =item. El número que se le pasa a =over indica el sangrado de los items. Veamos un ejemplo:

```
=over 4

=item *

Bla, bla, primer item

=item *

Mas bla, bla, segundo item

=back
```

  Mantenga la consitencia en la presentación: use =item * para producir puntos o bien la forma =item 1., =item 2., etc.

- =for, =begin, =end Nos permiten incluir secciones que no contienen texto POD sino que van dirigidas a formatos especiales. La directiva =for tiene como ámbito el siguiente párrafo:

```
=for html <br>
 <p> Esto sólo aparece en HTML</p>
```

  Mientras que =begin y =end delimitan el texto en cuestión.

  Ademas de html, otros nombres que se aceptan son roff, man, latex, tex y text.

6.13.3. Texto normal

El texto será formateado y se pueden usar secuencias de formateado como

- I<texto> itálicas
- B<texto> negritas
- C<texto> código literal
- S<texto> texto que no se puede romper por los espacios.
- F<fichero> usado para nombres de fichero
- L<texto> un enlace a un nombre en el manual.
- E<escape>: E<lt> E<gt> E<n> (n es un número)

6.14. Bancos de Pruebas y Extreme Programming

Queremos comprobar si nuestro código funciona. ¿Cómo hacerlo?. Lo adecuado es llevar una aproximación sistemática que permita validar el código. Esa es la función del programa `test.pl` que se genera automáticamente con `h2xs`.

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 [12]:

- 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

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 [12].

6.14.1. Versiones anteriores a la 5.8

En esta sección he usado la versión 5.6.1 de Perl. Creemos 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`?

6.14.2. Versiones posteriores a la 5.8

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`. El programa `test.pl` desaparece. La estructura queda como sigue:

```
$ perl -V | grep -i summary
Summary of my perl5 (revision 5 version 8 subversion 4) configuration:
$ h2xs -AXn 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
$ tree
.
`-- PL-Tutu
    |-- Changes
    |-- MANIFEST
    |-- Makefile.PL
    |-- README
    |-- lib
    |   `-- PL
    |       `-- Tutu.pm
    `-- t
        `-- PL-Tutu.t

4 directories, 6 files
```

Nos cambiamos al directorio:

```
$ cd PL-Tutu/
$ ls -l
total 24
-rw-r--r--  1 lhp lhp  154 Nov  3 12:59 Changes
-rw-r--r--  1 lhp lhp   63 Nov  3 12:59 MANIFEST
-rw-r--r--  1 lhp lhp  552 Nov  3 12:59 Makefile.PL
-rw-r--r--  1 lhp lhp 1196 Nov  3 12:59 README
drwxr-xr-x  3 lhp lhp 4096 Nov  3 12:59 lib
drwxr-xr-x  2 lhp lhp 4096 Nov  3 12:59 t
$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for PL::Tutu
$ ls -ltr
total 44
drwxr-xr-x  2 lhp lhp  4096 Nov  3 12:59 t
drwxr-xr-x  3 lhp lhp  4096 Nov  3 12:59 lib
-rw-r--r--  1 lhp lhp  1196 Nov  3 12:59 README
-rw-r--r--  1 lhp lhp   552 Nov  3 12:59 Makefile.PL
-rw-r--r--  1 lhp lhp    63 Nov  3 12:59 MANIFEST
-rw-r--r--  1 lhp lhp   154 Nov  3 12:59 Changes
-rw-r--r--  1 lhp lhp 19471 Nov  3 13:03 Makefile
```

Ahora podemos ejecutar el primer test:

```
$ make test
cp lib/PL/Tutu.pm blib/lib/PL/Tutu.pm
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/PL-Tutu....ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.04 cusr +  0.02 csys =  0.06 CPU)
```

Observe que, como consecuencia de esta primera ejecución se han creado nuevos directorios:

```
$ tree
.
|-- Changes
|-- MANIFEST
|-- Makefile
|-- Makefile.PL
|-- README
|-- blib
|   |-- arch
|   |   `-- auto
|   |       `-- PL
|   |           `-- Tutu
|   |-- lib
|   |   |-- PL
|   |   |   `-- Tutu.pm
|   |   `-- auto
|   |       `-- PL
|   |           `-- Tutu
|   `-- man3
|-- lib
|   `-- PL
|       `-- Tutu.pm
|-- pm_to_blib
`-- t
    `-- PL-Tutu.t

14 directories, 9 files
```

Los contenidos del fichero de prueba son similares al de las versiones previas:

```
$ cat t/PL-Tutu.t
# Before `make install' is performed this script should be runnable with
# `make test'. After `make install' it should work as `perl PL-Tutu.t'

#########################

# change 'tests => 1' to 'tests => last_test_to_print';

use Test::More tests => 1;
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.
```

Sólo que en vez de usar el módulo `Test` se usa el módulo `Test::More` el cual, como su nombre indica nos provee con un variado conjunto de funciones de prueba. Para mas información escriba `perldoc Test::More`.

Ahora ampliamos la batería de pruebas con una prueba mas `02lex.t`:

```
$ pwd
/home/lhp/perl/src/topdown/PL/5.8/PL-Tutu/t
$ ls -l
total 8
```

```
-rw-r--r--  1 lhp lhp 446 Nov  3 15:03 02lex.t
-rw-r--r--  1 lhp lhp 465 Nov  3 13:11 PL-Tutu.t
```

Pero antes renombramos el fichero `PL-Tutu.t`:

```
$ mv PL-Tutu.t 01load.t
```

Esto lo hacemos con dos objetivos en mente:

- Que el nombre del fichero sea significativo del tipo de prueba
- Que los prefijos de los nombres `01`, `02`, . . . nos garanticen el orden de ejecución

El programa `02lex.t` pone a prueba el analizador léxico:

```
$ cat 02lex.t
# change 'tests => 1' to 'tests => last_test_to_print';

use Test::More tests => 2;
BEGIN { use_ok('PL::Tutu') };

#########################

my $a = 'int a,b; string c; c = "hello"; a = 4; b = a +1; p b';
local @PL::Tutu::tokens = ();
Lexical::Analysis::scanner($a);
ok("@PL::Tutu::tokens" eq
'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');
```

Ahora probamos de nuevo los tests:

```
$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM"
                  "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/01load....ok
t/02lex.....ok
All tests successful.
Files=2, Tests=3,  0 wallclock secs ( 0.15 cusr +  0.02 csys =  0.17 CPU)
```

6.15. Práctica: Construcción de una Distribución

Reconstruya siguiendo los pasos descritos en las secciones 6.12 y 6.13 el módulo descrito en la práctica 6.10 en el que se proveía una función `AUTOLOAD` que emulaba a un conjunto de funciones. Compruebe que puede instalar el módulo como si se hubiera descargado desde CPAN siguiendo los pasos descritos en la sección 6.11.1.

6.16. Pruebas en la Construcción de una Distribución

Construiremos una distribución de un módulo que resuelve utilizando *Programación Dinámica* el *Problema de la Mochila 0-1*.

En la dirección:

http://nereida.deioc.ull.es/~lhp/perlexamples/Algorithm-Knap01DP-0.01.tar.gz.
puede encontrar una distribución del módulo presentado en esta sección.

A continuación presentaremos el problema, el algoritmo que lo resuelve y estudiaremos en detalle cada una de las partes que integran el módulo.

6.16.1. El Problema de la Mochila 0-1

El problema de la mochila 0-1 se enuncia asi: Se tiene una mochila de de capacidad M y N objetos de pesos w_k y beneficios p_k. Se trata de encontrar la combinación óptima de objetos que caben en la mochila y producen el máximo beneficio.

Para resolverlos denotemos por $f_k(c)$ el beneficio óptimo obtenido usando los primeros k objetos y una mochila de capacidad c. Entonces $f_0(c) = p_0$ si $c \geq w_0$ y 0 en otro caso. Además

$$f_k(c) = \max\{f_{k-1}(c), f_{k-1}(c - w_k) + p_k\} \tag{6.1}$$

si $c > w_k$ y $f_k(c) = f_{k-1}(c)$ en otro caso.

El esquema del algoritmo es:

```
 1   for my $c (0..$M) {
 2     $f[0][$c] = ($w[0] <= $c)? $p[0] : 0;
 3   }
 4
 5   for my $k (1..$N-1) {
 6     for my $c (0..$M) {
 7       my $n = $f[$k-1][$c];
 8       if ($c >= $w[$k]) {
 9         my $y = $f[$k-1][$c-$w[$k]]+$p[$k];
10         $f[$k][$c] = ($n < $y)? $y : $n;
11       }
12       else { $f[$k][$c] = $n; }
13     }
14   }
```

En las líneas 1-3 inicializamos la tabla `@f`. Después aparecen dos bucles anidados: el primero en los objetos y el segundo en las capacidades. Las líneas 8-11 no son mas que la transcripción de la ecuación de estado 6.1.

6.16.2. El Módulo

A continuación comentamos los contenidos del módulo:

```
~/Lperl/src/threads/knapsack/Algorithm-Knap01DP/lib/Algorithm$ cat -n Knap01DP.pm
     1  package Algorithm::Knap01DP;
     2  use 5.008004;
     3  use strict;
     4  use warnings;
     5  use Carp;
     6  use IO::File;
     7
     8  use Exporter; # Warning!! h2xs generated a require Exporter;
     9  our @ISA = qw(Exporter);
    10  our @EXPORT_OK = qw/Knap01DP ReadKnap/;
```

Exportamos las funciones `qw/Knap01DP` y `ReadKnap` sólo bajo demanda.

```
    11  our $VERSION = '0.01';
    12  # Still a very early "alpha" version
    13
    14  sub Knap01DP {
    15    my $M = shift;
    16    my @w = @{shift()};
```

```
  my @p = @{shift()};

  my $N = @w;
  my @f;

  croak "Profits and Weights don't have the same size"
                        unless scalar(@w) == scalar(@p);

  for my $c (0..$M) {
    $f[0][$c] = ($w[0] <= $c)? $p[0] : 0;
  }

  for my $k (1..$N-1) {
    for my $c (0..$M) {
      my $n = $f[$k-1][$c];
      if ($c >= $w[$k]) {
        my $y = $f[$k-1][$c-$w[$k]]+$p[$k];
        $f[$k][$c] = ($n < $y)? $y : $n;
      }
      else { $f[$k][$c] = $n; }
    }
  }
  return @f;
}

sub ReadKnap {
  my $filename = shift;

  my $file = IO::File->new("< $filename");
  croak "Can't open $filename" unless defined($file);
  my (@w, @p);

  my $N = <$file>; chomp($N);
  my $M = <$file>; chomp($M);
  for (0..$N-1) {
    $w[$_] = <$file>;
    $p[$_] = <$file>;
  }
  chomp @w; chomp @p;
  return ($M, \@w, \@p);
}

1;
__END__
```

6.16.3. La Documentación

A partir del final del código hemos colocado la documentación. No olvides nunca actualizarla de acuerdo con los cambios que hagas en tu módulo.

```
=head1 NAME

Algorithm::Knap01DP - Solves the 0-1 Knapsack problem using
                      the Dynamic Programming Technique
```

```

=head1 SYNOPSIS

  use Algorithm::Knap01DP;

  my ($M, $w, $p) = ReadKnap($file);

  my @f = KnapP01DP($M, $w, $p);

=head1 DESCRIPTION

Solves the 0-1 Knapsack problem using the Dynamic Programming Technique.

   my @f = KnapP01DP($M, $w, $p);

  $M is the capacity
  $w is a reference to the array of weights
  $p is a reference to the array of profits

Returns the table $f[$k][$c] containing the optimal value for
objects 0..$k and capacity $c.

=head2 EXPORT

None by default.

=head1 SEE ALSO

L<Algorithm::Knapsack>

=head1 AUTHOR

Casiano Rodriguez Leon E<lt>casiano@ull.esE<gt>

=head1 COPYRIGHT AND LICENSE

Copyright (C) 2005 by Casiano Rodriguez Leon

This library is free software; you can redistribute it and/or modify
it under the same terms as Perl itself, either Perl version 5.8.4 or,
at your option, any later version of Perl 5 you may have available.


=cut
```

6.16.4. MANIFEST

Añadimos un fichero `MANIFEST.SKIP` que nos ayude a construir de manera automática el `MANIFEST`:

```
~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ cat -n MANIFEST.SKIP
 1  \.o$
 2  ^\.cvsignore$
 3  /\.cvsignore$
 4  \.cvsignore$
```

```
CVS/[^/]+$
^Makefile$
/Makefile$
^blib/
\.swp$
\.bak$
\.pdf$
\.ps$
pm_to_blib
.pdf$
```

Además editamos `Makefile.PL` y `README` para actualizar y/o añadir la información que consideremos adecuada.

6.16.5. El fichero `pm_to_blib`

La utilidad `make` compara los tiempos de modificación de fuentes y objetos para reconstruir sólo aquellos objetivos que estan caducados. En ocasiones el rastreo de los tiempos de modificación de los ficheros que existen en el directorio `blib` puede ser complejo, de manera que el `makefile` crea un fichero vacío de nombre `pm_to_blib` en el directorio del módulo y usa su tiempo de modificación para determinar si es necesario reconstruir. Si ocurre que `pm_to_blib` tiene la fecha incorrecta puede que `make` crea que `blib/` esta actualizado y se niege a reconstruirlo. En ese caso, simplemente borralo y haz `make` de nuevo.

6.16.6. El fichero `META.yml`

El fichero `META.yml` describe propiedades de la distribución Perl. Los campos ayudan a los mantenedores de CPAN y a los desarrolladores de módulos como CPAN y CPANPLUS en sus tareas. Esta escrito en `YAML`.

6.16.7. Las Pruebas

Ahora estudiemos el programa de prueba. Está en el directorio `t`.

```
$ pwd
/home/lhp/Lperl/src/threads/knapsack/Algorithm-Knap01DP/t
$ ls -l
-rw-r--r--  1 lhp lhp 1693 2005-05-17 19:57 01MartelloAndTothBook.t
-rw-r--r--  1 lhp lhp  134 2005-05-16 18:37 knap21.dat
-rw-r--r--  1 lhp lhp  125 2005-05-16 18:37 knap22.dat
-rw-r--r--  1 lhp lhp  123 2005-05-16 18:37 knap23.dat
-rw-r--r--  1 lhp lhp  158 2005-05-16 18:37 knap25.dat
```

Además del programa `01MartelloAndTothBook.t` tenemos cuatro ficheros con cuatro diferentes problemas de la mochila. Los números corresponden a las páginas del clásico libro de Martello y Toth [13] en el que aparece el correspondiente problema.

```
$ cat -n 01MartelloAndTothBook.t
# Before ‘make install’ is performed this script should be runnable with
# ‘make test’. After ‘make install’ it should work as ‘perl Algorithm-Knap01DP.t’

#########################
use strict;
use Test::More  tests => 11;

BEGIN { use_ok(’Algorithm::Knap01DP’, qw/Knap01DP ReadKnap/); }
```

Realizaremos 11 pruebas de las que la primera es comprobar que el módulo se carga correctamente (línea 8).

```
10  ### main
11  my @inputfiles = qw/knap21.dat  knap22.dat  knap23.dat  knap25.dat/;
12  my @sol = (280, 107, 150, 900);
13  my $knap21 = ['102', [ '2', '20', '20', '30', '40', '30', '60', '10' ],
14                    [ '15', '100', '90', '60', '40', '15', '10', '1' ]];
15  my $knap22 = ['50',  [ '31', '10', '20', '19', '4', '3', '6' ],
16                    [ '70', '20', '39', '37', '7', '5', '10' ]];
17  my $knap23 = ['190', [ '56', '59', '80', '64', '75', '17' ],
18                    [ '50', '50', '64', '46', '50', '5' ]];
19  my $knap25 = ['104', [ '25', '35', '45', '5', '25', '3', '2', '2' ],
20                    [ '350', '400', '450', '20', '70', '8', '5', '5' ]];
21
22  my $knapsackproblem = [$knap21, $knap22, $knap23, $knap25];
```

La variable `@inputfiles` contiene los nombres de los ficheros de prueba. La variable `@sol` las soluciones óptimas a esos problemas. Las variables `$knap21` ... `$knap25` contienen estructuras de datos que definen los problemas: capacidad de la mochila, vector de pesos y vector de beneficios. Cuando la función `ReadKnap` lee un fichero de datos devuelve una estructura como esta:

```
~/Lperl/src/threads/knapsack/Algorithm-Knap01DP/t$ perl \
   -I/home/lhp//Lperl/src/threads/knapsack/Algorithm-Knap01DP/lib \
   -d 01MartelloAndTothBook.t
Loading DB routines from perl5db.pl version 1.25
1..11
ok 1 - use Algorithm::Knap01DP qw/Knap01DP ReadKnap/;;
main::(01MartelloAndTothBook.t:11):
11:     my @inputfiles = qw/knap21.dat  knap22.dat  knap23.dat  knap25.dat/;
  DB<1> b 31 # ponemos un break después de la llamada a ReadKnap
  DB<2> c # y ejecutamos hasta allí
main::(01MartelloAndTothBook.t:31):
31:       is_deeply($knapsackproblem->[$i], [$M, $w, $p], "ReadKnap $file");
  DB<2> x [$M, $w, $p] # Veamos que nos devuelve ReadKnap:
0  ARRAY(0x8555ad0)
   0  102
   1  ARRAY(0x845e034)
      0  2
      1  20
      2  20
      3  30
      4  40
      5  30
      6  60
      7  10
   2  ARRAY(0x845e04c)
      0  15
      1  100
      2  90
      3  60
      4  40
      5  15
      6  10
      7  1
```

De hecho, es usando el depurador, cortando y pegando que hemos construido las líneas 13-20 definiendo las estructuras de datos `$knapXX`. A continuación leeemos cada fichero y comprobamos que ambas `ReadKnap` y `Knap01DP` dan los resultados esperados. La función `is_deeply` nos dice si dos estructuras de datos son equivalentes. Véase `perldoc Test::More` para mas información sobre el módulo `Test::More` y las funciones `is_deeply` e `is`.

```
24  my $i = 0;
25  my ($M, $w, $p);
26  my @f;
27
28  # Now 2*@inputfiles = 8 tests
29  for my $file (@inputfiles) {
30    ($M, $w, $p) = ReadKnap((-e "t/$file")?"t/$file":$file);
31    is_deeply($knapsackproblem->[$i], [$M, $w, $p], "ReadKnap $file");
32    my $N = @$w;
33    @f = Knap01DP($M, $w, $p);
34    is($sol[$i++], $f[$N-1][$M], "Knap01DP $file");
35  }
```

A continuación realizamos una prueba para comprobar el funcionamiento cuando se le pasan a `Knap01DP` vectores de pesos y beneficios de distinto tamaño. Recordemos que en la rutina `Knap01DP` habíamos escrito el siguiente código:

```
22    croak "Profits and Weights don't have the same size"
                                    unless scalar(@w) == scalar(@p)
```

por tanto, pasarle a la rutina vectores de distinto tamaño hace que el programa *muera*. Es por esto que protegeremos la ejecución dentro de un `eval`:

```
37  # test to check when weights and profits do not have the same size
38  $M = 100; @$w = 1..5; @$p = 1..10;
39  eval { Knap01DP($M, $w, $p) };
40  like $@, qr/Profits and Weights don't have the same size/;
```

Observe las salidas y los comentarios en inglés. Si tu intención es hacer público el módulo en CPAN es recomendable que las salidas, los nombres de variables y los comentarios estén en ese idioma.

Vamos a hacer una prueba mas. Supongamos que tengo la intención de añadir una función `GenKnap` que genere aleatoriamente un problema de la mochila. Como no esta hecho, lo declaramos como una prueba a hacer (`TODO`). Es decir, se trata de un test que fallará, pero que se espera que deje de hacerlo en el futuro.

```
42  TODO: { # I plan to provide a function to find the vector solution ...
43    local $TODO = "Return vector solution";
44    can_ok('Algorithm::Knap01DP', 'GenKnap');
45  }
```

Primero una ejecución *a mano*:

```
~/Lperl/src/threads/knapsack/Algorithm-Knap01DP/t$ perl \
-I/home/lhp//Lperl/src/threads/knapsack/Algorithm-Knap01DP/lib 01MartelloAndTothBook.t
1..11
ok 1 - use Algorithm::Knap01DP qw/Knap01DP ReadKnap/;;
ok 2 - ReadKnap knap21.dat
ok 3 - Knap01DP knap21.dat
ok 4 - ReadKnap knap22.dat
ok 5 - Knap01DP knap22.dat
ok 6 - ReadKnap knap23.dat
```

```
ok 7 - Knap01DP knap23.dat
ok 8 - ReadKnap knap25.dat
ok 9 - Knap01DP knap25.dat
ok 10
not ok 11 - Algorithm::Knap01DP->can('GenKnap') # TODO Randomly generated problem
#     Failed (TODO) test (01MartelloAndTothBook.t at line 45)
#     Algorithm::Knap01DP->can('GenKnap') failed
```

Obsérvese que:

- Estabamos en el directorio `t`.
- Usamos `-I` para que pueda encontrar el módulo.
- Nos da el mensaje: `not ok 11 - ... # TODO Return ...` indicando que falla y que es una prueba `TODO`.

Sigue una ejecución con `make test` (un directorio por encima):

```
~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" \
                                "-e" "test_harness(0, 'blib/lib', 'blib/arch')" \
                                t/*.t
t/01MartelloAndTothBook....ok
All tests successful.
Files=1, Tests=11,  0 wallclock secs ( 0.09 cusr +  0.00 csys =  0.09 CPU)
```

Observa como ahora se informa que todas las pruebas fueron correctamente. Se ha ocultado que hay una prueba `TODO` y su fallo no se considera significativo para la posible instalación del módulo. De este modo el directorio de pruebas puede ser utilizado como lista recordatorio de objetivos y requerimientos a realizar.

6.16.8. El Módulo `test_harness` y el guión `prove`

El módulo `ExtUtils::Command::MM` proporciona la función `test_harness` la cual tiene dos argumentos:

```
test_harness($verbose, @test_libs)
```

Lo que hace es ejecutar todas las pruebas usando un módulo denominado `Test::Harness`. Si `verbose` se pone a 1 se obtiene información mas completa:

```
:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ PERL_DL_NONLAZY=1 /usr/bin/perl \
    "-MExtUtils::Command::MM" "-e" "test_harness(1, 'blib/lib', 'blib/arch')" t/*.t
t/01MartelloAndTothBook....1..11
ok 1 - use Algorithm::Knap01DP qw/Knap01DP ReadKnap/;;
ok 2 - ReadKnap knap21.dat
ok 3 - Knap01DP knap21.dat
ok 4 - ReadKnap knap22.dat
ok 5 - Knap01DP knap22.dat
ok 6 - ReadKnap knap23.dat
ok 7 - Knap01DP knap23.dat
ok 8 - ReadKnap knap25.dat
ok 9 - Knap01DP knap25.dat
ok 10
not ok 11 - Algorithm::Knap01DP->can('GenKnap') # TODO Randomly generated problem

#     Failed (TODO) test (t/01MartelloAndTothBook.t at line 44)
```

```
#     Algorithm::Knap01DP->can('GenKnap') failed
ok
All tests successful.
Files=1, Tests=11,  0 wallclock secs ( 0.08 cusr +  0.02 csys =  0.10 CPU)
```

El módulo `Test::Harness` controla a los otros módulos de la familia `Test::`. Exporta la función `runtests(@test_files)` la cual ejecuta todas los programas de prueba pasados en `@test_files`. La función `runtests` intercepta la salida de las pruebas y las interpreta. Se supone que toda prueba debe escribir primero la cadena `1..M` donde `M` es el número de pruebas y después, por cada prueba la salida debe contener la cadena `ok N`, donde N es el numeral de la prueba. Cuando una prueba es `TODO` su salida debe contener la cadena `# TODO` después de `not ok` o de `ok`. El texto que sigue después de esto es la descripción de lo que se supone que debería hacerse.

La utilidad `prove` provee un acceso a las funcionalidades de `Test:Harness` y constituye una alternativa a `make test` durante el desarrollo del módulo. Veamos un ejemplo e uso:

```
~/Lperl/src/threads/knapsack/Algorithm-Knap01DP/t$ prove -I../lib 01MartelloAndTothBook.t
01MartelloAndTothBook....ok
All tests successful.
Files=1, Tests=11,  0 wallclock secs ( 0.07 cusr +  0.02 csys =  0.09 CPU)
```

La opción `v` activa el modo *verbose*:

```
$ prove -v -I../lib 01MartelloAndTothBook.t
01MartelloAndTothBook....1..11
ok 1 - use Algorithm::Knap01DP qw/Knap01DP ReadKnap/;;
ok 2 - ReadKnap knap21.dat
ok 3 - Knap01DP knap21.dat
ok 4 - ReadKnap knap22.dat
ok 5 - Knap01DP knap22.dat
ok 6 - ReadKnap knap23.dat
ok 7 - Knap01DP knap23.dat
ok 8 - ReadKnap knap25.dat
ok 9 - Knap01DP knap25.dat
ok 10
not ok 11 - Algorithm::Knap01DP->can('GenKnap') # TODO Randomly generated problem

#     Failed (TODO) test (01MartelloAndTothBook.t at line 44)
#     Algorithm::Knap01DP->can('GenKnap') failed
ok
All tests successful.
Files=1, Tests=11,  0 wallclock secs ( 0.11 cusr +  0.00 csys =  0.11 CPU)
```

6.16.9. Práctica: Pruebas

Añada al módulo la función `GenKnap` y añada al programa de pruebas una nueva prueba para comprobar su funcionamiento. Use las funciones `rand` y `srand`. Use la función `int` para obtener un entero (p. ej `int(rand(10)` devuelve un entero entre 0 y 9). La prueba puede consistir en comprobar que los pesos y los beneficios están en un rango dado. Otra posible prueba es llamara a `GenKnap` dos veces y comprobar que los problemas generados son distintos.

6.16.10. Hallar la Versión de un Módulo

La respuesta es:

```
$ perl -MTest::More -e 'print "$Test::More::VERSION\n"'
0.47
```

6.16.11. Dónde está un Módulo

Si lo que se quiere es saber donde está instalado o donde esta la documentación puede usar un guión como este:

```
$ cat /usr/local/bin/which.pl
#!/usr/bin/perl -w

my $pattern = shift || '';
$pattern =~ s{::}{/}g; # En unix ...
my @dirs = @INC;

for (@dirs) {
  my $file = $_."/".$pattern.'.pm';
  print "$file\n" if (-e $file);
  $file = $_."/".$pattern.'.pod';
  print "$file\n" if (-e $file);
  $file = $_."/pod/".$pattern.'.pod';
  print "$file\n" if (-e $file);
}
```

El guión muestra todos los módulos con nombre dado asi como los PODs:

```
$ which.pl Test::More
/usr/local/share/perl/5.8.4/Test/More.pm
/usr/share/perl/5.8/Test/More.pm
```

6.16.12. Ejecutables

Si se tienen guiones ejecutables que acompañan el módulo es conveniente hacerlo notar en `Makefile.PL` usando la opción `EXE_FILES` de `WriteMakefile`. El valor para esta clave es una referencia a un array de ficheros ejecutables. Por ejemplo:

```
# Verify perl version.
 require 5.000;
 use ExtUtils::MakeMaker;
 my @scripts = qw(fontsampler pfb2pfa pfa2pfb);
 WriteMakefile
   (
    NAME      => 'PostScript::Font',
    ($[ >= 5.005) ?
    (AUTHOR   => 'Johan Vromans (jvromans@squirrel.nl)',
     ABSTRACT => 'Modules to get info from PostScript fonts') : (),
    VERSION   => "0.03",
    PREREQ_PM => { 'Getopt::Long' => 2.00, 'IO' => 0 },
    EXE_FILES => [ map { "script/$_" } @scripts ],
    # *.pm files will be picked up automatically from ./lib
  );
```

Los ficheros serán copiados al directorio `INST_SCRIPT` (por defecto `./blib/script`) cuando se haga `make`. Hacer `make realclean` borrará estas copias. Hacer `make install` los copiará desde `INST_SCRIPT` hasta `INSTALLSCRIPT`. Veamos un ejemplo típico de su valor:

```
~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ grep '^INSTALLSCRIPT' Makefile
INSTALLSCRIPT = $(PERLPREFIX)/bin
~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ grep '^PERLPREFIX' Makefile
PERLPREFIX = $(PREFIX)
```

```
~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ grep '^PREFIX' Makefile
PREFIX = /usr
```

El método `WriteMakefile` sobreescribirá la cabecera del ejecutable, por ejemplo`#!/usr/bin/perl`, por el camino que lleva a la versión del compilador Perl que se usó para ejecutar `Makefile.PL`.

6.16.13. Práctica: Ejecutable en una Distribución

Añada a la distribución de `Algorithm::Knap01DP` un guión al que se le pasa como argumento el nombre de un fichero conteniendo un problema de la mochila y que produce como salida la solución.

6.16.14. Práctica: Pruebas SKIP

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`. Construya una prueba `SKIP` para el módulo `Algorithm::Knap01DP`. Por ejemplo, si el módulo Algorithm::Knapsack está instalado en la máquina, genere un problema aleatorio y compruebe que la solución dada por ambos algoritmos es la misma.

El módulo `Algorithm::Knap01DP` puede obtenerlo desde:

http://nereida.deioc.ull.es/~lhp/perlexamples/Algorithm-Knap01DP-0.01.tar.gz.

6.16.15. A Veces las Pruebas Tienen Fallos

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

Use `perldoc Test::Tutorial` para estudiar el tutorial de Michael Schwern sobre pruebas.

Capítulo 7

Programación Orientada a Objetos

Las características de la Programación orientada a objetos en Perl pueden resumirse en:

- Para crear una *clase* se construye un "package"
- Para crea un *método* se escribe una subrutina
- 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.
- *Constructores*: En Perl son rutinas que retornan una referencia a un objeto recién creado e inicializado.
- *Destructores*: Los destructores son rutinas que son llamadas cuando un objeto deja de existir porque deja de ser referenciado o porque su ámbito termina.
- Herencia: Perl soporta herencia de clases, simple y múltiple.
- Sobrecarga: Los métodos pueden sobrecargarse. El módulo `overload` permite la sobrecarga de operadores.

7.1. Introducción

Sigue un ejemplo que gestiona una biblioteca:

```
package Biblio::Doc;
use strict;

{
  my $_count = 0;
  sub get_count { $_count }
  sub _incr_count { $_count++ }
}

sub new {
  my $class = shift;
  my ($identifier, $author, $publisher, $title,  $year, $url) = @_;
  $class->_incr_count();

  my $paper = {
    _identifier      => $identifier,
    _author    => $author,
    _publisher => $publisher,
    _title     => $title,
```

```
    _year      => $year,
    _url       => $url
  };

  bless  $paper, $class;
}

sub get_identifier { $_[0]->{_identifier} }
sub get_author { $_[0]->{_author} }
sub get_publisher { $_[0]->{_publisher} }
sub get_title { $_[0]->{_title} }
sub get_year { $_[0]->{_year} }
sub get_url { $_[0]->{_url} }

sub set_url {
  my ($self, $url) = @_;
  $self ->{_url} = $url if $url;
  return ($self ->{_url});
}

1;
```

La clase `Biblio::Doc` se implanta a través del package `Biblio::Doc`

```
package Biblio::Doc;
use strict;
...
```

Los paquetes proporcionan un conjunto de características que son comunes a las *clases*: proporcionan un espacio de nombres separado, agrupando así un código que ofrece un conjunto de funcionalidades relacionadas y aislándolo del resto; tienen un nombre que puede ser usado para identificar los datos y subrutinas en el paquete (*fully qualified names*) y, si el paquete es un módulo, diferencian entre la parte a exportar y la parte interna. Por todo ello Perl usa los paquetes para implantar las clases.

Obsérvese la clausura creada para el contador de referencias bibliográficas `$_count`. Este contador es el único *atributo de clase* en este ejemplo. Los restantes son *atributos del objeto*.

```
{
  my $_count = 0;
  sub get_count { $_count }
  sub _incr_count { $_count++ }
}
```

Es un convenio en Perl que las variables que comienzan con un guión bajo son *variables privadas*. Consecuencia de este convenio es que debemos entender que el autor pretende que `$_count` y `_incr_count` sean privadas.

La variable privada `$_count` contiene el contador de documentos. Su acceso queda restringido no ya al ámbito del módulo sino al ámbito léxico establecido por la clausura. La única forma de acceder a `$_count` es a través de las funciones `_incr_count` y `get_count`.

En general, en programación orientada a objetos se aconseja que se envuelvan todos los accesos a atributos en subrutinas y se prohiba la posible modificación exterior de los mismos ([14] .

A continuación viene el constructor de objetos `new` (su nombre es arbitrario, pero la costumbre es llamar así a los constructores) de la clase `Biblio::Doc`

```
sub new {
  my $class = shift;
  my ($identifier, $author, $publisher, $title,  $year, $url) = @_;
```

```
 4   $class->_incr_count();
 5
 6   my $paper = {
 7     _identifier      => $identifier,
 8     _author    => $author,
 9     _publisher => $publisher,
10     _title     => $title,
11     _year      => $year,
12     _url       => $url
13   };
14
15   bless  $paper, $class;
16 }
```

Un objeto Perl es un dato que ha sido bendecido con un paquete. La operación de bendición `bless` toma como primer argumento una referencia y como segundo un nombre de paquete. El argumento "nombre del paquete" es opcional: se usará el nombre del package actual si se omite. Una vez que un dato es "bendecido" (¿Quizá deberíamos decir "bautizado como miembro de la clase"?) el dato pasa a "saber" a que clase-package pertenece. En el ejemplo se bendice una referencia a un hash. Sin embargo, Perl admite que se bendiga una referencia a cualquier otro tipo de objeto.

Podremos crear el objeto con una llamada como la siguiente:

```
$obj = Biblio::Doc->new(1, "Asimov", "Bruguera",
   "Los propios dioses", 1989, "unknown");
```

o bien usando esta otra sintáxis:

```
$obj = new Biblio::Doc(1, "Asimov", "Bruguera",
   "Los propios dioses", 1989, "unknown");
```

Lo que sucede cuando Perl ve una llamada a una subrutina seguida del nombre de un paquete (`new Biblio::Doc`) o una llamada con la sintáxis de paquete seguido de flecha seguido de un método (`Biblio::Doc->new`) es que llama a una subrutina con ese nombre en ese paquete, y le pasa como primer argumento el nombre del paquete. Así, el primer argumento que recibe el constructor `new` es el nombre de la clase, esto es `Biblio::Doc`. Así pues la línea 2

```
  my $class = shift;
```

Obtiene la clase `Biblio::Doc`.

La llamada de la línea 4:

```
$class->_incr_count();
```

es, por tanto, otro ejemplo de esta forma de llamada. En este caso tenemos además un referenciado simbólico via `$class`.

El hash referenciado por `$paper` en las líneas de la 6 a la 13 es bendecido en la línea 15 con el nombre de la clase. Esto establece un campo interno dentro de la representación interna del hash anónimo. En cierto sentido es `%$paper` quien es bendecido, no `$paper`. La figura 7.1 ilustra este punto: En la izquierda de la figura aparecen la referencia y el referente antes de la bendición. En la parte de la derecha, después de ejecutar `bless  $paper, $class`. Obsérvese que se ha añadido una marca con el nombre de la clase-paquete que identifica la estructura de datos como un objeto de esa clase.

Una vez que el dato apuntado por la referencia ha sido bendecido, el operador `ref` aplicado a dicha referencia devuelve el nombre del paquete (esto es, el nombre de la clase) en vez de `HASH`.

Una comprobación de que quien es bendecido es el dato referido no la referencia la da el siguiente código:

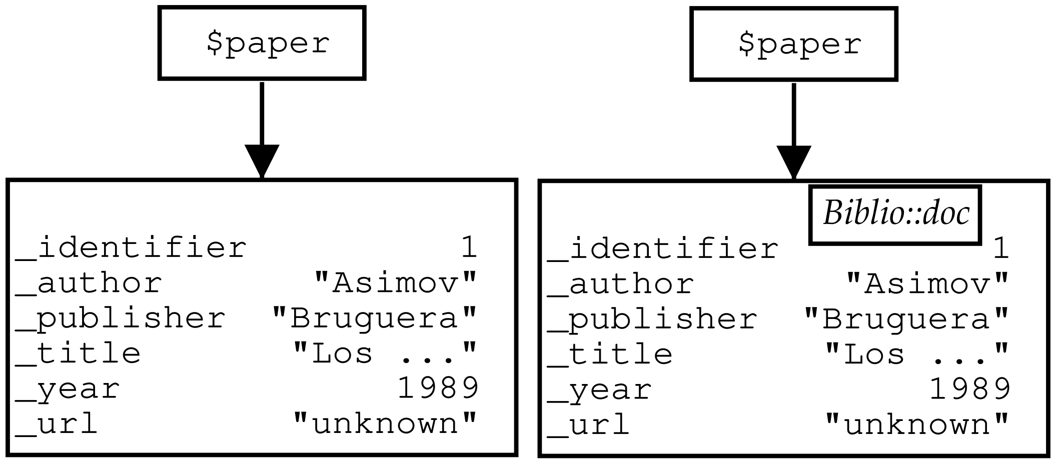

Figura 7.1: Al bendecir un objeto este es marcado con la clase a la que pertenece

```
$obj = new Biblio::Doc(1, "Asimov", "Bruguera",
   "Los propios dioses", 1989, "unknown");
$obj2 = $obj;
print "obj2 es un ",ref($obj2),"\n";
```

Se imprimirá que `obj2 es un Biblio::Doc`

Para escribir un método para una clase (package) dada, basta con escribir la correspondiente subrutina dentro del paquete.

Así el método `get_author` queda descrito mediante la subrutina:

```
sub get_author { $_[0]->{_author} }
```

Sin embargo, llamar a `Biblio::Doc::get_author` como método conlleva una extensión a la *sintáxis flecha* de Perl. Si se tiene que `$obj` es una referencia a un objeto de la clase `Biblio::Doc`, podemos acceder a cualquier método del paquete usando la notación `$obj->method` donde `method` es cualquier método de la clase. Por ejemplo:

```
$obj->get_author()
```

Es equivalente a

```
Biblio::Doc::get_author($obj);
```

Esto es: *cuando una subrutina es llamada como método, la lista de argumentos que recibe comienza con la referencia al objeto a través del cual fué llamada y va seguida de los argumentos que fueron explicitados en la llamada al método.*

Observe que la última llamada trata a `get_author` como una subrutina ordinaria, mientras que la primera la usa "como método".

La sintáxis "flecha" para la llamada a un método es consistente con el uso de la flecha para otros tipos de referencias en Perl. Así, al igual que escribimos `$hsh_ref->{clave}` o bien `$sub_ref->(4,7,8)` también escribimos `$obj_ref->metodo(@args)`. Una diferencia de esta última notación con las anteriores estriba en que el referente tiene diferentes conductas, esto es, tiene varios metodos y la conducta concreta que queremos producir se especifica escribiendo el nombre del método después de la flecha.

Un método que es invocado a través de un objeto vía `$obj_ref->metodo(@args)` (por oposición a un *objeto de clase* que es aquel que es invocado vía una clase/paquete `Class->metodo(@args)`) se denomina un *método dinámico* o *método de objeto*.

Nótese el uso de los prefijos `get_` y `set_` utilizados para establecer el tipo de acceso al atributo. Un método a través del cual se accede a un atributo recibe el nombre de *accesor*. Un método que permite modificar el estado interno de un atributo se denomina *mutator*.

Un programador C se sentiría tentado de usar prototipos para especificar mas claramente el tipo de acceso a un método. Perl no comprueba prototipos cuando una subrutina de paquete es llamada como método, utilizando la sintáxis `$objref->method(@args)`. En el caso mas general resulta imposible conocer hasta el momento de la ejecución que subrutina debe ser invocada en respuesta a una llamada de un método.

7.1.1. Práctica: Un Módulo OOP Simple

Escriba un programa que haga uso del módulo `Biblio::Doc` presentado en la sección 7.1. Haga uso de `h2xs`.

7.1.2. Ejercicio: Número de argumentos de `bless`

Suponga que el constructor `new` de la clase `Biblio::Doc` es llamado desde otro paquete usando la sintáxis flecha como sigue:

```
#!/usr/bin/perl -w -I.

use Biblio::Doc;

package X;

$obj = X->Biblio::Doc::new(1, "Asimov", "Bruguera", "Los propios dioses", "1970", "/");

print ("El nombre del autor es: ", $obj->get_author(), "\n");
```

¿En que paquete es bautizado el objeto `$obj`? ¿Cuál será la salida?

7.1.3. Práctica: Métodos Privados

Modifique la clase `Biblio::Doc` para que la subrutina `_incr_count` sea realmente privada, esto es, su ámbito de visibilidad quede limitado al fichero que describe la clase. ¿Qué cambios son necesarios?

7.1.4. Práctica: Generación Automática de Métodos

Usando la función `AUTOLOAD` escriba un método genérico para `Biblio::Doc` que responda a cualquier llamada a los métodos de acceso a los atributos. Esto es, que sustituya a las funciones `get_identifier`, `get_author`, `get_publisher`, `get_title`, etc.

Recuerde que la variable `$AUTOLOAD` contiene el nombre de la rutina llamada. Si dicho nombre contiene el prefijo `get_` seguido de un identificador, se trata, posiblemente de una llamada a un método de lectura. Use expresiones regulares para determinar de cual se trata y, si efectivamente es un campo legal, retorne su valor. Para ello deberá usar la función `exists`, la cual recibe como argumento una “presunta” clave de un “hash” `expr` y nos retorna cierto cuando dicha clave existe en el “hash” (incluso si su valor es `undef`).

7.2. Generación Automática de Accesors/Mutators

La solución del ejercicio 7.1.4 es ineficiente. es posible mejorar la eficiencia, haciendo que cada vez que se busca una función y no se encuentra, `AUTOLOAD` dinámicamente se instale una entrada con la función en la tabla de símbolos, de manera que la siguiente llamada encuentre el método. Para ello usaremos un “typeglob” y una subrutina anónima. Sigue el código del módulo:

```
package Biblio::Doc1;
$VERSION = 1.00;
use strict;
use vars qw/$AUTOLOAD/;
```

La declaración `use vars` es similar a `ours`.

```
 6 {
 7   my $_count = 0;
 8   sub get_count { $_count }
 9   sub incr_count { $_count++ }
10 }
11
12 sub new {
13   my $class = shift;
14   my ($identifier, $author, $publisher, $title, $url) = @_;
15   incr_count();
16   my $document = {
17     _identifier      => $identifier,
18     _author    => $author,
19     _publisher => $publisher,
20     _title     => $title,
21     _url       => $url,
22   };
23
24   bless $document, $class;
25 }
26
27 sub AUTOLOAD {
28   no strict "refs";
29
30   my $self = shift;
31   if ($AUTOLOAD =~ /\w+::\w+::get(_.*)/) {
32     my $n = $1;
33     return unless exists $self->{$n};
34
35     # Declarar el metodo get_*****
36     *{$AUTOLOAD} = sub { return "(nuevo get) ".$_[0]->{$n}; };
37
38     return "(autoload) ".$self->{$n};
39   } elsif ($AUTOLOAD =~ /\w+::\w+::set(_.*)/) {
40     my $n = $1;
41     return unless exists $self->{$n};
42     $self->{$n} = "(autoload) ".shift;
43
44     # Declarar el metodo set_*****
45     *{$AUTOLOAD} = sub { $_[0]->{$n} = "(nuevo set) ".$_[1]; };
46   } else {
47     @_ = map { "\"".$_."\""; } @_;   # Comillas a los argumentos...
48     print "Has llamado a $AUTOLOAD(",join(", ",@_),") y no existe!\n";
49   }
50 }
51
52 1;
```

En la línea 31 obtenemos el nombre del atributo y lo guardamos (línea 32) en `$n`. En la línea 33 comprobamos que tal atributo existe. La subrutina anónima que crea `AUTOLOAD` en la línea 36 es una clausura con respecto a `$n`: el valor se recuerda, incluso después que se haya salido del ámbito.

Obsérvese la asignación de la línea 36:

```
*{$AUTOLOAD} = sub { return "(nuevo get) ".$_[0]->{$n}; };
```

Esto hace que se instale una entrada con el nombre del método deseado en la tabla de símbolos. El `use strict` hace que el compilador se queje, ya que el lado derecho es una referencia a subrutina y el lado izquierdo un "typeglob":

```
~/perl/src> use_Biblio_Doc.1.pl
Can't use string ("get_author") as a symbol ref while "strict refs" in use at Biblio/Doc1.pm line 36.
Has llamado a Biblio::Doc1::DESTROY() y no existe!
```

Podemos hacer que la queja desaparezca, escribiendo al comienzo de `AUTOLOAD` la directiva:

```
no strict "refs";
```

hace que `strict` deje de controlar el uso de referenciado simbólico en el ámbito en el que se ubique.

Sigue un ejemplo de programa cliente:

```
#!/usr/bin/perl -w -I.

use Biblio::Doc1;

$obj = Biblio::Doc1->new(1, "Asimov", "Bruguera",
   "Los propios dioses", "unknown");

print ("El nombre del autor es: ", $obj->get_author(), "\n");
print ("La URL es: ", $obj->get_url(), "\n");

$obj->set_author("El buen doctor");
print ("El nuevo nombre del autor es: ", $obj->get_author(), "\n");

$obj->set_author("Isaac Asimov");
print ("El nombre del autor definitivo es: ", $obj->get_author(), "\n");
```

La ejecución del programa anterior produce la salida:

```
~/perl/src> use_Biblio_Doc.1.pl
El nombre del autor es: (autoload) Asimov
La URL es: (autoload) unknown
El nuevo nombre del autor es: (nuevo get) (autoload) El buen doctor
El nombre del autor definitivo es: (nuevo get) (nuevo set) Isaac Asimov
Has llamado a Biblio::Doc1::DESTROY() y no existe!
```

7.2.1. Ejercicio: Mutators y Autocarga

¿Que pasaría en el ejemplo de la sección anterior (sección 7.2) si un método de acceso recibiera dos o mas argumentos? Por ejemplo: `get_issue($url, $year)`? (Suponemos que `issue` no es un atributo de la clase). ¿Que ocurriría en tu programa si ya existiera un método con tal nombre?

7.2.2. Práctica: Instalación Automática de Métodos

En la dirección http://nereida/~lhp/perlexamples/Algorithm-Knap01DP-0.20.tar.gz encontrará una versión orientada a objetos del módulo que desarrollamos en la sección 6.16.

La estructura del hash puede verse consultando el constructor `new`:

```
sub new {
    my $class = shift;
    my $self = {
        capacity    => 0,       # total capacity of this knapsack
```

```
        numobjects  => 0,       # number of objects
        weights     => [],      # weights to be packed into the knapsack
        profits     => [],      # profits to be packed into the knapsack
        tableval    => [],      # f[k][c] DP table of values
        tablesol    => [],      # x[k][c] DP table of sols
                                # (0 = out, 1 = in, 2 = in and out)
        solutions   => [],      # list of lists of object indexes
        filename    => "",      # name of the file the problem was read from
        @_,
    };

    croak "Profits and Weights don't have the same size"
          unless scalar(@{$self->{weights}}) == scalar(@{$self->{profits}});

    bless $self, $class;
}
```

Escriba el conjunto de métodos de acceso, haciendo que cada vez que se busca una función y no se encuentra, `AUTOLOAD` dinámicamente instale una entrada con la función en la tabla de símbolos, de manera que la siguiente llamada encuentre el método. Introduzca una prueba de regresión que verifique el funcionamiento. Modifique la documentación.

7.3. Constructores

Se suelen llamar *constructores* a aquellos métodos de la clase que asignan memoria para la estructura de datos en la que subyace el nuevo objeto, inicializándolo y bendiciéndolo. A la hora de escribir un constructor es conveniente tener en cuenta los siguientes puntos:

1. Lo habitual es que el nombre de un constructor sea `new`.

2. Es conveniente escribir el proceso de inicialización en un método privado separado que es llamado desde el constructor.

3. Es habitual que los constructores tengan muchos parámetros. Es conveniente utilizar la estrategia explicada en la subsección 1.9.8 de usar un hash para proveer paso de argumentos con nombre.

4. Adicionalmente, es necesario proveer valores por defecto en la inicialización, de manera que campos no especificados por el usuario resulten inicializados de manera apropiada.

5. Se suele distinguir entre atributos de "sólo lectura" y atributos de "lectura/escritura", controlando el tipo de acceso que el programa cliente hace al atributo.

A continuación veamos un código que, siguiendo las recomendaciones establecidas, separa el proceso de iniciación, organiza los parámetros del constructor según un "hash" y provee valores por defecto:

```
package Biblio::Doc;
use strict;
use vars('$AUTOLOAD');

# Separamos inicialización de construcción

{
  my $_count = 0;
  sub get_count { $_count }
  sub _incr_count { $_count++ }
  sub _decr_count { $_count-- }
```

```
12 }
13
14 {
15   my %_defaults = (             #   Default   Access
16                   _identifier => ["unknown",'read'],
17                   _author     => ["unknown",'read/write'],
18                   _publisher  => ["unknown",'read'],
19                   _title      => ["unknown",'read'],
20                   _year       => ["unknown",'read'],
21                   _url        => ["unknown",'read/write']
22                 );
```

Se ha introducido un hash que contiene los valores por defecto y el tipo de acceso permitio (lectura/escritura) para cada clave.

```
24   sub _standard_keys {
25     return keys %_defaults;
26   }
```

Queremos además disponer de un método que nos diga que claves forman el objeto. Esa es la función del método privado `_standard_keys`.

```
28   sub _default_for {
29     my ($self, $attr) = @_;
30
31     return $_defaults{$attr}[0];
32   }
33
34   sub _accesible {
35     my ($self, $attr, $access) = @_;
36     return $_defaults{$attr}[1] =~ /$access/;
37   }
```

El método `_default_for` permite acceder al valor por defecto. El método `_accesible` nos dice si una clave esta accesible para el tipo de acceso requerido.

```
39   sub _init {
40     my ($self, %args) = @_;
41     my %inits;
42     my ($i, $j);
43
44     for $i ($self->_standard_keys) {
45       $j = $i;
46       $j =~ s/_//;
47       $inits{$i} = $args{$j} || $self->_default_for($i);
48     }
49     %$self = %inits;
50   }
51 }
```

El método privado `_init` inicializa el hash referenciado por `$self` según lo indicado en el hash `%args`. Las claves en `%args` no van precedidas de guión bajo (se supone que la llamada desde el programa cliente usará los nombres de los atributos sin guión bajo). Si la clave no figura en `%args` se inicializa al valor por defecto. Observe que el uso de `$self->_standard_keys` en el bucle de la línea 44 nos protege contra errores de inicializaciones con claves inexistentes en el objeto, posiblemente como consecuencia de un error en la llamada desde el cliente (quizá debidos a un error tipográfico).

```
53 sub new {
54   my $class = shift;
55   my $self = {};
56
57   bless $self, ref($class) || $class;
58
59   $self->_incr_count();
60   $self->_init(@_);
61
62   return $self;
63 }
```

En la línea 57 se bendice el objeto aún sin inicializar. Cuando el constructor es llamado mediante `Biblio::Doc->new()` la variable `$class` contendrá la cadena `'Biblio::Doc'` por lo que `ref($class)` devuelve `undef` y `$self` será bendecido en `'Biblio::Doc'`. La línea 57 esta escrita pensando que el constructor pueda ser llamado de la forma `$x->new` donde `$x` es un objeto de la clase `Biblio::Doc`. Como sabemos, en tal caso `ref($x)` contendrá la cadena `'Biblio::Doc'`. Asi pues, `$self` será, también en este caso, bendecido en `'Biblio::Doc'`.

Observe como hemos desacoplado la inicialización de la creación y bendición de la estructura. Es posible que durante el periodo de desarrollo del módulo la estructura del hash cambie, introduciendo nuevos atributos o suprimiendo algunos existentes. Con el desacoplado conseguimos que estos cambios no afecten a `new` (aunque si a `_init`).

```
65 sub DESTROY {
66   my $self = shift;
67
68   $self->_decr_count();
69 }
```

Perl elimina automáticamente la memoria de un objeto cuando es claro que ya no va a ser utilizado. Un método con el nombre especial `DESTROY` se dispara cuando la memoria asociada con un objeto debe ser eliminada (normalmente por que salimos de su ámbito de existencia).

```
71 sub AUTOLOAD {
72   no strict "refs";
73   my $self = shift;
74   if (($AUTOLOAD =~ /\w+::\w+::get(_.*)/) && ($self->_accesible($1,'read'))) {
75     my $n = $1;
76     return unless exists $self->{$n};
77
78     # Declarar el metodo get_*****
79     *{$AUTOLOAD} = sub { return "(nuevo get) ".$_[0]->{$n}; };
80
81     return "(autoload) ".$self->{$n};
82   } elsif (($AUTOLOAD =~ /\w+::\w+::set(_.*)/) && ($self->_accesible($1,'write'))) {
83     my $n = $1;
84     return unless exists $self->{$n};
85     $self->{$n} = "(autoload) ".shift;
86
87     # Declarar el metodo set_*****
88     *{$AUTOLOAD} = sub { $_[0]->{$n} = "(nuevo set) ".$_[1]; };
89   } else {
90     @_ = map { "\"".$_."\""; } @_;   # Comillas a los argumentos...
91     print "Has llamado a $AUTOLOAD(",join(", ",@_),")\n";
92   }
```

```
93 }
94
95 1;
```

También hemos extendido la subrutina `AUTOLOAD` para que compruebe (líneas 74 y 82) que el método de acceso requerido está permitido.

7.4. Copia de Objetos

Es costumbre en Perl que, cuando se llama a un constructor como método de un objeto (esto es `$objref->new()` en vez de como método de una clase (`Biblio::Doc->new()`) el constructor utilice como valores de inicialización por defecto los que tiene ese objeto (`$objref->new()`). En resumen: el constructor permite realizar la creación de una copia de un objeto.

Obsérvese que, con respecto al punto planteado sobre la creación de duplicados de un objeto, no basta con copiar el "hash", ya que una asignación a un nuevo "hash" no copia la "bendición". Existe además el incoveniente de que si uno de los atributos del objeto es una referencia, deberíamos crear un duplicado de lo referenciado mas que copiar la referencia.

Para diferenciar entre una llamada al constructor a traves de un objeto (`$objref->new()`) y una llamada como método de una clase (`Biblio::Doc->new()`) se usa la función `ref()` (véase la subsección 5.3). Cuando se llama a través de una referencia a un objeto, la función `ref()` devuelve la cadena conteniendo la clase. Si se llama como método de una clase, `ref()` se aplica a una cadena y nos devuelve una cadena vacía. De este modo el constructor puede reconocer el formato de la llamada.

7.4.1. Práctica: Constructores-Copia

Utilizando las ideas esbozadas en el párrafo anterior, reescriba el constructor de la clase `Biblio::Doc->new()` descrita en la sección 7.3 (Puede encontrar una copia en http://nereida/~lhp/perlexamples/Doc.pm) de manera que cuando sea llamado como método de un objeto produzca una copia del objeto.

Una llamada como `$newobj = $objref->new(arg1 => "nuevo valor1", arg4 => "nuevo valor4")` debería producir un objeto `$newobj` cuyos atributos son iguales a los de `$objref`, salvo que los atributos `arg1` y `arg4` son cambiados a `"nuevo valor1"` y `"nuevo valor4"` respectivamente.

Modo de uso:

```
#!/usr/bin/perl -w -I.
use Biblio::Doc2;
...
$newobj = $obj->new(author => "Gardner", title => "Left and Right in the Universe");
$newobj->print();
```

7.5. Destructores

Perl elimina automáticamente la memoria de un objeto cuando es claro que ya no va a ser utilizado. Un método con el nombre especial `DESTROY` se dispara cuando la memoria asociada con un objeto debe ser eliminada (normalmente por que salimos de su ámbito de existencia).

Observe el siguiente código:

```
$ cat destructors.pl
#!/usr/bin/perl -w

{
my $a = 4;

print $a,"\n";
}
```

```
 8 $a = "objeto apuntado por \$b";
 9 $b = \$a;
10 bless $b;
11
12 sub DESTROY {
13   my $self = shift;
14   printf("\n****\n$$self eliminado a las %s\n", scalar localtime);
15 }
```

Al ejecutarlo obtenemos la salida:

```
$ ./destructors.pl
4

****
objeto apuntado por $b eliminado a las Thu May 20 14:31:42 2004
```

La eliminación de la variable léxica `$a` en la línea 4 no conlleva la llamada a `DESTROY`. sin embargo si que la implica la eliminación del objeto referenciado por `$b`.

Realmente la programación explícita de un *destructor* es requerida en muy pocas ocasiones: cuando el objeto es una interfase con el mundo exterior o cuando contiene estructuras de datos circulares. En esos casos podemos aprovecharlo para suprimir ficheros temporales, romper enlaces circulares, desconectarnos de un "socket" o "matar" ciertos subprocesos que hayamos generado. No es necesario liberar explícitamente memoria, simplemente debemos colocar ahi cualquier código de liberación de recursos y limpieza que sea razonable para la finalización del objeto en cuestión.

Mientras que un constructor tiene un nombre arbitrario, un destructor no. Esto es debido a que los destructores se llaman automáticamente por el sistema de recolección de basura ("garbage collection") de Perl. En general, Perl utiliza el convenio de que, aquellas funciones que son llamadas automáticamente se escriben en mayúsculas. Otras funciones que son llamadas implícitamente son `BEGIN`, `END` y `AUTOLOAD`.

Una sutileza con los destructores aparece cuando se ha escrito una rutina `AUTOLOAD` en el paquete. Como se sabe, `AUTOLOAD` es ejecutada *siempre* que no este definido el método invocado. La clave aquí es que "*siempre*" incluye también las llamadas a los destructores.

Obsérve la queja producida al ejecutar el programa cliente descrito en la sección 7.2:

```
Has llamado a Biblio::Doc::DESTROY() y no existe!
```

Una solución simple para esta situación consiste en definir un método `DESTROY` vacío.

7.6. Herencia

Una clase informa a Perl que desea heredar de otra clase añadiendo el nombre de esa clase a la variable `ISA` de su paquete. Por ejemplo:

```
package A;
@ISA = ( "B" );
```

indica que la clase `A` hereda de la clase `B`.

La *herencia* en Perl determina el recorrido 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`. Para ser mas precisos, cuando Perl busca por una llamada a un método como `$obj->method()`, realiza la siguiente secuencia de búsqueda:

1. Si la clase en la cuál el objeto fué bendecido (digamos `MyClass`) tiene una subrutina `method` se llama

2. Si no, si existe un vector `@ISA`, para cada una de las clases en el vector `@ISA` se repiten los pasos 1 y 2

3. Si no, si la clase `UNIVERSAL` (véase la sección 7.6.1) tiene un método con ese nombre, se le llama

4. Si no, si la clase actual `MyClass` tiene un método `AUTOLOAD` se le llama

5. Si no, si una de las clases antepasadas de esta (una vez mas búscadas en orden primero profundo) contiene un método `AUTOLOAD`, se le llama

6. Si no, si la clase `UNIVERSAL` tiene un método `AUTOLOAD`, se le llama

7. Si no, se abandona la búsqueda con un mensaje de error

Esta búsqueda sólo se hace una vez por método. Una vez localizado el método se utiliza una "cache" para acceder al método rápidamente. Si el vector `@ISA` o el vector `@ISA` de cualquiera de los antepasados de la clase es modificado, se limpia la "cache".

7.6.1. La clase `UNIVERSAL`

Como vemos existe una clase especial denominada clase `UNIVERSAL` de la cual implícitamente hereda toda clase. Esta clase provee los métodos *isa*, *can* y *VERSION* (véase la sección 6.5). Es posible añadir métodos o atributos a *UNIVERSAL*.

El método `isa` nos permite saber si una clase hereda de otra:

```
if ($a->isa("B")) { # El objeto a es de la clase B ... }
```

El método `isa` memoriza los valores que retorna, de manera que una vez que conoce un par no necesita realizar una segunda búsqueda. Eso significa que una modificación dinámica del vector `@ISA` puede dar lugar a que el método retorne valores obsoletos.

Hay ocasiones en las que lo que nos preocupa no es tanto a que clase pertenece un objeto como saber si dispone de un cierto método. El método `can` devuelve verdadero si el objeto puede llamar al método solicitado:

```
if ($a->can("display_object")) { # el objeto dispone del método ... }
```

De hecho, el valor que devuelve `can` es una referencia al método por el que se pregunta.

7.6.2. Práctica: Ancestros de un Objeto

Escriba un método `ancestors` que devuelve una lista con la jerarquía de clases de la que hereda el objeto o clase con la que `ancestors` es llamado. Introduzca dicho método en la clase `UNIVERSAL`. Compruebe su funcionamiento.

7.6.3. Práctica: Un Método Universal de Volcado

Utilizando el módulo `Data::Dumper` (descárgelo de CPAN si es necesario) o bien la librería `dumpvar.pl` descrita en la sección 5.7 escriba un método `dump` que vuelque los contenidos del objeto e incorpórelo a la clase `UNIVERSAL`. Puede serle útil repasar la función `caller` descrita en la sección 1.9.11. Compruebe el buen funcionamiento del método.

7.6.4. Ejercicio: Búsqueda de Métodos

Puesto que los constructores son métodos, son buscados siguiendo el mecanismo descrito en la sección 7.6. Considere el código en la tabla 7.1. Las clases en el ejemplo heredan según el esquema que se muestra en la figura 7.2. ¿Cuál es la salida del programa? ¿A que clase pertenece el objeto `$a`? Esto es, ¿En que paquete es bendecido? ¿A que clase pertenece el objeto `$b`? ¿Cual es la diferencia entre bendecir con un argumento y con dos argumentos?

```
#!/usr/bin/perl -w

package B;

package C;
sub new {
  my ($self, %args) = @_;
  print "C::new\n";
  bless { %args },
     ref($self) || $self;
}

package E;
sub new {
  my ($self, %args) = @_;
  print "E::new\n";
  bless { %args };
}

package F;

package G;
@ISA = qw(E C);
```

```
package A;
@ISA = qw(B C);

package D;
@ISA = qw(E F G);

sub new {
  my ($self, %args) = @_;
  print "D::new\n";
  bless
    { %args }, ref($self) || $self;
}

package H;
@ISA = qw(A D);

package main;

my $a = H->new();
print ref($a),"\n";

my $b = G->new();
print ref($b),"\n";
```

Cuadro 7.1: Un programa con una herencia complicada

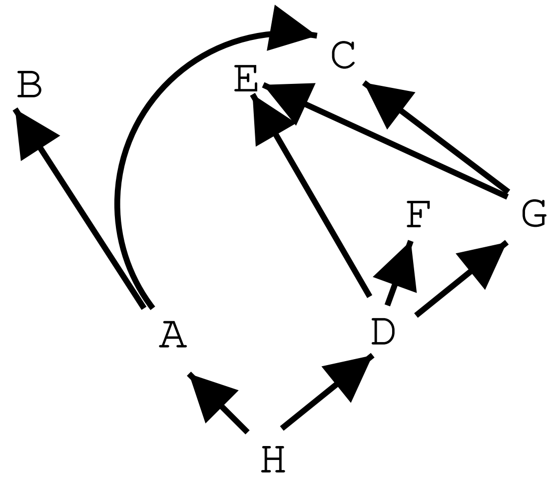

Figura 7.2: Formas de bendición: esquema de herencia del programa

7.6.5. Delegación en la Inicialización

Cuando existe una jerarquía de herencia, es común un diseño que usa una *delegación* planificada en las clases antecesoras. Una solución consiste en separar la creación del objeto de su inicialización. Esto es: el código de un inicializador llama a los correspondientes inicializadores de las clases antepasadas y añade las correspondientes inicializaciones de la clase.

Estudie este ejemplo:

```
#!/usr/bin/perl -d

package _I;
use strict;

sub new {
  my ($class, %args) = @_;
  my $self = bless {}, ref($class) || $class;
  $self->_init(%args);
  return $self;
}

package A;
@A::ISA = qw( _I);

sub _init {
  my ($self, %args) = @_;
  $self->{_a1} = $args{a1};
  $self->{_a2} = $args{a2};
}


package B;
@B::ISA = qw( _I);

sub _init {
  my ($self, %args) = @_;
  $self->{_b1} = $args{b1};
}

package C;
```

```
@C::ISA = qw( _I B A);

sub _init {
  my ($self, %args) = @_;
  $self->A::_init(%args);
  $self->B::_init(%args);
  $self->{_c1} = $args{c1};
}

package main;

C->new(a1=>"a1", a2=>"a2", b1=>"b1", c1=>"c1");
```

La idea es que la clase `_I` provee un constructor genérico. Las otras clases heredan dicho constructor y no lo reescriben. La figura 7.3 muestra el esquema de herencia y el de delegación (flechas punteadas) usado para el método `_init`.

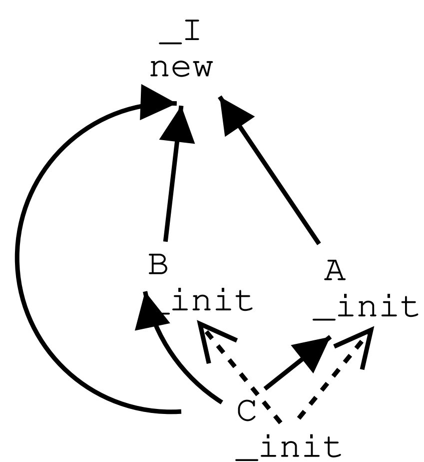

Figura 7.3: Esquema de herencia/delegación del programa

Este esquema es posiblemente mas usual que el reemplazamiento completo del método por uno nuevo.

Véase una ejecución con el depurador:

```
$ ./inicializadores.pl
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

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

A::(./inicializadores.pl:14):   @A::ISA = qw( _I);
  DB<1> n
B::(./inicializadores.pl:24):   @B::ISA = qw( _I);
  DB<1>
C::(./inicializadores.pl:32):   @C::ISA = qw( _I B A);
  DB<1>
main::(./inicializadores.pl:43):        C->new(a1=>"a1", a2=>"a2", b1=>"b1", c1=>"c1");
```

el método `new` en la clase `_I` acaba siendo llamado:

```
  DB<1> s
```

```
_I::new(./inicializadores.pl:7):          my ($class, %args) = @_;
  DB<1>
_I::new(./inicializadores.pl:8):          my $self = bless {}, ref($class) || $class;
  DB<1> p $class
C
  DB<2> n
_I::new(./inicializadores.pl:9):          $self->_init(%args);
```

Obsérvese que:

- El objeto ha sido bendecido en la clase llamadora `C`, no en la clase `_I`.
- `new` llama al método `_init` de la clase llamadora (`C->_init(%args)` no al método `_init` de la clase `_I`.

```
  DB<2> s
C::_init(./inicializadores.pl:35):        my ($self, %args) = @_;
  DB<2>
C::_init(./inicializadores.pl:36):        $self->A::_init(%args);
```

Las llamadas en las líneas 36 y 37 explicitan el paquete: `$self->A::_init(%args)` y `$self->B::_init(%args)`. Normalmente, cuando Perl comienza la busca en el paquete en el cual ha sido bendecido el objeto, pero si en la llamada con el operador flecha se detalla el *nombre completo del método* (fully qualified name) Perl ignora la clase del objeto y comienza la búsqueda en la tabla de símbolos del paquete prefijo.

```
  DB<2>
A::_init(./inicializadores.pl:17):        my ($self, %args) = @_;
  DB<2>
A::_init(./inicializadores.pl:18):        $self->{_a1} = $args{a1};
  DB<2>
A::_init(./inicializadores.pl:19):        $self->{_a2} = $args{a2};
  DB<2>
C::_init(./inicializadores.pl:37):        $self->B::_init(%args);
  DB<2>
B::_init(./inicializadores.pl:27):        my ($self, %args) = @_;
  DB<2>
B::_init(./inicializadores.pl:28):        $self->{_b1} = $args{b1};
  DB<2>
C::_init(./inicializadores.pl:38):        $self->{_c1} = $args{c1};
  DB<2>
_I::new(./inicializadores.pl:10):         return $self;
  DB<2> p %$self
_a2a2_b1b1_c1c1_a1a1
  DB<3>
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.
```

El proceso de destrucción es equivalente. Puesto que se trata de métodos, se sigue el mismo algoritmo de búsqueda. Si queremos que los destructores sean llamados de manera organizada, es responsabilidad del programador el hacerlo.

7.6.6. Diamantes

El método de delegación expuesto anteriormente falla si una clase hereda de un ancestro por dos caminos distintos (en la jerga esto se llama un *diamante*). En tal caso existe el riesgo de que una llamada por delegación a un `_init` de un antepasado se repita dos veces (una por cada camino hasta el antepasado) para el mismo objeto. Una solución es rastrear que inicializadores han sido visitados y evitar las subsiguientes visitas. Por ejemplo, si la clase `C` del ejemplo anterior es un candidato a inicializaciones repetidas (esto es, si puede ser la cúspide de un diamante), la podemos proteger usando el condicional que aparece en la línea 6 (Véase [14] para mas detalles):

```
 1  package C;
 2  @C::ISA = qw( _I B A);
 3
 4  sub _init {
 5    my ($self, %args) = @_;
 6    return if $self->{_init}{__PACKAGE__}++;
 7    $self->A::_init(%args);
 8    $self->B::_init(%args);
 9    $self->{_c1} = $args{c1};
10  }
```

La macro `__PACKAGE__` es igual al nombre del paquete actual (en este caso el paquete `C`). El uso de `__PACKAGE__` puede ser de ayuda si por alguna razón decidimos copiar el código de `_init` en otro paquete: No tenemos que sustituir las apariciones de `C` por el nombre del nuevo paquete.

Una estrategia parecida puede aplicarse para evitar la llamada reiterada a destructores cuando se producen estructuras jerarquicas en diamante.

7.6.7. La notación `SUPER`

La delegación de parte de las fase de inicialización a las clases antepasadas es común cuando se usa herencia. Es conveniente evitar la codificación "dura" de clases antepasadas en el código como se hizo en la sección 7.6.5. El problema con una llamada de la forma

```
$self->B::method();
```

es que si se cambia el nombre de la clase o se modifica la jerarquía de clases habrá que reescribir esa parte del código. Es posible en ocasiones evitar dicha codificación con el nombre explícito de la clase madre utilizando un identificador de paquete `SUPER` el cual nos permite referirnos a las clases madres de la clase actual:

```
$self->SUPER::method();
```

Mas que un sinónimo de `B`, lo que hace `SUPER` es indicarle a Perl que la búsqueda del método debe realizarse desde el vector `@ISA` de la clase en la que *el método está siendo llamado*. Veamos un ejemplo:

```
~/perl/src> cat A.pl
#!/usr/bin/perl -w -I.
use A;

$x = A->new();
print ref($x),"\n";
$x->x();
```

Los módulos implicados son:

```
$ cat ./A.pm
package A;
use B;
use C;

@ISA = ("B", "C");

sub x {
  $self = shift;

  $self->SUPER::t();
}

sub t {
  print "t de A\n";
}


1;
```

```
$ cat B.pm
package B;

sub new {
  my $class = shift;
  bless {}, $class;
}

1;
```

```
$ cat C.pm
package C;
sub new {
  my $class = shift;
  bless {}, $class;
}

sub t {
  print "C\n";
}

1;
```

La ejecución da como resultado:

```
/perl/src> A.pl
A
C
```

La llamada `$x->x()` produce una llamada a `$self->SUPER::t()`. Dado que la búsqueda comienza en las clases especificadas en el vector `@ISA` de la clase `A` y que la clase `B` no tiene ningún método denominado `t()`, se encuentra el método `t()` de la clase `C`.

7.6.8. Ejercicio: SUPER

Recuerde que `SUPER` indica que la búsqueda del método debe realizarse desde el vector `@ISA` de la clase en la que *el método está siendo llamado y no de la clase del objeto que se ha usado en la llamada.* Dado el programa en la tabla 7.2 cuyo esquema aparece en la figura 7.4, explique como se producirán las llamadas y cual será la salida.

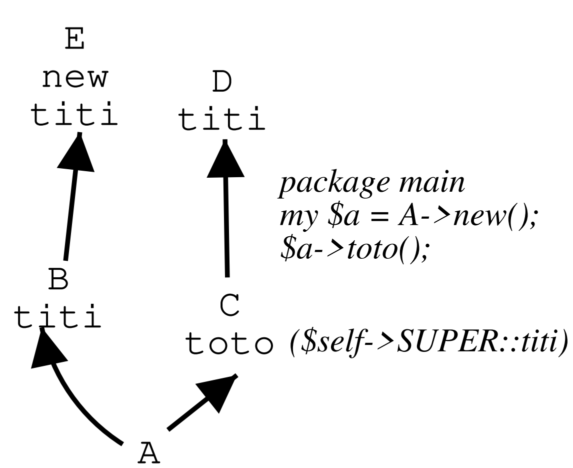

Figura 7.4: Esquema del programa

```perl
#!/usr/bin/perl -w
use strict;

package E;

sub new {
  my ($class, %args) = @_;
  my $self =
    bless {}, ref($class)
              || $class;
  return $self;
}

sub titi {
  print "E\n";
}

package D;
sub titi {
  print "D\n";
}
```

```perl
package B;
@B::ISA = qw(E);

sub titi {
  print "B\n";
}

package C;
@C::ISA = qw(D);

sub toto {
my $self = shift;
  print "C::toto\n";
  print "Clase del Objeto: ",
         ref $self,"\n";
  $self->SUPER::titi();
}

package A;
@A::ISA = qw(B C);


package main;

my $a = A->new();
   $a->toto();
```

Cuadro 7.2: Ejemplo de uso de `SUPER`

7.6.9. Métodos Abstractos

Un *método abstracto* es uno que, mas que proveer un servicio representa un servicio o categoría. La idea es que al definir una 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 interfase que indica la necesidad de definir su funcionalidad en las clases descendientes, pero que no se define en la clase base.

Para conseguir este efecto, lo normal en Perl es que nos aseguremos que nuestro método abstracto produce una excepción con el mensaje de error adecuado. Cuando el número de métodos abstractos es grande puede suponer un ahorro utilizar un método genérico como este:

```perl
package Abstract;

sub ABSTRACT {
  my $self = shift;
  my ($file, $line, $method) = (caller(1))[1..3];

  die("call to abstract method ${method} at $file, line $line\n");
}

1;
```

Obsérve el uso de la función `caller` introducida en la sección 1.9.11 para determinar la localización exacta de la llamada.

7.6.10. Práctica: Herencia

Escriba una clase `Biblio::Doc::Article` que herede de la clase `Biblio::Doc` descrita en la sección 7.3. Ademas de los atributos de esta última, `Biblio::Doc::Article` deberá tener los atributos: `journal`, `volume` y `pages`. Procure que los métodos `new` y `AUTOLOAD` de la clase paterna sean suficientes (si es necesario reescribalos).

Para ello, declare en la nueva clase un hash con los atributos adicionales `journal`, `volume` y `pages` conteniendo los valores por defecto y el modo de acceso (`read/write`).

Algunas observaciones para escribir esta práctica con mas posibilidades de éxito:

El constructor `new` en la clase `Biblio::Doc` descrito en la sección 7.3 venía dado por:

```
1 sub new {
2   my $class = shift;
3   my $self = {};
4
5   bless $self, ref($class) || $class;
6
7   $self->_incr_count();
8   $self->_init(@_);
9
10  return $self;
11 }
```

Observe que puesto que en la línea 5 se usa la versión de `bless` con dos argumentos, el objeto es bendecido en la clase llamadora, de manera que una llamada a `Biblio::Doc::Article->new` produce un objeto de la clase `Biblio::Doc::Article`. Nótese también que en la llamada de la línea 8, dado que `$self` ya ha sido bendecido en `Biblio::Doc::Article`, la subrutina `_init` que es llamada es, si existe, la de la clase `Biblio::Doc::Article`. Si no existe sería llamada la de `Biblio::Doc`, la cual ha sido escrita como:

```
sub _init {
  my ($self, %args) = @_;
  my %inits;
  my ($i, $j);

  for $i ($self->_standard_keys) {
    $j = $i;
    $j =~ s/_//;
    $inits{$i} = $args{$j} || $self->_default_for($i);
  }
  %$self = %inits;
}
```

Esta subrutina confía en la llamada `$self->_standard_keys` para obtener las claves del objeto. Es posible entonces escribir un método `Biblio::Doc::Article::_standard_keys` el cual proporcione las claves nuevas junto con las que produce el método equivalente en la clase padre `SUPER::_standard_keys`. Esta estrategia es posible por que se envolvió la estructura de datos `%_defaults`

```
my %_defaults = (                # Default   Access
                    _identifier => ["unknown",'read'],
                    _author     => ["unknown",'read/write'],
                    _publisher  => ["unknown",'read'],
                    _title      => ["unknown",'read'],
```

```
                _year       => ["unknown",'read'],
                _url        => ["unknown",'read/write']
              );
```

con los métodos `_standard_keys` y `_default_for`. Si en la línea 6 del código de `_init` hubieramos escrito:

`for $i (keys %_defaults)`

la resolución del problema por delegación sería mas complicada.

Algo análogo puede hacerse con la subrutina `_default_for`.

De la misma forma, la función `AUTOLOAD` de la clase `Biblio::Doc` es:

```
 1 sub AUTOLOAD {
 2   no strict "refs";
 3   my $self = shift;
 4   if (($AUTOLOAD =~ /\w+::\w+::get(_.*)/) && ($self->_accesible($1,'read'))) {
 5     my $n = $1;
 6     return unless exists $self->{$n};
 7 
 8     # Declarar el metodo get_*****
 9     *{$AUTOLOAD} = sub { return $_[0]->{$n}; };
10 
11     return $self->{$n};
12   } elsif (($AUTOLOAD =~ /\w+::\w+::set(_.*)/) && ($self->_accesible($1,'write'))) {
13     my $n = $1;
14     return unless exists $self->{$n};
15     $self->{$n} = shift;
16 
17     # Declarar el metodo set_*****
18     *{$AUTOLOAD} = sub { $_[0]->{$n} = $_[1]; };
19   } else {
20     @_ = map { "\"".$_."\""; } @_;   # Comillas a los argumentos...
21     print "Has llamado a $AUTOLOAD(",join(", ",@_),")\n";
22   }
23 }
```

Observe que la subrutina `_accesible` que es llamada en las líneas 4 y 12 es la de `Biblio::Doc::Article`. Una buena estrategia al escribir dicha rutina es usar *delegación*: `Biblio::Doc::Article->_accesible` comprueba la accesibilidad de las claves del objeto que son específicas de `Biblio::Doc::Article` y delega en el método `SUPER::_accesible` para los restantes atributos.

7.7. ¿Atados? ó ¿Corbatas? ó `Ties`

La función `tie` asocia una variable con un conjunto de rutinas de un determinado paquete, de manera que a partir del momento en que se realiza la unión (`tie`) el acceso a la variable es gobernado por los métodos en el paquete. Por ejemplo, para una variable escalar escribiríamos:

```
tie $var, "Package::Name";
```

Los métodos dentro del paquete deben tener nombres especiales: `TIESCALAR`, `FETCH`, `STORE` y `DESTROY`.

El método `TIESCALAR` es invocado cuando se llama a la función `tie`. Se le pasan los mismos argumentos que se hayan pasado a la función `tie`, excepto por la variable en cuestión `$var`. Así, la llamada anterior se traduce en una llamada a `Package::Name->TIESCALAR()`. Si, por ejemplo atamos una variable con:

```
tie $var, "Package::Name", $arg1, $arg2, $arg3
```

Los argumentos adicionales se pasan a `TIESCALAR`, produciendo la llamada:

```
Package::Name->TIESCALAR($arg1, $arg2, $arg3)
```

La tarea de `TIESCALAR` es crear un nuevo objeto de la clase apropiada y devolver una referencia al mismo. El objeto implementa el estado interno de la variable original.

El método `FETCH` se llama siempre que se lee el valor de la variable. Se le pasa como único argumento una referencia al objeto que implementa la variable (el que fué creado por el constructor `TIESCALAR`). El valor devuelto por `FETCH` se usa como valor de la variable.

La subrutina `STORE` se llama siempre que se actualiza el valor de `$var`. Se le pasan dos argumentos: la referencia al objeto que implementa la variable y el nuevo valor a almacenar. No devuelve valor alguno.

El destructor, `DESTROY`, es llamado cuando `$var` desaparece (técnicamente, cuando el contador de referencias a `$var` que internamente lleva Perl alcanza cero). No suele ser necesaria, pero puede ser usada para realizar las labores de limpieza que la implementación requiera. El destructor es llamado también cuando se rompe explicitamente la atadura o compromiso usando la función `untie`:

```
untie $var;
```

7.7.1. Relaciones entre Persistencia y `ties`

El módulo *Multi level DBM MLDBM* permite guardar estructuras de datos Perl en ficheros DBM (Véase 1.8.11). Para tenerlo instalado es necesario tener previamente instaladas las rutinas *Berkeley Database Manager*. El módulo utiliza DBM, proveyendo la funcionalidad para serializar estructuras de datos anidadas. DBM es una librería que maneja tablas hash en disco. La librería ha dado lugar a un buen número de variantes: SDBM, NDBM, GDBM, etc las cuales pueden ser accedidas a través de los correspondientes módulos Perl, los cualés hacen uso de `tie` para proporcionar un acceso transparente a la tabla almacenada en el disco.

La capacidad de un sistema o módulo para hacer que la vida o ámbito de los datos de una aplicación se prolongen mas allá de la ejecución de la aplicación se conoce con el nombre de *persistencia*. Se conoce por *serialización* a los procesos encargados de empaquetar estructuras de datos anidadas como arrays de hashes etc. en una estructura plana y lineal. Es deseable que la representación de los datos sea independiente de la máquina, evitando problemas de representación de enteros y flotantes o del orden de los bytes.

Véase un ejemplo de uso del módulo `MLDBM`:

```
#!/usr/bin/perl -w
use MLDBM qw( DB_File Data::Dumper );
use Fcntl;

unlink 'mldbmtest.dat';

tie my %db1, 'MLDBM', 'mldbmtest.dat', O_CREAT | O_RDWR, 0666
    or die "No se pudo inicializar el fichero MLDBM: $!\n";

%db1 = (
    'alu2511' => {
        nombre => 'Josefina Fernández Pérez',
        tel => '922 00 00 00',
        fecha => '22/07/84'
    },
    'alu2233' => {
        nombre => 'Ana Feliú Forner',
        tel => '922 00 11 22',
        fecha => '14/06/85'
```

```
    }
);

untie %db1;

tie my %db2, 'MLDBM', 'mldbmtest.dat', O_RDWR, 0666
    or die "No se pudo inicializar el fichero MLDBM: $!\n";

print Data::Dumper->Dump( [ \%db2 ] );

untie %db2;

exit;
```

Cuando se ejecuta se obtiene el siguiente resultado:

```
 ./mldbmtest2
$VAR1 = {
  'alu2511' => {
     'fecha' => '22/07/84',
     'tel' => '922 00 00 00',
     'nombre' => 'Josefina Fernández Pérez'
  },
  'alu2233' => {
     'fecha' => '14/06/85',
     'tel' => '922 00 11 22',
     'nombre' => 'Ana Feliú Forner'
  }
};
```

7.7.2. Volcado automático de una variable

Como ejemplo, supongamos que queremos depurar una variable de paquete. Podemos modificar la interface de acceso a dicha variable para que, cada vez que activemos el "modo depuración" se imprima el valor de dicha variable. La utilización se muestra con el programa de uso:

```
#!/usr/bin/perl -w -I.
use Inspector;

  $x = 10;
  Inspector->scalar($x, 1);
  $x = 20;
  Inspector->debug($x,0);
  $x = 30;
  $x += 5;
  Inspector->debug($x,1);
  $x += 10;
```

al ejecutarse da como resultado:

```
~/perl/src> use_Inspector.pl
val=20 use_Inspector.pl:6:main
val=45 use_Inspector.pl:11:main
```

La idea es que una llamada a `Inspector->scalar($x, 1)` ata la variable `$x` (que será su argumento `$_[1]`) a la clase `Inspector` (argumento `$_[0]`). El tercer argumento lógico, indica si queremos activar o desactivar la depuración de la variable. Una vez atada, podemos usar el método `Inspector->debug` para controlar el rastreo de la variable.

```
package Inspector;

sub TIESCALAR {
  my ($class, $val, $debug) = @_;
  bless { val => $val, debug => $debug }, $class;
}

sub FETCH {
  my $impl = shift;
  return $impl->{val};
}

sub STORE {
  my ($implementation, $newval) = @_;
  $implementation->{val} = $newval;
  if ($implementation->{debug}) {
    my ($cur_pkg, $cur_file, $cur_line) = caller;
    print STDERR "val=$implementation->{val} $cur_file:$cur_line:$cur_pkg\n";
  }
}

sub scalar {
  my ($class, $var, $debug) = @_;
  tie $_[1], $class, $_[1], $debug;
}

sub debug {
  my $impl = tied($_[1]);
  my $deb = $_[2];

  $impl->{debug} = $deb;
}

1;
```

La llamada a `Inspector->scalar($x, 1)` conlleva a través de `tie` una llamada a la función `TIESCALAR` que es la que construye el "hash" anónimo. La entrada `val` guarda el valor de la variable. La entrada `debug` indica si los valores de la variable deben o no ser volcados a `STDERR` cuando se cambia su valor. Por último `TIESCALAR` bendice el "hash" y devuelve su referencia como el objeto que reimplementa a la variable original.

Toda modificación de la variable se hace a través de `STORE`, la cuál después de cumplir con la reglamentaria asignación imprime los valores por `STDERR` si es el caso.

Puesto que no hemos provisto de métodos de acceso a la implementación interna de la variable hemos tenido que usar la función `tied`. Esta función recibe una variable como argumento y, si esta "atada" a un objeto devuelve una referencia al objeto. En otro caso devuelve `undef`. Asi pues, `debug` llama primero a `tied` y después establece el valor de la entrada `debug` del "hash".

7.7.3. Acceso a las variables de entorno

Perl proporciona acceso a las variables de entorno a través del hash `%ENV`. En el ejemplo que sigue *ataremos* variables escalares Perl a las correspondientes variables de entorno con el mismo nombre, de manera que la modificación de la variable Perl atada conlleve la modificación de la variable de entorno:

```
 1 package Myenv;
 2
 3 sub import {
 4     my ($callpack) = caller(0); # nombre del paquete que usa a esta rutina
 5     my $pack = shift;
 6     my @vars = grep /^[A-Za-z_]\w*$/, (@_ ? @_ : keys(%ENV));
 7     return unless @vars;
 8
 9     foreach (@vars) {
10       tie ${"${callpack}::$_"}, Myenv, $_;
11     }
12 }
13
14 sub TIESCALAR {
15     bless \($_[1]);
16 }
17
18 sub FETCH {
19     my ($self) = @_;
20     $ENV{$$self};
21 }
22
23 sub STORE {
24     my ($self, $value) = @_;
25     if (defined($value)) {
26       $ENV{$$self} = $value;
27     } else {
28       delete $ENV{$$self};
29     }
30 }
31
32 1;
```

La función `import` obtiene en la línea 4 el nombre del paquete que usa `Myenv`. Después, si se han explicitado argumentos en el pragma `use` se dejan en `@vars` (línea 6) y si no `@vars` es inicializada con todas las claves del hash `%ENV`. Repase la sección 6.6 para mas detalles sobre el proceso de importación. En el bucle de las líneas 9-11 instalamos en la tabla de símbolos del paquete usuario las entradas a las variables de entorno.

Sigue el programa de uso:

```
#!/usr/bin/perl -d
use Myenv qw(HOME);

print "home = $HOME\n";
$HOME = "/tmp/";
system 'echo $HOME';
```

y su ejecución con el depurador:

```
$ ./env.pl
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.
```

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

main::(./env.pl:4):     print "home = $HOME\n";
  DB<1> s
Myenv::FETCH(Myenv.pm:19):          my ($self) = @_;
  DB<1>
Myenv::FETCH(Myenv.pm:20):          $ENV{$$self};
  DB<1> x $self
0  Myenv=SCALAR(0x811fa8c)
   -> 'HOME'
  DB<2>
home = /home/lhp
main::(./env.pl:5):     $HOME = "/tmp/";
  DB<2> s
Myenv::STORE(Myenv.pm:24):          my ($self, $value) = @_;
  DB<2>
Myenv::STORE(Myenv.pm:25):          if (defined($value)) {
  DB<2>
Myenv::STORE(Myenv.pm:26):              $ENV{$$self} = $value;
  DB<2>
main::(./env.pl:6):     system 'echo $HOME';
  DB<2>
/tmp/
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.
```

7.7.4. Práctica: Tie Escalar

Escriba una clase que permita conocer el número de asignaciones realizadas a una variable dada.

7.8. Sobrecarga de Operadores

Supongamos que queremos escribir un módulo que permite trabajar con números en punto flotante de tamaño arbitrario. Su uso sería algo así:

```
#!/usr/bin/perl -w
use strict;
use Math::BigFloat;

my $a = Math::BigFloat->new('123_456_789_123_456_789');
my $y = $a->copy()/1_000_000_000;

print "a = $a\n";
print "-a = ",-$a,"\n";
print "y = $y\n";
print "a+y = ",$a+$y,"\n";
```

cuya ejecución nos da:

```
$ ./bigfloat.pl
a = 123456789123456789
-a = -123456789123456789
y = 123456789.123456789
a+y = 123456789246913578.123456789
```

y queremos que el módulo, como ilustra el ejemplo, *sobrecargue* las operaciones binarias y unarias usuales así como el uso de las constantes. Los mecanismos para escribir un módulo como este los proporciona el módulo `overload.pm` debido a Ilya Zakharevich, el cual se incluye en la distribución estandard de Perl. Este módulo permite la sobrecarga de operadores. Para sobrecargar los operadores para una clase dada, hay que pasarle a la sentencia `use` una lista de pares `operador, referencia a código`:

```
package Math::BigFloat;

use overload "*"   => \&fmul,
             "+"   => "fadd",
             "neg" => sub { Math::BigInt->new($_[0]->fneg()) };
```

Cada pareja consiste de una clave, que especifica el operador a sobrecargar, y una referencia a una subrutina, que será invocada cuando se encuentre el operador. La clave `neg` corresponde al operador de negación unaria.

La clave puede ser cualquiera de las reseñadas en la tabla 7.8. La referencia a la subrutina puede ser

- una referencia a una subrutina con nombre,
- una referencia simbólica o
- una referencia a una subrutina anónima.

La subrutina de implementación es llamada cada vez que un objeto de la clase en cuestión (en el ejemplo la clase `Math::BigFloat`) es un operando del operador correspondiente. Por ejemplo, si ejecutamos con el depurador el programa:

```
$ cat -n ./bigfloat2.pl
1  #!/usr/bin/perl -d
2  use strict;
3  use Math::BigFloat;
4
5  my $a = Math::BigFloat->new('123_456_789_123_456_789');
6  my $y = $a->copy()/1_000_000_000;
7
8  print "a+y = ",$a+$y,"\n";
```

observaremos como la subrutina asociada con el `+` es llamada:

```
$ ./bigfloat2.pl
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

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

main::(./bigfloat2.pl:5):       my $a = Math::BigFloat->new('123_456_789_123_456_789');
  DB<1> n
main::(./bigfloat2.pl:6):       my $y = $a->copy()/1_000_000_000;
  DB<1>
main::(./bigfloat2.pl:8):       print "a+y = ",$a+$y,"\n";
  DB<1> s
```

Hemos pulsado `s` para entrar en la subrutina . . .

```
Math::BigInt::CODE(0x82f9acc)(/usr/share/perl5/Math/BigInt.pm:50):
50:     '+'     =>      sub { $_[0]->copy()->badd($_[1]); },
  DB<1> p "@_"
123456789123456789 123456789.123456789
```

Observe los dos argumentos: los dos números grandes. Seguimos ...

```
  DB<2> n
Math::BigInt::CODE(0x830c9f0)(/usr/share/perl5/Math/BigInt.pm:110):
110:    '""' => sub { $_[0]->bstr(); },
```

esta es la conversión necesaria del resultado a una cadena para ser impresa, continuamos ...

```
  DB<2>
a+y = 123456789246913578.123456789
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.
```

Si la sobrecarga fué especificada a través de una referencia a subrutina se usa una llamada como subrutina mientras que si se especificó como referencia simbólica se usa la sintáxis de método. Por ejemplo, si `$a` y `$b` son dos objetos `Math::BigFloat`, para las declaraciones

```
package Math::BigFloat;

use overload "*"   => \&fmul,
             "+"   => "fadd",
             "neg" => sub { Math::BigInt->new($_[0]->fneg()) };
```

tendríamos los siguientes ejemplos de traducciones:

`$a*$b`	`Math::BigFloat::fmul($a, $b, "")`
`$a+$b`	`$a->fadd($b, "")`
`-$a`	`(sub { Math::BigFloat->new($_[0]->fneg()) })->($a, undef, "")`

La diferencia entre proporcionar una referencia a una subrutina o un nombre de método está relacionada con la herencia. Cuando se proporciona una referencia estamos asegurándonos de que se llama a la rutina con ese nombre, desactivando el mecanismo de búsqueda asociado con la herencia. Cuando proporcionamos un nombre de método, el operador sobrecargado va a llamar a ese método siguiendo los mecanismos de búsqueda asociados con la herencia (véase la sección 7.6).

Obsérvese que, en cualquier caso, la subrutina de implementación es llamada siempre con tres argumentos:

1. El primer operando
2. El segundo operando (`undef` si no existe)
3. Un `flag` indicando cuando los operandos fueron intercambiados

La necesidad del `flag` proviene del requerimiento de que el primer argumento debe ser un objeto de la clase sobrecargada (en el ejemplo la clase `Math::BigFloat`). Si Perl detecta una expresión de la forma `4+$a` la traduce por `$a->fadd(4,1)`, donde el segundo argumento avisa de la inversión producida.

Es por esto que, para operaciones no conmutativas como la resta o la divisón, la función de implementación suele comenzar asi:

```
sub substract {
  my ($op1, $op2, $reversed) = @_;
  ($op1, $op2) = ($op2, $op1) if $reversed;
```

```
  if (UNIVERSAL::isa($op1, 'Math::BigFloat') {
    ... # $op1 es un objeto
  }
  ...
}
```

Veamos otro ejemplo, en el que el objeto esta a la derecha y a la izquierda tenemos una constante:

```
$ cat ./bigfloat3.pl
#!/usr/bin/perl -d
use strict;
use Math::BigFloat;

my $a = Math::BigFloat->new('123_456_789_123_456_789');

print "123_456_789_123_456_789-a = ",'123_456_789_123_456_789'-$a,"\n";
```

Al ejecutar, tenemos:

```
$ ./bigfloat3.pl
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

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

main::(./bigfloat3.pl:5):       my $a = Math::BigFloat->new('123_456_789_123_456_789');
  DB<1> n
main::(./bigfloat3.pl:7):       print "123_456_789_123_456_789-a = ",'123_456_789_123_456_789'-$a,"\n";
  DB<1> s
Math::BigInt::CODE(0x82f997c)(/usr/share/perl5/Math/BigInt.pm:47):
47:     '-'     =>      sub { my $c = $_[0]->copy; $_[2] ?
48:                        $c->bneg()->badd($_[1]) :
49:                        $c->bsub( $_[1]) },
  DB<1> p "@_"
123456789123456789 123_456_789_123_456_789 1
```

Vemos como el tercer argumento esta a `1` y como la subrutina anónima que trata el caso del menos binario convierte la expresión `$a-$b` en `(-$b)+$a` en el caso en que el objeto es el segundo término.

Continuamos la ejecución:

```
  DB<2> c
123_456_789_123_456_789-a = 0
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.
```

Observe que, en general se espera que los operadores sobrecargados puedan manejar la operación de números y objetos de la clase como en `4+$a`. Por tanto nuestro código debe manipular no solo la operación de objetos sino también la de objetos y números. El ejemplo anterior del método `substract`, a través del uso del método `UNIVERSAL::isa` (véase sección 7.6.1) muestra una forma de hacerlo.

7.8.1. Propagación de la Sobrecarga

El módulo `overload.pm` asume las relaciones habituales entre operadores y aprovecha este conocimiento. Así, si se da una implementación para el `-` binario, el automáticamente sobrecargará el operador de asignación `-=`, los dos de decremento (`- -`) y el `-` unario (`-$x = 0 - $x`).

<table>
<tr><td>Categoría</td><td>Operadores / Claves</td></tr>
<tr><td>Aritmética</td><td><code>+ - * / % ** x . neg</code></td></tr>
<tr><td>Bit</td><td><code><< >> & | ^ ~</code></td></tr>
<tr><td>Asignación</td><td><code>+= -= *= /= %= **= <<= >>= x= .= ++ --</code></td></tr>
<tr><td>Comparación</td><td><code>< <= > >= == != <=> lt le gt ge eq ne cmp</code></td></tr>
<tr><td>Funciones</td><td><code>atan cos sin exp abs log sqrt</code></td></tr>
<tr><td>Conversiones</td><td><code>q("") 0+ bool</code></td></tr>
<tr><td>Seudo-operadores</td><td><code>nomethod fallback =</code></td></tr>
</table>

Cuadro 7.3: Operadores que pueden ser sobrecargados en Perl. `neg` es la negación unaria

Del mismo modo, si sobrecargamos el operador de comparación (`<=>`), automáticamente sobrecargamos los restantes operadores de comparación, ya que estos pueden ser deducidos de aquél.

Esta sobrecarga automática se realiza si no se declara específicamente la sobrecarga del operador. En caso contrario se usará la definida por el programador.

7.8.2. Búsqueda de la Implementación de un Operador

Normalmente la implementación de un operador sobrecargado se busca en el siguiente orden:

1. Si el operador ha sido explícitamente sobrecargado se llama a la correspondiente subrutina
2. Si no, se mira a ver si el mecanismo de propagación automático descrito en 7.8.1 puede ser aplicado
3. En otro caso, se mira a ver si el seudo operador `nomethod` fué definido. Si es así se llama a la subrutina asociada con `nomethod`.
4. En cualquier otro caso generar una excepción

Este orden puede alterarse si se define el seudo operador `fallback`.

Si `fallback` esta definido y es cierto, la secuencia es la misma que en el caso anterior, excepto que, en vez de generar una excepción, la operación pasa a implementarse siguiendo la conducta normal de Perl para ese operador:

1. Si el operador ha sido explícitamente sobrecargado se llama a la correspondiente subrutina
2. Si no, se mira a ver si el mecanismo de propagación automático descrito en 7.8.1 puede ser aplicado
3. En otro caso, se mira a ver si el seudo operador `nomethod` fué definido. Si es así se llama a la subrutina asociada con `nomethod`
4. Aplicar la conducta normal de Perl para ese operador

Si `fallback` esta definido pero es falso, el paso 2 es eliminado y la secuencia es:

1. Si el operador ha sido explícitamente sobrecargado se llama a la correspondiente subrutina
2. En otro caso, se mira a ver si el seudo operador `nomethod` fué definido. Si es así se llama a la subrutina asociada con `nomethod`.
3. En cualquier otro caso generar una excepción

Este último orden proporciona un mecanismo para aquellas situaciones en las que se quiere evitar el proceso de propagación automática de la sobrecarga. Por ejemplo, supongamos que queremos sobrecargar la operación de diferencia entre días de la semana (lunes, martes, ...) de manera que podamos determinar el número de días entre las mismos. Parece que no tendría sentido usar el operador de negación unaria. Para ello podemos asociar con el operador `neg` una referencia a una subrutina que provoque una excepción:

```
package DaysOfTheWeek;
use overload
  "-"         =>  \delta,
  "nomethod"  =>  sub { croak "No tiene sentido negar un dia de la semana\n" };
```

Sin embargo, este método es ineficiente si el número de operadores autogenerados cuyo uso se quiere prohibir es grande. Una mejor solución es:

```
package DaysOfTheWeek;
use overload
  "-"         =>  \delta,
  "fallback"  =>        0,
  "nomethod"  =>  sub { croak "No tiene sentido $_[3]\n" };
```

Los argumentos que se pasan a la subrutina asociada con `nomethod` son:

1. El primer operando
2. El segundo operando (`undef` si no existe)
3. Un `flag` indicando cuando los operandos fueron intercambiados
4. El operador "realmente" solicitado (`$_[3]`)

7.8.3. Sobrecarga y herencia

La sobrecarga puede combinarse con la herencia. Cualquier subclase heredará los métodos de la clase materna y puede sobreescribir cualquier operador a conveniencia. Hay que tener en cuenta:

- Sobrecarge los métodos por nombre no por referencia.
- Bendiga siempre usando la forma con dos argumentos
- Cuando compruebe la clase de un objeto use `isa()` no utilice `ref($objeto)`

7.8.4. Sobrecarga de las Operaciones de Conversión

Es posible sobrecargar también las operaciones de conversión, esto es, las operaciones que Perl realiza cuando un objeto es convertido a numérico o cadena o lógico.

Para especificar como sobrecargar el operador convertir un objeto a cadena hay que sobrecargar el operador de *stringification* denotado como `"\"\""` ó también `q("")` o incluso `'""'`. La rutina asociada será llamada siempre que el objeto de la clase correspondiente aparezca en un contexto que requiera una cadena.

```
package DaysOfTheWeek;
...
my @_day_name = qw(Sun Mon Tue Wed Thu Fri Sat)
use overload
  q("") => sub { $_day_name[$_[0]->val] };
```

De este modo cuando se cree un objeto `DaysOfTheWeek`:

```
my $day = DaysOfTheWeek->new(3);
```

y se use en un contexto que requiere una cadena como es la función `print`:

```
print $day,"\n";
```

la función asociada con `q("")` será llamada, dando lugar a la salida:

```
Tue
```

Otros contextos en los que se requiere una cadena y, por tanto, se producirá la conversión son:

- Interpolación en una cadena
- Concatenación de cadenas
- Uso como clave de un hash (por ejemplo `$menu{$day}`)

Análogamente se puede controlar el modo en que Perl realiza las conversiones en aquellos contextos en los que requiere un número. El operador encargado de ello es `0+`:

```
package DaysOfTheWeek;
...
my @_day_name = qw(Sun Mon Tue Wed Thu Fri Sat)
use overload
  q("") => sub { $_day_name[$_[0]->val] };
  "0+"  => sub { $_[0]->{val} };
```

Asi la subrutina asociada con `0+` será invocada dondequiera que el objeto sea usado y Perl espere que aparezca un valor numérico:

```
print "*" x $day
```

imprimirá 3 asteriscos puesto que se supone que el segundo operador de `x` es un entero. Otra cosa ocurriría si el operador `x` estuviera ya sobrecargado, en cuyo caso se usaría la correspondiente subrutina asociada. Otros contextos en los que se espera un valor numérico son:

- Donde una función espera como argumento un número
- Siempre que el objeto aparezca como operando del operador de rango (`..`)
- Siempre que el objeto es usado como índice de un vector

Es curioso que, a menos que el seudooperador `fallback` este definido a cierto, los operandos de una operación aritmética no sobrecargada no implican un contexto numérico. Así la expresión `$day+1` no conlleva una conversión de `$day` a numérico.

El operador de sobrecarga `bool` se encarga de la conversión a valores lógicos. La correspondiente subrutina será invocada en cualquier contexto en el que aparezca el objeto de la clase y Perl espere un valor lógico.

Por ejemplo en el módulo `Set::Scalar::Base` debido a Jarkko Hietaniemi y que proporciona acceso al álgebra de conjuntos vemos la siguiente declaración:

```
use overload
    '+'   => \&_union_overload,
    '*'   => \&_intersection_overload,
    '-'   => \&_difference_overload,
    'neg' => \&_complement_overload,
    '%'   => \&_symmetric_difference_overload,
    '/'   => \&_unique_overload,
    'eq'  => \&is_equal,
    '=='  => \&is_equal,
    '!='  => \&is_disjoint,
    '<=>' => \&compare,
    '<'   => \&is_proper_subset,
    '>'   => \&is_proper_superset,
    '<='  => \&is_subset,
    '>='  => \&is_superset,
    'bool'  => \&size;
```

Obsérvese como el manipulador para `bool` hace que un conjunto en un contexto lógico devuelva su tamaño (`\&size`). El siguiente ejemplo ilustra este punto. Consideremos el programa:

```
$ cat ./scalar_sets2.pl
#!/usr/local/bin/perl5.8.0 -d
use Set::Scalar;

$A = Set::Scalar->new(’a’..’z’);

print "$A\n" if $A;
```

Al ejecutarlo podemos contemplar como se llama a `&size` para evaluar la condición `if $A`:

```
$ ./scalar_sets2.pl
Loading DB routines from perl5db.pl version 1.19
Editor support available.
Enter h or ‘h h’ for help, or ‘man perldebug’ for more help.

main::(./scalar_sets2.pl:4):    $A = Set::Scalar->new(’a’..’z’);
  DB<1> n
main::(./scalar_sets2.pl:6):    print "$A\n" if $A;
  DB<1> s
Set::Scalar::Base::size(/usr/local/lib/perl5/site_perl/5.8.0/Set/Scalar/Base.pm:123):
123:        my $self = shift;
  DB<1> T
```

El comando `T` nos permite ver la pila de llamadas:

```
$ = Set::Scalar::Base::size(ref(Set::Scalar), undef, ’’) called from file ‘./scalar_sets2.pl’ line 6
  DB<1>
Set::Scalar::Base::size(/usr/local/lib/perl5/site_perl/5.8.0/Set/Scalar/Base.pm:125):
125:        return scalar keys %{ $self->{’elements’} };
  DB<1> c
(a b c d e f g h i j k l m n o p q r s t u v w x y z)
```

7.8.5. Números Fraccionarios

Sobrecargue los operadores para permitir el uso de números fraccionarios como $\frac{4}{5}, \frac{2}{9}, \ldots$, etc.

Para simplificar la construcción de la clase, parta del siguiente constructor (debido a Dave Cross):

```
sub new {
  my $class = shift;

  my $self;
  if (@_ >= 2) {
    return unless $_[0] =~ /^-?\d+$/ and $_[1] =~ /^-?\d+$/;

    $self->{num} = $_[0];
    $self->{den} = $_[1];
  } elsif (@_ == 1) {
    if (ref $_[0]) {
      if (UNIVERSAL::isa($_[0], $class)) {
        return $class->new($_[0]->{num},
                           $_[0]->{den});
      } else {
        croak "Can’t make a $class from a ",
```

```
          ref $_[0];
      }
    } else {
      return unless $_[0] =~ m|^(-?\d+)/(-?\d+)|;

      $self->{num} = $1;
      $self->{den} = $2;
    }
  } else {
    $self->{num} = 0;
    $self->{den} = 1;
  }
  bless $self, $class;

  $self->normalise;

  return $self;
}
```

Explique las múltiples funcionalidades del constructor. A continuación sigue el código de `normalise`:

```
sub normalise {
  my $self = shift;

  my $hcf = _hcf($self->{num}, $self->{den});

  for (qw/num den/) {
    $self->{$_} /= $hcf;
  }

  if ($self->{den} < 0) {
    for (qw/num den/) {
      $self->{$_} *= -1;
    }
  }
}
```

la subrutina `normalise` llama a `_hcf` (del inglés *highest common factor* o en español máximo común divisor) para permitir la reducción de la fracción:

```
sub _hcf {
  my ($x, $y) = @_;
  ($x, $y) = ($y, $x) if $y > $x;
  return $x if $x == $y;
  while ($y) {
    ($x, $y) = ($y, $x % $y);
  }
  return $x;
}
```

7.8.6. Constantes

El paquete desarrollado en el ejercicio anterior permite trabajar cómodamente con números fraccionarios. Sin embargo, la forma de crearlos implica el uso explícito del constructor:

```
my $d = fraction->new(4,5);
```

Mientras que para un número podemos escribir directamente `$d = 4`, ya que Perl es capaz de deducir el tipo. Sería bueno que Perl pudiera convertir automáticamente constantes de tipo cadena o numéricas directamente al tipo fracción.

Para cambiar el modo en que Perl interpreta las constantes entera, flotantes, cadenas y expresiones regulares podemos crear un conjunto de "manejadores" mediante la subrutina `overload::constant`.

Se espera que dicho manejador devuelve un valor escalar que es usado en lugar de la interpretación normal. Por ejemplo, para usar `overload::constant` en el paquete `Math::BigFloat` para modificar el modo en que las constantes enteras y flotantes se interpretan en Perl haríamos:

```
package Math::BigFloat;
use Math::BigInt;
use overload;

my %_constant_handlers = (
  integer => sub { return Math::BigInt->new($_[0]) },
  float   => sub { return Math::BigFloat->new($_[0]) }
);

sub import { overload::constant %_constant_handlers }
sub unimport { overload::remove_constant %_constant_handlers }
```

Obsérve el uso de `import` (véase sección 6.6) que hace que `overload::constant %_constant_handlers` sea ejecutada en tiempo de compilación cada vez que se usa el módulo.

La subrutina `overload::constant` toma como argumento un hash y espera que las entradas tengan una de las siguientes claves:

- `integer`, indicando el manipulador para enteros decimales,
- `float`, indicando el manejador de números en punto flotante,
- `binary`, indicando el manipulador de constantes octales y hexadecimales,
- `q`, indicando el manipulador para las cadenas constantes (esto es, las cadenas `'...'`, `"..."`, `q{...}`, y `qq{...}`, los argumentos de `tr/.../.../`, o el segundo argumento de una sustitución `s/.../.../`,
- `qr`, indicando el manipulador de una expresión regular (por ejemplo, el primer argumento de `m/.../` o de `s/.../.../`).

El correspondiente valor para cada clave debe ser una referencia a una subrutina. La subrutina es responsable de proporcionar un valor final al tipo particular de constante que esta siendo interpretado. Se le pasan tres parámetros:

- Una cadena conteniendo los caracteres originales en la constante
- El valor que Perl atribuiría a la constante en condiciones normales
- Una cadena indicando el fuente de la constante

La cadena fuente pasada como tercer argumento del manejador esta definida únicamente para los manejadores `q` y `qr`. Para esos manipuladores toma uno de los siguientes valores:

- `q`, indicando que la cadena viene de un contexto no interpolable como `'...'` o `q{...}` o `qw{...}`,
- `qq`, indicando que la cadena aparece en un contexto interpolable como `"..."` o `qq{...}` o `m/.../` o el primer argumento de una sustitución `s/.../.../`,
- `tr`, indicando que la cadena aparece en `tr/.../.../` o `y/.../.../`,
- `s`, indicando que la cadena es el segundo argumento de una sustitución `s/.../.../`.

Constante	Manejador	Argumentos
"doble comi"	q	('doble comi', 'doble comi', 'q')
qq{qq comi}	q	('qq comi', 'qq comi', 'qq')
'comi simples'	q	('comi simples', 'comi simples', 'q')
q{q comi}	q	('q comi', 'q comi', 'q')
qw{qw comi}	q	('qw comi', 'qw comi', 'q')
<<HERE docu HERE	q	("docu \n", "docu \n", 'qq')
<<'HERE' com docu HERE	q	("com docu \n", "com docu \n", 'qq')
tr/desde/hacia/	q q	('desde', 'desde', 'tr') ('hacia', 'hacia', 'tr')
qr{qr comi}	qr	('qr comi', 'qr comi', 'qq')
s/s_pat/s_text/	qr q	('s_pat', 's_pat', 'qq') ('s_text', 's_text', 's')
m/m patron/	qr	('m patron', 'm patron', 'qq')
12345	integer	('12345', 12345, undef)
12_345	integer	('12_345', 12345, undef)
12345.0	float	('12345.0', 12345.0, undef)
12345e1	float	('12345e1', 123450.0, undef)
012345	binary	('012345', 5349, undef)
0x12345	binary	('0x12345', 74565, undef)
0xBadDeed	binary	('0xBadDeed', 195941733, undef)

Cuadro 7.4: Invocación de los manejadores de constantes

El módulo `Number::Fraction` permite la sobrecarga de constantes. Véase un ejemplo:

```
$ cat fractions.pl
#!/usr/local/bin/perl5.8.0
use Number::Fraction ':constants';

my $trescuartos = '1/2'+'1/4';
print "$trescuartos\n";

my $x = '1hola'+2;
print "$x\n";

no Number::Fraction;
my $trescuartos = '1/2'+'1/4';
print "$trescuartos\n";
```

La salida de este program es:

```
$ ./fractions.pl
3/4
3
2
```

Esta sobrecarga se consigue definiendo `import` en la línea 22 del listado que sigue a continuación, para que llame a `overload::constant` sobre el hash `%_const_handlers`.

En este caso el hash tiene una única clave, ya que sólo estamos interesados en tratar las constantes de tipo cadena. Recuerde que el constructor `new` es el especificado en la sección 7.8.5, el cuál cuando recibe la cadena produce el objeto `Number::Fraction`.

En el caso de que el constructor no devuelva un objeto válido, devuelve el segundo argumento, que es el valor que Perl le da por defecto a la constante. El resultado es la conversión correcta de aquellas cadenas que pueden ser interpretadas como fracciones y que otras cadenas como `'1hola'` sean interpretadas de acuerdo a las tradiciones de Perl.

```
 1 package Number::Fraction;
 2
 3 use 5.006;
 4 use strict;
 5 use warnings;
 6 use Carp;
 7
 8 our $VERSION = sprintf "%d.%02d", '$Revision: 1.33 $ ' =~ /(\d+)\.(\d+)/;
 9
10 use overload
11   q("") => 'to_string',
12   '0+' => 'to_num',
13   '+' => 'add',
14   '*' => 'mult',
15   '-' => 'subtract',
16   '/' => 'div',
17   fallback => 1;
18
19 my %_const_handlers =
20   (q => sub { return __PACKAGE__->new($_[0]) || $_[1] });
21
22 sub import {
```

```
23   overload::constant %_const_handlers if $_[1] and $_[1] eq ':constants';
24 }
25
26 sub unimport {
27   overload::remove_constant(q => undef);
28 }
29
30 sub new {
31  ...
32 }
```

Nótese la condición para la importación en la línea 23: `if $_[1] and $_[1] eq ':constants'`. Esto significa que para activar el manejador de constantes de tipo cadena el programa cliente ha de declarar

```
use Number::Fraction ':constants';
```

esto es una buena idea: cambiar el modo en que funcionan las constantes es un cambio lo suficientemente trascendental como para sólo hacerlo bajo petición explícita del usuario.

Recuerde que la función `unimport` se ejecutará cuando el usuario haga una llamada en su programa a `no Number::Fraction`. En este caso la negación produce el abandono del manejo de las constantes.

7.8.7. Constructor de copia

La sobrecarga de operadores da lugar a confusión en ocasiones, supongamos por ejemplo que `$d` y `$t` son dos `Math::BigFloat`. Una asignación como:

```
$v = $d/$t;
```

trata a `$d` y `$t` como valores en vez de como las referencias que son a los correspondientes objetos. Consideremos una asignación como:

```
$v1 = $v;
```

La asignación hace que `$v1` y `$v` sean una referencia al mismo objeto. Si posteriormente incrementamos `$v1`:

```
$v1++;
```

tendremos un efecto lateral, ya que `$v` se ve incrementado. Este efecto es, cuando menos, inconsistente con el uso normal de las operadores sobrecargados, en el sentido de que estos "trabajan" los referentes (objetos) y la asignación "trabaja" con las referencias.

Los operadores que cambian el valor de sus operandos se llaman "mutantes". El módulo `overload` proporciona un medio para interceptar a los mutantes y clonar los objetos que están siendo mutados de manera que la mutación sólo afecte a la copia del objeto. Esta intercepción se hace sobrecargando la operador engañosamente denominado `=`

```
package Math::BigFloat;

sub copy { Math::BigFloat->new(${$_[0]}) }

use overload
      ...
      '++' => "incr",
      '='  => "copy";
```

Esta sobrecarga no cambia la conducta del operador de asignación. La rutina `copy` es llamada antes que la subrutina asociada con cualqueir mutante. Asi pues, el código:

```
$v1++;
```

Se traduce por:

```
$v1 = $v1->copy(undef,"");
$v1->incr(undef,"");
```

7.8.8. Ejercicio: Sobrecarga de Operadores

Considere el siguiente programa:

```
$ cat -n ./fractions2.pl
1  #!/usr/local/bin/perl5.8.0  -w
2  use Number::Fraction ':constants';
3
4  my $t = '1/2';
5  my $x = $t;
6  $t += '1/2';
7  print "t=$t, ref(t)=",ref($t),"\n";
8  print "x=$x, ref(x)=",ref($x),"\n";
```

Al ejecutarlo, produce la siguiente salida:

```
$ ./fractions2.pl
t=1/1, ref(t)=Number::Fraction
x=1/2, ref(x)=Number::Fraction
```

¿Porqué la asignación de la línea 5 (`$x = $t`) no parece comportarse como una asignación de referencias como se señaló en la sección anterior?. La modificación de `$t` no produce un efecto lateral sobre `$x`. Nótese que el operador `=` no ha sido sobrecargado en `Number::Fraction`. Puede comprobarlo en el listado de la cabecera de `Number::Fraction` que aparece en la sección 7.8.6.

Para ayudar a entender lo que esta ocurriendo, aqui tiene el código del método `add` que sobrecarga el operador `+`

```
sub add {
  my ($l, $r, $rev) = @_;

  if (ref $r) {
    if (UNIVERSAL::isa($r, ref $l)) {
      return (ref $l)->new($l->{num} * $r->{den} + $r->{num} * $l->{den},
   $r->{den} * $l->{den});
    } else {
      croak "Can't add a ", ref $l, " to a ", ref $l;
    }
  } else {
    if ($r =~ /^[-+]?\d+$/) {
      return $l + (ref $l)->new($r, 1);
    } else {
      return $l->to_num + $r;
    }
  }
}
```

Capítulo 8

CGI

8.1. Introducción a CGI (Common Gateway Inteface)

CGI (Commnon Gateway Interface) permite ejecutar programas en un servidor web.

CGI proporciona un canal consistente e independiente del lenguaje de programación, para el acceso remoto a bases de datos o aplicaciones.

CGI es el medio que tiene un servidor que habla HTTP para comunicarse con un programa.

La idea es que cada cliente y programa servidor (independientemente del sistema operativo) tengan los mismo mecanismos para intercambiar datos, y que esos mecanismos los implemente de alguna forma el programa que hace de puente.

Un puente es un programa que actúa como intermediario entre el servidor HTTP y otro programa que puede ser ejecutado por medio de la línea de comandos (por ejemplo, una base de datos relacional).

Una transacción con CGI realiza los siguientes pasos:

1. El cliente envía al servidor una petición respetando el formato estándar de una URL (este incluye el tipo de servicio y la ubicación del mismo). Además se envia un encabezado con los datos del cliente.

2. El servidor procesa la petición cuando llega y dependiendo del tipo de servicio solicitado decide que hacer luego. Si se solicita una página HTML, la devuelve. Si se solicita un programa CGI:

   - Envia al programa puente, si existe, el encabezado de datos del cliente. Estos datos son pasados al programa como variables de entorno (`$ENV{}` en Perl).
   - Los parámetros de ejecución del programa, si existen, son tomados por el programa. Estos datos se pueden pasar por medio de variables de entorno (`$ENV{QUERY_STRING}`) o por la entrada estándar (`< STDIN>`). La forma en la que el cliente debe enviar los datos la decide el programador del puente.

3. El programa CGI devuelve una respuesta, como un documento HTML al servidor. El programa CGI siempre debe devolver una respuesta. Los datos deben estar precedidos por un encabezado que respete las convenciones MIME, indicando el tipo de datos devuelto en la cabecera.

4. La salida es devuelta al cliente por el servidor y se corta la comunicación.

8.1.1. Permisos y Configuración del Servidor

Para poder ejecutar CGIs, el primer paso es que el administrador del servidor web deberá darte los permisos apropiados. Los pasos necesarios para poder dar permisos de ejecución a usuarios conllevan la modificación de dos ficheros de configuración. Los ficheros implicados en apache son `srm.conf` y `access.conf`. La directiva a utilizar en `srm.conf` es `ScriptAlias` y en `access.conf` deberás escribir:

```
<Directory "/home/alu[0-9]*/public_html/cgi-bin">
AllowOverride None
```

```
Options ExecCGI
AddHandler cgi-script .cgi .pl
</Directory>
```

También debes recordar ponerle permisos de lectura y ejecución (`chmod a+rx script`) a tu script.

8.1.2. Hola Mundo

```
#!/usr/bin/perl -w
print "Content-type: text/html \n\n";
print "<!--Esta pagina se ha generado de forma dinamica -->\n";
print "<html>\n";
print "<head>";
print "<title>\"Hola Mundo\"</title>\n";
print "</head>";
print "<body>";
print "<h1>Hola mundo. Este es mi primer CGI en Perl </h1> \n";
print "</body>";
print "</html>\n";
```

8.2. Usando CGI.pm

8.2.1. Look ma, no hands

```
#!/usr/bin/perl
use CGI qw/:standard/;
print header(),
     start_html(-title=>'Wow!'),
     h1('Wow!'),
     'Look Ma, no hands!',
     end_html();
```

8.2.2. Un reloj virtual

```
#!/usr/bin/perl -w
use CGI ':standard';

$current_time = localtime;

print header,
      start_html('A virtual Clock'),
      h1('A virtual clock'),
      img({-src=>'http://nereida.deioc.ull.es/images/ball-green.gif', -alt=>'CLOCK'}),
      "The current time is $current_time.",
      hr,
      a({-href=>'/'},"Go to the home page."),
      end_html;
```

8.2.3. La propiedad distributiva

```
#!/usr/bin/perl -w
use CGI qw/:standard :html3/;
print header,
      start_html('Vegetables'),
      h1('Eat your vegetables');
```

```
print table({-border=>undef},
        caption(strong('When Should You Eat Your Vegetables?')),
        Tr({-align=>CENTER,-valign=>TOP},
          [
           th(['','Breakfast','Lunch','Dinner']),
           th('Tomatoes').td(['no','yes','yes']),
           th('Broccoli').td(['no','no','yes']),
           th('Onions').td(['yes','yes','yes'])
          ]
        )
      );
print hr,
      end_html;
```

Otra forma de hacer una tabla:

```
#!/usr/bin/perl -w
use CGI qw/:standard :html3/;
print header,
      start_html('Vegetables'),
      h1('Eat your vegetables');

     @values = (1..5);

     @headings = ('N','N'.sup('2'),'N'.sup('3'));
     @rows = th(\@headings);
     foreach $n (@values) {
        push(@rows,td([$n,$n**2,$n**3]));
     }
     print table({-border=>undef,-width=>'25%'},
                 caption(b('Wow.  I can multiply!')),
                 Tr(\@rows)
                );
print hr,
      end_html;
```

8.2.4. Uso de un formulario

```
#!/usr/bin/perl -w
use CGI qw(:standard);

print header;
print start_html('A Simple Example'),
    h1('A Simple Example'),
    start_form,
    "What's your name? ",textfield('name'),
    p,
    "What's the combination?",
    p,
    checkbox_group(-name=>'words',
                   -values=>['eenie','meenie','minie','moe'],
                   -defaults=>['eenie','minie']),
    p,
```

```
    "What's your favorite color? ",
    popup_menu(-name=>'color',
               -values=>['red','green','blue','chartreuse']),
    p,
    submit,
    end_form,
    hr;

if (param()) {
    print
        "Your name is",em(param('name')),
        p,
        "The keywords are: ",em(join(", ",param('words'))),
        p,
        "Your favorite color is ",em(param('color')),
        hr;
}
print end_html;
```

8.3. Depuración

Si estas ejecutando el script desde la línea de comandos puedes pasarle una lista de keywords o bien parejas `parameter=value` ya sea como argumentos o bien desde standard input. Puedes pasar keywords asi:

```
my_script.pl keyword1 keyword2 keyword3
```

o bien:

```
my_script.pl keyword1+keyword2+keyword3
```

o bien:

```
my_script.pl name1=value1 name2=value2
```

o bien:

```
my_script.pl name1=value1&name2=value2
```

o incluso enviando parámetros delimitados por caracteres newline:

```
% my_script.pl
first_name=fred
last_name=flintstone
occupation='granite miner'
^D
```

Cuando depures, puedes usar comillas y el caracter backslash para introducir secuencias de escape y caracteres especiales. Esto te permite hacer cosas como:

```
my_script.pl 'name 1=I am a long value' name\ 2=two\ words
```

Si ejecutas un script que hace uso de CGI.pm desde la línea de comandos sin proporcionar ningún argumento, te aparecerá la línea:

```
(offline mode: enter name=value pairs on standard input)
```

y se queda esperando para que escribas los parámetros. Termina presionando `^D` (`^Z` en NT/DOS). Si no quieres dar los parámetros a CGI.pm, presiona `^D`.

Puedes cambiar la conducta por defecto de uno de los siguientes métodos:

- Llamar al script con un número de parámetros vacío Ejemplo:

  ```
  my_script.pl ''
  ```

- Redirigir standard input desde `/dev/null` o desde un fichero vacío. Ejemplo:

  ```
  my_script.pl </dev/null
  ```

- Incluir `-no_debug` en la lista de símbolos a importar en lalínea `use`. Ejemplo:

  ```
  use CGI qw/:standard -no_debug/;
  ```

```
#!/usr/bin/perl
use CGI;
use CGI::Carp qw(fatalsToBrowser carpout);

open (LOG,">>/home/fred/logs/search_errors.log") ||
      die "couldn't open log file: $!";
carpout(LOG);
```

8.4. Usando Formas

8.4.1. Un reloj configurable

```
#!/usr/bin/perl
#script: time4.pl

use CGI ':standard';
use POSIX 'strftime';

# print the HTTP header and the HTML document
print header,
    start_html('A Virtual Clock'),
    h1('A Virtual Clock');
print_time();
print_form();
print end_html;

# print out the time
sub print_time {
    my($format);
    if (param) {
  $format = (param('type') eq '12-hour') ? '%r ' : '%T ' if param('time');
  $format .= '%d ' if param('day');
  $format .= '%B ' if param('month');
  $format .= '%A ' if param('day-of-month');
  $format .= '%Y ' if param('year');
```

```
    } else {
  $format = '%r %A %B %d %Y';
    }
    $current_time = strftime($format,localtime);
    print "The current time is ",strong($current_time),".",hr;
}

# print the clock settings form
sub print_form {
    print start_form,
    "Show: ",
    checkbox(-name=>'time',-checked=>1),
    checkbox(-name=>'day',-checked=>1),
    checkbox(-name=>'month',-checked=>1),
    checkbox(-name=>'day-of-month',-checked=>1),
    checkbox(-name=>'year',-checked=>1),
    p(),
    "Time style: ",
    radio_group(-name=>'type',
                -values=>['12-hour','24-hour']),
    p(),
    reset(-name=>'Reset'),
    submit(-name=>'Set'),
    end_form;
}
```

8.4.2. Elegir la página de salida

```
!/usr/bin/perl
use CGI qw/:standard/;
@advice = (
     'A stitch in time saves nine.',
     'Look both ways before crossing the street.',
     'Chew completely before swallowing.',
     'A penny saved is a penny earned.',
     'Fools rush in where angels fear to tread.'
     );
print header,
    start_html("Good Advice"),
    h1("Good Advice");
$action = param('action');

if ($action=~/Message (\d+)/) {
    $message_no = $1-1;
    print strong($advice[$message_no]);
}
print start_form;
foreach (1..5) {
    print submit(-name=>'action',-value=>"Message $_");
}

print end_form,end_html;
```

8.4.3. Usando hidden para formularios multipágina

```
#!/usr/bin/perl

# script: loan.cgi
use CGI qw/:standard :html3/;

# this defines the contents of the fill out forms
# on each page.
@PAGES = ('Personal Information','References','Assets','Review','Confirmation');
%FIELDS = ('Personal Information' => ['Name','Address','Telephone','Fax'],
     'References'           => ['Personal Reference 1','Personal Reference 2'],
     'Assets'               => ['Savings Account','Home','Car']
     );
# accumulate the field names into %ALL_FIELDS;
foreach (values %FIELDS) {
    grep($ALL_FIELDS{$_}++,@$_);
}


# figure out what page we're on and where we're heading.
$current_page = calculate_page(param('page'),param('go'));
$page_name = $PAGES[$current_page];

print_header();
print_form($current_page)         if $FIELDS{$page_name};
print_review($current_page)       if $page_name eq 'Review';
print_confirmation($current_page) if $page_name eq 'Confirmation';
print end_html;

# CALCULATE THE CURRENT PAGE
sub calculate_page {
    my ($prev,$dir) = @_;
    return 0 if $prev eq '';  # start with first page
    return $prev + 1 if $dir eq 'Submit Application';
    return $prev + 1 if $dir eq 'Next Page';
    return $prev - 1 if $dir eq 'Previous Page';
}

# PRINT HTTP AND HTML HEADERS
sub print_header {
    print header,
    start_html("Your Friendly Family Loan Center"),
    h1("Your Friendly Family Loan Center"),
    h2($page_name);
}

# PRINT ONE OF THE QUESTIONNAIRE PAGES
sub print_form {
    my $current_page = shift;
    print "Please fill out the form completely and accurately.",
       start_form,
       hr;
    draw_form(@{$FIELDS{$page_name}});
```

```
    print hr;
    print submit(-name=>'go',-value=>'Previous Page')
  if $current_page > 0;
    print submit(-name=>'go',-value=>'Next Page'),
       hidden(-name=>'page',-value=>$current_page,-override=>1),
       end_form;
}

# PRINT THE REVIEW PAGE
sub print_review {
    my $current_page = shift;
    print "Please review this information carefully before submitting it. ",
       start_form;
    my (@rows);
    foreach $page ('Personal Information','References','Assets') {
  push(@rows,th({-align=>LEFT},em($page)));
  foreach $field (@{$FIELDS{$page}}) {
      push(@rows,
     TR(th({-align=>LEFT},$field),
        td(param($field)))
     );
      print hidden(-name=>$field);
  }
    }
    print table({-border=>1},caption($page),@rows),
       hidden(-name=>'page',-value=>$current_page,-override=>1),
       submit(-name=>'go',-value=>'Previous Page'),
       submit(-name=>'go',-value=>'Submit Application'),
       end_form;
}

# PRINT THE CONFIRMATION PAGE
sub print_confirmation {
    print "Thank you. A loan officer will be contacting you shortly.",
       p,
       a({-href=>'../source.html'},'Code examples');
}


# CREATE A GENERIC QUESTIONNAIRE
sub draw_form {
    my (@fields) = @_;
    my (%fields);
    grep ($fields{$_}++,@fields);
    my (@hidden_fields) = grep(!$fields{$_},keys %ALL_FIELDS);
    my (@rows);
    foreach (@fields) {
  push(@rows,
       TR(th({-align=>LEFT},$_),
    td(textfield(-name=>$_,-size=>50))
    )
       );
    }
```

```
    print table(@rows);

    foreach (@hidden_fields) {
        print hidden(-name=>$_);
    }
}
```

8.5. Acción: enviar por correo

```
#!/usr/bin/perl -w

$sendmail = '/usr/sbin/sendmail';
$sender = '...';
$recipient = '...';
$site_name = 'LHP';
$site_url = '...';
$filename = 'alumnos_LHP_2003-2004.txt';

use CGI;

$query = new CGI;

foreach $field (sort ($query->param)) {
foreach $value ($query->param($field)) {
$mail_body .= "$field: $value \n";
}
}

if (($email = $query->param('07_email')) and
($query->param('07_email') =~ /@/)) {

if ($name = $query->param('01_name')) {
$name =~ s/"//g;
$sender = "\"$name\" <$email>";
} else {
$sender = "$email";
}
}

#=======================
#send the email message
open(MAIL, "|$sendmail -oi -t") or die "Can't open pipe to $sendmail: $!\n";
print MAIL "To: $recipient\n";
print MAIL "From: $sender\n";
print MAIL "Subject: Ficha alumno LHP\n\n";
print MAIL "$mail_body";
close(MAIL) or die "Can't close pipe to $sendmail: $!\n";

print "Content-type: text/html\n\n";
print <<"EOF";
<HTML>
<HEAD>
<TITLE>Gracias!</TITLE>
```

```
</HEAD>
<BODY LINK="#0000ff" VLINK="#800080"
 BACKGROUND="http://nereida.deioc.ull.es/images/BACKORCHESTRA.JPG">
<H1><Gracias!</H1>
      <B>
      LHP
      </B>
      </P>

<P><img src="http://nereida.deioc.ull.es/images/birds1.gif"></P>
<P>Volver a <A HREF="$site_url" target=_top>$site_name</A>.</P>
</BODY>
</HTML>
EOF
```

8.5.1. Salvando resultados: concurrencia

```
#!/usr/bin/perl
# guestbook.pl

use CGI qw/:standard :html3 :netscape/;
use POSIX;

@REQUIRED = qw/name e-mail/;
@OPTIONAL = qw/location comments/;
$TIMEOUT = 10;  # allow up to 10 seconds for waiting on a locked guestbook
$GUESTBOOKFILE = "./guestbookfile.txt";
%ENTITIES = ('&'=>'&amp;', '>'=>'&gt;', '<'=>'&lt;', '\"'=>'&quot;' );

print header,
    start_html('Guestbook'),
    h1("Guestbook");

$_ = param('action');

 CASE: {
     /^sign/i and do    { sign_guestbook(); last CASE; };
     /^confirm/i and do { write_guestbook() and view_guestbook(); last CASE; };
     /^view/i and do    { view_guestbook(); last CASE; };
     # default
     generate_form();
 }

sub sign_guestbook {
    my @missing = check_missing(param());
    if (@missing) {
        print_warning(@missing);
        generate_form();
        return undef;
    }
    my @rows;
    foreach (@REQUIRED,@OPTIONAL) {
        push(@rows,TR(th({-align=>LEFT},$_),td(escapeHTML(param($_)))));
    }
```

```
    print "Here is your guestbook entry.  Press ",
          em('Confirm')," to save it, or ",em('Change'),
          " to change it.",
          hr,
          table(@rows),
          hr;

    print start_form;
    foreach (@REQUIRED,@OPTIONAL) {
        print hidden(-name=>$_);
    }
    print submit(-name=>'action',
                 -value=>'Change Entry'),
          submit(-name=>'action',
                 -value=>'Confirm Entry'),
          end_form;
}

print end_html;

sub check_missing {
    my (%p);
    grep (param($_) ne '' && $p{$_}++,@_);
    return grep(!$p{$_},@REQUIRED);
}

sub print_warning {
    print font({-color=>'red'},
          'Please fill in the following fields: ',
          em(join(', ',@_)),
          '.');
}

sub generate_form {
    print start_form,
     table(
        TR({-align=>LEFT},
          th('Your name'),
          td(textfield(-name=>'name',-size=>50))
        ),
        TR({-align=>LEFT},
          th('Your e-mail address'),
          td(textfield(-name=>'e-mail',-size=>50))
        ),
        TR({-align=>LEFT},
          th('Your location (optional)'),
          td(textfield(-name=>'location',-size=>50))
        ),
        TR({-align=>LEFT},
          th('Comments (optional)'),
          td(textarea(-name=>'comments',-rows=>4,
                      -columns=>50,
                      -wrap=>1))
```

```
        )
     ),
     br,
     submit(-name=>'action',-value=>'View Guestbook'),
     submit(-name=>'action',-value=>'Sign Guestbook'),
     end_form;
}

sub write_guestbook {
    my $fh = lock($GUESTBOOKFILE,1);
    unless ($fh) {
       print strong('Sorry, an error occurred: unable to open guestbook file.'),p();
       Delete('action');
       print a({-href=>self_url},'Try again');
       return undef;
    }
    my $date = strftime('%D',localtime);
    print $fh join("\t",$date,map {CGI::escape(param($_))} (@REQUIRED,@OPTIONAL)),"\n";
    print $fh "\n";
    print "Thank you, ",param('name'),", for signing the guestbook.\n",
      p(),
      a({href=>"../source.html"},'Code Examples');
    unlock($fh);
    1;
}

sub view_guestbook {

    print start_form,
          submit(-name=>'Sign Guestbook'),
          end_form
          unless param('name');

    my $fh = lock($GUESTBOOKFILE,0);

    my @rows;
    unless ($fh) {
       print strong('Sorry, an error occurred: unable to open guestbook file.'),br;
       Delete('action');
       print a({-href=>self_url},'Try again');
       return undef;
    }
    while (<$fh>) {
       chomp;
       my @data = map {CGI::unescape($_)} split("\t");
       foreach (@data) { $_ = escapeHTML($_); }
       unshift(@rows,td(\@data));
    }
    unshift(@rows,th(['Date',@REQUIRED,@OPTIONAL]));
    print table({-border=>''},
          caption(strong('Previous Guests')),
          TR(\@rows));
    print p,a({href=>"../source.html"},'Code Examples');
```

```
    1;
}

sub escapeHTML {
    my $text = shift;
    $text =~ s/([&\"><])/$ENTITIES{$1}/ge;
    return $text;
}

sub LOCK_SH { 1 }
sub LOCK_EX { 2 }
sub LOCK_UN { 8 }

sub lock {
    my $path = shift;
    my $for_writing = shift;

    my ($lock_type,$path_name,$description);
    if ($for_writing) {
        $lock_type = LOCK_EX;
        $path_name = ">>$path";
        $description = 'writing';
    } else {
        $lock_type = LOCK_SH;
        $path_name = $path;
        $description = 'reading';
    }

    local($msg,$oldsig);
    my $handler = sub { $msg='timed out'; $SIG{ALRM}=$oldsig; };
    ($oldsig,$SIG{ALRM}) = ($SIG{ALRM},$handler);
    alarm($TIMEOUT);

    open (FH,$path_name) or
       warn("Couldn't open $path for $description: $!"), return undef;

    # now try to lock it
    unless (flock (FH,$lock_type)) {
       warn("Couldn't get lock for $description (" . ($msg || "$!") . ")");
       alarm(0);
       close FH;
       return undef;
    }

    alarm(0);
    return FH;
}

sub unlock {
    my $fh = shift;
    flock($fh,LOCK_UN);
    close $fh;
}
```

8.6. Redirección

```
#!/usr/bin/perl
#redirect.pl

use CGI qw/:standard/;

print redirect('http://www.wiley.com/compbooks/stein/'),
    start_html("Moved"),
    h1("Document Moved"),
    "The document you have requested has moved ",
    a({href=>'http://www.wiley.com/compbooks/stein/'},"here"),
    '.';
```

8.7. Gráficos aleatorios

```
#!/usr/bin/perl
# script: random_pict2.pl

use CGI qw/:standard/;
$PICTURE_PATH = path_translated();
$PICTURE_URL = path_info();
chdir $PICTURE_PATH
    or die "Couldn't chdir to pictures directory: $!";
@pictures = <*.{jpg,gif}>;
$lucky_one = $pictures[rand(@pictures)];
die "Failed to pick a picture" unless $lucky_one;

print redirect("$PICTURE_URL/$lucky_one");
```

8.8. .htaccess

Ejemplo de fichero .htaccess

```
Authtype Basic
AuthName doctorado
AuthUserFile /home/doct3/etc/.passwd
Require valid-user
Satisfy All
```

Haga `man htpasswd` para ver como crear passwords

8.9. Quadraphobia

```
#!/usr/bin/perl

use CGI qw/:standard/;

print header,
    start_html('QuadraPhobia'),
    h1('QuadraPhobia'),
    start_form(),
    image_button(-name=>'square',
```

```
     -src=>'red_square.gif',
     -width=>200,
     -height=>200,
     -align=>MIDDLE),
    end_form();
if (param()) {
    ($x,$y) = (param('square.x'),param('square.y'));
    $pos = 'top-left' if $x < 100 && $y < 100;
    $pos = 'top-right' if $x >= 100 && $y < 100;
    $pos = 'bottom-left' if $x < 100 && $y >= 100;
    $pos = 'bottom-right' if $x >= 100 && $y >= 100;
    print b("You clicked on the $pos part of the square.");
}
print p,a({href=>'../source.html'},"Code examples");
print end_html();
```

8.10. Graffiti

```
#!/usr/bin/perl

# script: graffiti2.pl

# Keep a growing list of phrases from the user.

# CONSTANTS
$STATE_DIR = "./STATES";  # must be writable by 'nobody'

use CGI qw/:html/;
$q = new CGI;
$session_key = $q->path_info();
$session_key =~ s|^/||;       # get rid of the initial slash

# If no valid session key has been provided, then we
# generate one, tack it on to the end of our URL as
# additional path information, and redirect the user
# to this new location.
unless (valid($session_key)) {
    $session_key = generate_session_key();
    print $q->redirect($q->url() . "/$session_key");
    exit 0;
}

$old = fetch_old_state($session_key);

# Add the new item(s) to the old list of items
if ($q->param('action') eq 'ADD') {
    $q->param(-name=>'item',
              -value=>[$old->param('item'),$q->param('item')]);
} elsif ($action eq 'CLEAR') {
    $q->Delete('item');
}

# Save the new list to disk
```

```
save_state($session_key,$q);

# Now, at last, generate something for the use to look at.
print $q->header,
    $q->start_html("The growing list"),<<END;
<h1>The Growing List</h1>
Type a short phrase into the text field below.  Press <i>ADD</i>,
to append it to the history of the phrases that you've typed.  The
list is maintained on disk at the server end, so it won't get out of
order if you press the "back" button.  Press <i>CLEAR</i> to clear the
list and start fresh.  Bookmark this page to come back to the list later.
END
    ;
print $q->start_form,
    $q->textfield(-name=>'item',-default=>'',
      -size=>50,-override=>1),p(),
    $q->submit(-name=>'action',-value=>'CLEAR'),
    $q->submit(-name=>'action',-value=>'ADD'),
    $q->end_form,
    $q->hr,
    $q->h2('Current list');

if ($q->param('item')) {
    my @items = $q->param('item');
    print ol(li(\@items));
} else {
    print em('Empty');
}
print <<END;
<hr>
<a href="../../source.html">Code examples</a></address>
END
    ;
print $q->end_html;

# Silly technique: we generate a session key from a random number
# generator, and keep calling until we find a unique one.
sub generate_session_key {
    my $key;
    do {
  $key = int(rand(1000000));
    } until (! -e "$STATE_DIR/$key");
    return $key;
}

# make sure the session ID passed to us is a valid one by
# looking for a numeric-only string
sub valid {
    my $key = shift;
    return $key=~/^\d+$/;
}

# Open the existing file, if any, and read the current state from it.
```

```
# We use the CGI object here, because it's straightforward to do.
# We don't check for success of the open() call, because if there is
# no file yet, the new CGI(FILEHANDLE) call will return an empty
# parameter list, which is exactly what we want.
sub fetch_old_state {
    my $session_key = shift;
    open(SAVEDSTATE,"$STATE_DIR/$session_key") || return;
    my $cgi = new CGI(SAVEDSTATE);
    close SAVEDSTATE;
    return $cgi;
}

sub save_state {
    my($session_key,$q) = @_;
    open(SAVEDSTATE,">$STATE_DIR/$session_key") ||
  die "Failed opening session state file: $!";
    $q->save(SAVEDSTATE);
    close SAVEDSTATE;
}
```

8.11. cookies

```
#!/usr/bin/perl

# file: cookie.pl

use CGI qw(:standard :html3);

# Some constants to use in our form.
@colors=qw/aqua black blue fuschia gray green lime maroon navy olive
    purple red silver teal white yellow/;
@sizes=("<default>",1..7);

# recover the "preferences" cookie.
%preferences = cookie('preferences');

# If the user wants to change the background color or her
# name, they will appear among our CGI parameters.
foreach ('text','background','name','size') {
    $preferences{$_} = param($_) || $preferences{$_};
}

# Set some defaults
$preferences{'background'} = $preferences{'background'} || 'silver';
$preferences{'text'} = $preferences{'text'} || 'black';

# Refresh the cookie so that it doesn't expire.
$the_cookie = cookie(-name=>'preferences',
                     -value=>\%preferences,
                     -path=>'/',
                     -expires=>'+30d');
print header(-cookie=>$the_cookie);
```

```
# Adjust the title to incorporate the user's name, if provided.
$title = $preferences{'name'} ?
    "Welcome back, $preferences{name}!" : "Customizable Page";

# Create the HTML page.  We use several of the HTML 3.2
# extended tags to control the background color and the
# font size.  It's safe to use these features because
# cookies don't work anywhere else anyway.
print start_html(-title=>$title,
                 -bgcolor=>$preferences{'background'},
                 -text=>$preferences{'text'}
     );

print basefont({-size=>$preferences{size}}) if $preferences{'size'} > 0;

print h1($title),<<END;
You can change the appearance of this page by submitting
the fill-out form below.  If you return to this page any time
within 30 days, your preferences will be restored.
END
    ;

# Create the form
print hr,
    start_form,

    "Your first name: ",
    textfield(-name=>'name',
              -default=>$preferences{'name'},
              -size=>30),br,

    table(
    TR(
       td("Preferred"),
       td("Page color:"),
       td(popup_menu(-name=>'background',
         -values=>\@colors,
         -default=>$preferences{'background'})
    ),
       ),
    TR(
       td(''),
       td("Text color:"),
       td(popup_menu(-name=>'text',
         -values=>\@colors,
         -default=>$preferences{'text'})
    )
       ),
    TR(
       td(''),
       td("Font size:"),
       td(popup_menu(-name=>'size',
         -values=>\@sizes,
```

```
        -default=>$preferences{'size'})
    )
       )
    ),

    submit(-label=>'Set preferences'),
    end_form,
    hr;

print a({HREF=>"../source.html"},'Code examples');
```

8.12. Upload

```
!/usr/bin/perl
#file: upload.pl

use CGI qw/:standard/;

print header,
    start_html('file upload'),
    h1('file upload');
print_form()    unless param;
print_results() if param;
print end_html;

sub print_form {
    print start_multipart_form(),
       filefield(-name=>'upload',-size=>60),br,
       submit(-label=>'Upload File'),
       end_form;
}

sub print_results {
    my $length;
    my $file = param('upload');
    if (!$file) {
  print "No file uploaded.";
  return;
    }
    print h2('File name'),$file;
    print h2('File MIME type'),
    uploadInfo($file)->{'Content-Type'};
    while (<$file>) {
  $length += length($_);
    }
    print h2('File length'),$length;
}
!/usr/bin/perl
#file: upload.pl

use CGI qw/:standard/;

print header,
```

```
    start_html('file upload'),
    h1('file upload');
print_form()    unless param;
print_results() if param;
print end_html;

sub print_form {
    print start_multipart_form(),
       filefield(-name=>'upload',-size=>60),br,
       submit(-label=>'Upload File'),
       end_form;
}

sub print_results {
    my $length;
    my $file = param('upload');
    if (!$file) {
  print "No file uploaded.";
  return;
    }
    print h2('File name'),$file;
    print h2('File MIME type'),
    uploadInfo($file)->{'Content-Type'};
    while (<$file>) {
  $length += length($_);
    }
    print h2('File length'),$length;
}
```

8.13. Frames

```
#!/usr/bin/perl

#file: frames.pl

use CGI qw/:standard/;
print header,
start_html('Popup Window');

if (!param) {
    print h1("Ask your Question"),
    start_form(-target=>'answer'),
    "What's your name? ",textfield('name'),
    p(),
    "What's the combination?",
    checkbox_group(-name=>'words',
       -values=>['eenie','meenie','minie','moe'],
       -defaults=>['eenie','moe']),
    p(),
    "What's your favorite color? ",
    popup_menu(-name=>'color',
         -values=>['red','green','blue','chartreuse']),
    p(),
```

```
    submit,
    end_form;
} else {
    print h1("And the Answer is..."),
    "Your name is ",em(param(name)),
    p(),
    "The keywords are: ",
    em(join(", ",param('words'))),
    p(),
    "Your favorite color is ",em(param('color'));
}

print end_html;
```

8.14. JavaScript

```
#!/usr/bin/perl
# Script: javascript.pl

use CGI qw(:standard);

# Here's the javascript code.
$JSCRIPT=<<EOF;
// validate that the user is the right age.  Return
// false to prevent the form from being submitted.
function validateForm() {
   var today = new Date();
   var birthday = validateDate(document.form1.birthdate);
   if (birthday == 0) {
      document.form1.birthdate.focus()
      document.form1.birthdate.select();
      return false;
   }
   var milliseconds = today.getTime()-birthday;
   var years = milliseconds/(1000 * 60 * 60 * 24 * 365.25);
   if ((years > 20) || (years < 5)) {
      alert("You must be between the ages of 5 and 20 " +
            "to submit this form");
      document.form1.birthdate.focus();
      document.form1.birthdate.select();
      return false;
   }
   // Since we've calculated the age in years already,
   // we might as well send it up to our CGI script.
   document.form1.age.value=Math.floor(years);
   return true;
}

// make sure that the contents of the supplied
// field contain a valid date.
function validateDate(element) {
   var date = Date.parse(element.value);
   if (0 == date) {
```

```
      alert("Please enter date in format MMM DD, YY");
      element.focus();
      element.select();
   }
   return date;
}

// Compliments, compliments
function doPraise(element) {
   if (element.checked) {
      self.status=element.value +
                  " is an excellent choice!";
      return true;
   } else {
      return false;
   }
}

function checkColor(element) {
   var color = element.options[element.selectedIndex].text;
   if (color == "blonde") {
      if (confirm(
          "Is it true that blondes have more fun?"))
             alert("Darn.  That leaves me out.");
      } else
      alert(color + " is a fine choice!");
   }
EOF
;

# here's where the execution begins
print header;
print start_html(-title=>'Personal Profile',
                 -script=>$JSCRIPT);
print h1("Big Brother Wants to Know All About You");
print_prompt(),hr;
print_response() if param;
print end_html;

sub print_prompt {
   print start_form(-name=>'form1',
                    -onSubmit=>"return validateForm()"),
        "Birthdate (e.g. Jan 3, 1972): ",
        textfield(-name=>'birthdate',
                  -onBlur=>"validateDate(this)"),p(),
        "Sex: ",
        radio_group(-name=>'gender',
                    -value=>[qw/male female/],
                    -onClick=>"doPraise(this)"),p(),
        "Hair color: ",
        popup_menu(-name=>'color',
                   -value=>[qw/brunette blonde red gray/],
                   -default=>'red',
```

```
                   -onChange=>"checkColor(this)"),p(),
        hidden(-name=>'age',-value=>0),
        submit(),
        end_form;
}

sub print_response {
   import_names('Q');
   print h2("Your profile"),
         "You claim to be a ",b($Q::age),
        " year old ",b($Q::color,$Q::gender),".",
        "You should be ashamed of yourself for lying so ",
        "blatantly to big brother!",
        hr;
}
```

Capítulo 9

Mejora del Rendimiento

9.1. `B::Xref`

El módulo `B::Xref` permite construir un listado de referencias cruzadas: Por ejemplo, dado el programa:

```
$ cat -n variables.pl
 1  #!/usr/bin/perl
 2  #Crear una cadena de caracteres
 3  $variable="Soy una variable que contiene una cadena";
 4  print("El contenido de \$variable es: \t $variable\n");
 5  #Crear una un numero real
 6  $variable=3.141616;
 7  print("El contenido de \$variable es: \t $variable\n");
 8  #Crear una un numero entero
 9  $variable=55;
10  print("El contenido de \$variable es: \t $variable\n");
```

Podemos obtener un listado haciendo:

```
$ perl -MO=Xref variables.pl
File variables.pl
  Subroutine (definitions)
    Package UNIVERSAL
      &VERSION          s0
      &can              s0
      &isa              s0
    Package attributes
      &bootstrap        s0
  Subroutine (main)
    Package main
      $variable         3, 4, 6, 7, 9, 10
variables.pl syntax OK
```

Si queremos redirigir la salida a fichero, podemos usar la opción `-o`

```
$ perl -MO=Xref,-omatrixP.idx matrixP.pl
$ cat -n matrixP.idx
     1  File matrixP.pl
     2    Subroutine (definitions)
     3      Package Internals
     4        &HvREHASH         s0
     5        &SvREADONLY       s0
```

```
      &SvREFCNT         s0
      &hash_seed        s0
      &hv_clear_placeholders  s0
      &rehash_seed      s0
    Package PerlIO
      &get_layers       s0
    Package Regexp
      &DESTROY          s0
    Package UNIVERSAL
      &VERSION          s0
      &can              s0
      &isa              s0
    Package main
      &matrixProd       s30
      &printMat         s47
  Subroutine (main)
    Package main
      $matA             53, 55, 58
      $matB             54, 55, 60
      $matC             55, 62
      &matrixProd       &55
      &printMat         &58, &60, &62
  Subroutine matrixProd
    Package (lexical)
      $i                i5, 20
      $j                i6, 17, 20, 22, 22, 25, 25
      $k                i6, 19, 22, 22
      $matA             i4, 7
      $matB             i4, 9
      $nColsA           i8, 13, 23
      $nColsB           i10, 25
      $nRowsA           i7, 27
      $nRowsB           i9, 13
      $refMatC          i11, 20
      @$i               20, 22
      @$k               22, 23, 23
      @$matA            7, 8
      @$matB            9, 10
      @$refMatC         20, 22
      @@$i              22, 22
      @@$matA           8, 22
      @@$matB           10, 22
      @@$refMatC        22, 29
      @@@$i             22, 27, 27
      @@@$matA          22
      @@@$matB          22
    Package ?
      @??               8, 10
    Package main
      @_                4
  Subroutine printMat
    Package (lexical)
      $i                i36, 41
```

```
  $mat            i33, 34
  $nCols          i35, 42
  $nRows          i34, 45
  @$i             41, 45, 45
  @$mat           34, 35
  @@$mat          35, 41
  @@@$mat         41
Package ?
  @??             35
Package main
  $j              39, 41, 42, 42
  @_              33
```

9.2. Devel::Coverage

Veamos un ejemplo de uso:

```
perl -d:Coverage  /usr/local/bin/latex2html lhp
... salida de nuestr programa
$ ls -ltr | tail -2
drwxr-xr-x  2 lhp lhp   4096 2005-04-27 16:59 lhp
-rw-r--r--  1 lhp lhp 185967 2005-04-27 16:59 latex2html.cvp
$ coverperl latex2html.cvp | head -30
Total of 1 instrumentation runs.

/home/lhp/.latex2html-init
        7   main::bot_navigation_panel
                7    line 283
        0   main::do_cmd_originalTeX
                0    line 158
                0    line 159
                0    line 160
        0   main::do_cmd_spanishTeX
                0    line 151
                0    line 152
                0    line 153
        2   main::spanish_titles
                1    line 37
                1    line 40
                1    line 43
                1    line 46
                1    line 49
                1    line 52
                1    line 54
                1    line 57
                1    line 66
                1    line 69
                1    line 73
                1    line 76
                1    line 79
        0   main::spanish_today
                0    line 215
                0    line 216
```

9.3. Devel::Cover

El módulo `Devel::Cover` proporciona medidas de cubrimiento del código Perl.

```
$ perl -d:Cover=-coverage,statement matrixP.pl
Devel::Cover 0.53: Collecting coverage data for statement.
Selecting packages matching:
Ignoring packages matching:
    /Devel/Cover[./]
Ignoring packages in:
    .
    /etc/perl
    /usr/lib/perl/5.8.4
    /usr/lib/perl5
    /usr/local/lib/perl/5.8.4
    /usr/local/share/perl/5.8.4
    /usr/share/perl/5.8.4
    /usr/share/perl5
Matriz A
1       2       3
2       4       6
3       6       9

Matriz B
1       2
2       4
3       6

Matriz C
14      28
28      56
42      84
Devel::Cover: Writing coverage database to /home/lhp/projects/perl/src/cover_db/runs/1114597775.12157.23925
---------------------------------------------------------------- ------ ------
File                                                               stmt  total
---------------------------------------------------------------- ------ ------
matrixP.pl                                                         97.9   97.9
Total                                                              97.9   97.9
---------------------------------------------------------------- ------ ------
```

Esto genera un fichero de datos `cover_db`. A partir del mismo es posible generar un informe utilizando `cover`:

```
$ cover -report text
Reading database from /home/lhp/projects/perl/src/cover_db


---------------------------------------------------------------- ------ ------
File                                                               stmt  total
---------------------------------------------------------------- ------ ------
matrixP.pl                                                         97.9   97.9
Total                                                              97.9   97.9
---------------------------------------------------------------- ------ ------
```

```
Run:          matrixP.pl
Perl version: 5.8.4
OS:           linux
Start:        Wed Apr 27 10:29:35 2005
Finish:       Wed Apr 27 10:29:35 2005

matrixP.pl

line  err   stmt   code
1                  #!/usr/bin/perl -w
2
3                  sub matrixProd {
4              1     my ($matA, $matB) = @_;
5              1     my $i = 0;
6              1     my ($j, $k);
7              1     my $nRowsA = @{$matA};
               1
8              1     my $nColsA = @{$matA->[0]};
               1
9              1     my $nRowsB = @{$matB};
               1
10             1     my $nColsB = @{$matB->[0]};
               1
11             1     my $refMatC;
12
13             1     if ($nColsA != $nRowsB) {
14    ***      0       die ("Index of two matrices must be agree\n");
15                   }
16             1     while ($i < $nRowsA)  {
17             3       $j = 0;
18             3       while ($j < $nColsB) {
19             6         $k = 0;
20             6         $refMatC->[$i][$j] = 0;
21             6         while ($k < $nColsA) {
22            18           $refMatC->[$i][$j] += $matA->[$i][$k] * $matB->[$k][$j];
23            18           $k++;
24                       }
25             6         $j++;
26                     }
27             3       $i++;
28                   }
29             1     return $refMatC;
30                 }
31
32                 sub printMat {
33             3     my $mat = shift;
34             3     my $nRows = @{$mat};
               3
35             3     my $nCols = @{$mat->[0]};
               3
36             3     my $i = 0;
37
38             3     while ($i < $nRows) {
```

```
39            9      $j = 0;
40            9      while ($j < $nCols) {
41           21        print $mat->[$i][$j], "\t";
42           21       $j++;
43                   }
44            9      print "\n";
45            9      $i++;
46                 }
47               }
48
49               #
50               # ---- Main ----
51               #
52
53            1   $matA = [[1,2,3],[2,4,6],[3,6,9]];
54            1   $matB = [[1,2],[2,4],[3,6]];
55            1   $matC = matrixProd($matA,$matB);
56
57            1   print "Matriz A \n";
58            1   printMat($matA);
59            1   print "\nMatriz B \n";
60            1   printMat($matB);
61            1   print "\nMatriz C \n";
62            1   printMat($matC);
```

Ejercicio 9.3.1. *Escriba un comando/filtro que permita ver que líneas no se ejecutaron.*

Ejercicio 9.3.2. *Escriba un guión en el que las líneas salgan por orden de visitas.*

9.4. DProf

To profile a Perl script run the perl interpreter with the -d switch. So to profile script test.pl with Devel::DProf the following command should be used.

```
$ perl5 -d:DProf test.pl
```

Then run dprofpp to analyze the profile. The output of dprofpp depends on the flags to the program and the version of Perl you're using.

```
$ perl -d:DProf /usr/local/bin/latex2html perlexamples

$ ls -ltra
...................................................................
-rw-r--r--     1 lhp lhp   44459 2005-04-25 14:07 referencias.tex
drwxr-xr-x     7 lhp lhp    4096 2005-04-25 14:07 .
-rw-r--r--     1 lhp lhp 4354790 2005-04-25 14:08 tmon.out
drwxr-xr-x     2 lhp lhp   20480 2005-04-25 14:08 perlexamples
```

The *dprofpp* command interprets profile data produced by a profiler, such as the Devel::DProf profiler. Dprofpp will read the file tmon.out and will display the 15 subroutines which are using the most time. By default the times for each subroutine are given exclusive of the times of their child subroutines.

```
$ dprofpp
File::Glob::GLOB_QUOTE has 1 unstacked calls in outer
File::Glob::GLOB_TILDE has 1 unstacked calls in outer
Exporter::Heavy::heavy_export has 2 unstacked calls in outer
File::Glob::GLOB_ALPHASORT has 1 unstacked calls in outer
File::Glob::GLOB_BRACE has 1 unstacked calls in outer
Exporter::export has -2 unstacked calls in outer
File::Glob::GLOB_NOMAGIC has 1 unstacked calls in outer
File::Glob::AUTOLOAD has -5 unstacked calls in outer
Total Elapsed Time = 31.84545 Seconds
  User+System Time = 17.98545 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 9.88   1.777  5.582  13872   0.0001 0.0004  main::process_command
 8.71   1.567  6.438   9608   0.0002 0.0007  main::translate_environments
 7.66   1.377  1.377   1266   0.0011 0.0011  main::text_cleanup
 5.93   1.067  1.067  11290   0.0001 0.0001  main::iso_map
 4.16   0.748  5.708   9105   0.0001 0.0006  main::translate_commands
 3.58   0.643  0.848    387   0.0017 0.0022  main::add_real_child_links
 3.09   0.556  0.556  11998   0.0000 0.0000  NDBM_File::STORE
 2.91   0.524  0.576   4358   0.0001 0.0001  main::revert_to_raw_tex
 2.80   0.504  0.537  13872   0.0000 0.0000  main::convert_iso_latin_chars
 2.47   0.444  0.458    307   0.0014 0.0015  main::replace_verb_marks
 2.41   0.434  0.434  33015   0.0000 0.0000  NDBM_File::FETCH
 2.37   0.426  1.170    833   0.0005 0.0014  main::replace_sensitive_markers
 2.10   0.378  4.301   9105   0.0000 0.0005  main::real_replace_strange_accents
 2.07   0.373  1.424   6158   0.0001 0.0002  main::accent_safe_for_ij
 1.98   0.356  0.693   5725   0.0001 0.0001  main::mark_string
$
```

La salida:

```
%Time
    Percentage of time spent in this routine.

#Calls
    Number of calls to this routine.

sec/call
    Average number of seconds per call to this routine.

Name
    Name of routine.

CumulS
    Time (in seconds) spent in this routine and routines called from it.

ExclSec
    Time (in seconds) spent in this routine (not including those called
    from it).

Csec/c
    Average time (in seconds) spent in each call of this routine
    (including those called from it).
```

Opciones:

```
-a   Sort alphabetically by subroutine names.
-d   Reverse whatever sort is used
-R   Count anonymous subroutines defined in the same package separately.
-E   (default) Display all subroutine times exclusive of child
     subroutine times.
-I   Display all subroutine times inclusive of child subroutine times.
-l   Sort by number of calls to the subroutines. This may help identify
     candidates for inlining.
-O cnt
     Show only *cnt* subroutines. The default is 15.
-p script
     Tells dprofpp that it should profile the given script and then
     interpret its profile data. See -Q.
-Q   Used with -p to tell dprofpp to quit after profiling the script,
     without interpreting the data.
-q   Do not display column headers.
```

El siguiente ejemplo ordena la lista por número de llamadas:

```
$ dprofpp -lq
File::Glob::GLOB_QUOTE has 1 unstacked calls in outer
File::Glob::GLOB_TILDE has 1 unstacked calls in outer
Exporter::Heavy::heavy_export has 2 unstacked calls in outer
File::Glob::GLOB_ALPHASORT has 1 unstacked calls in outer
File::Glob::GLOB_BRACE has 1 unstacked calls in outer
Exporter::export has -2 unstacked calls in outer
File::Glob::GLOB_NOMAGIC has 1 unstacked calls in outer
File::Glob::AUTOLOAD has -5 unstacked calls in outer
 0.00       - -0.000    178        -      -  NDBM_File::FETCH
 0.00       - -0.000    161        -      -  main::normalize
 1.17   0.009  0.065    153   0.0001 0.0004  main::translate_environments
 0.00       - -0.000    145        -      -  main::balance_tags
 2.46   0.019  0.045    133   0.0001 0.0003  main::process_command
 2.46   0.019  0.019    133   0.0001 0.0001  main::convert_iso_latin_chars
 0.00       - -0.000    133        -      -  main::spanish_translation
 0.00       - -0.000    111        -      -  main::make_end_cmd_rx
 0.00       - -0.000     95        -      -  main::replace_cross_ref_marks
 0.00       - -0.000     89        -      -  Getopt::Long::ParseOptionSpec
 0.00       - -0.000     87        -      -  NDBM_File::STORE
 0.00       - -0.001     87        -      -  main::simplify
    -       -  0.008     81        - 0.0001  main::process_group_env
 1.30   0.010  0.010     80   0.0001 0.0001  main::iso_map
 0.00       - -0.000     78        -      -  main::escape_rx_chars
```

```
-r   Display elapsed real times rather than user+system times.
-s   Display system times rather than user+system times.
-T   Display subroutine call tree to stdout. Subroutine statistics are
     not displayed.
-t   Display subroutine call tree to stdout. Subroutine statistics are
     not displayed. When a function is called multiple consecutive times
     at the same calling level then it is displayed once with a repeat
     count.
```

Veamos un ejemplo, usando estas dos últimas opciones:

```
$ perl -d:DProf matrixP.pl
Matriz A
1       2       3
2       4       6
3       6       9

Matriz B
1       2
2       4
3       6

Matriz C
14      28
28      56
42      84
$ dprofpp
Total Elapsed Time =   -2e-05 Seconds
  User+System Time =  0.00998 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 0.00       - -0.000      1        -      -  main::matrixProd
 0.00       - -0.000      3        -      -  main::printMat
$ dprofpp -t
main::matrixProd
main::printMat (3x)
$ dprofpp -T
main::matrixProd
main::printMat
main::printMat
main::printMat
$ cat matrixP.pl
#!/usr/bin/perl -w

sub matrixProd {
  my ($matA, $matB) = @_;
  my $i = 0;
  my ($j, $k);
  my $nRowsA = @{$matA};
  my $nColsA = @{$matA->[0]};
  my $nRowsB = @{$matB};
  my $nColsB = @{$matB->[0]};
  my $refMatC;

  if ($nColsA != $nRowsB) {
    die ("Index of two matrices must be agree\n");
  }
  while ($i < $nRowsA)  {
    $j = 0;
    while ($j < $nColsB) {
      $k = 0;
      $refMatC->[$i][$j] = 0;
      while ($k < $nColsA) {
```

```
        $refMatC->[$i][$j] += $matA->[$i][$k] * $matB->[$k][$j];
        $k++;
      }
      $j++;
    }
    $i++;
  }
  return $refMatC;
}

sub printMat {
  my $mat = shift;
  my $nRows = @{$mat};
  my $nCols = @{$mat->[0]};
  my $i = 0;

  while ($i < $nRows) {
    $j = 0;
    while ($j < $nCols) {
      print $mat->[$i][$j], "\t";
     $j++;
    }
    print "\n";
    $i++;
  }
}

#
# ---- Main ----
#

$matA = [[1,2,3],[2,4,6],[3,6,9]];
$matB = [[1,2],[2,4],[3,6]];
$matC = matrixProd($matA,$matB);

print "Matriz A \n";
printMat($matA);
print "\nMatriz B \n";
printMat($matB);
print "\nMatriz C \n";
printMat($matC);
```

Otras opciones:

```
    -U   Do not sort. Display in the order found in the raw profile.
    -u   Display user times rather than user+system times.
    -V   Print dprofpp's version number and exit. If a raw profile is found
         then its XS_VERSION variable will be displayed, too.
    -v   Sort by average time spent in subroutines during each call. This
         may help identify candidates for inlining.
    -z   (default) Sort by amount of user+system time used. The first few
         lines should show you which subroutines are using the most time.
    -g "subroutine"
         Ignore subroutines except "subroutine" and whatever is called from
```

```
        it.
    -G <regexp>
        Aggregate "Group" all calls matching the pattern together. For
        example this can be used to group all calls of a set of packages

          -G "(package1::)|(package2::)|(package3::)"

        or to group subroutines by name:

          -G "getNum"

    -P   Used with -G to aggregate "Pull" together all calls that did not
         match -G.

    -f <regexp>
         Filter all calls matching the pattern.
```

Veamos otro ejemplo usando la opción `-G`. Hemos usado la traza de `latex2html`:

```
$ dprofpp -G "main" -O 30
File::Glob::GLOB_QUOTE has 1 unstacked calls in outer
File::Glob::GLOB_TILDE has 1 unstacked calls in outer
Exporter::Heavy::heavy_export has 2 unstacked calls in outer
File::Glob::GLOB_ALPHASORT has 1 unstacked calls in outer
File::Glob::GLOB_BRACE has 1 unstacked calls in outer
Exporter::export has -2 unstacked calls in outer
File::Glob::GLOB_NOMAGIC has 1 unstacked calls in outer
File::Glob::AUTOLOAD has -5 unstacked calls in outer
Option G Grouping: [main]
Grouping [main] Calls: [3796]
Grouping [main] Times: [42.4062000000002]
Grouping [main] IncTimes: [170.386800000001]
Total Elapsed Time =  1.05213 Seconds
  User+System Time = 0.772130 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 54.9   0.424  1.704   3796   0.0001 0.0004  main
 3.89   0.030  0.030      8   0.0037 0.0037  L2hos::Unix::BEGIN
 1.30   0.010  0.010      2   0.0050 0.0050  Exporter::as_heavy
 1.30   0.010  0.010      3   0.0033 0.0033  vars::BEGIN
 1.30   0.010  0.010     20   0.0005 0.0005  NDBM_File::TIEHASH
 0.00   0.000  0.000      2   0.0000 0.0000  Exporter::Heavy::heavy_export
 0.00   0.000  0.000      1   0.0000 0.0000  File::Glob::GLOB_BRACE
 0.00   0.000  0.000      1   0.0000 0.0000  File::Glob::GLOB_NOMAGIC
 0.00   0.000  0.000      1   0.0000 0.0000  File::Glob::GLOB_QUOTE
 0.00   0.000  0.000      1   0.0000 0.0000  File::Glob::GLOB_TILDE
 0.00   0.000  0.000      1   0.0000 0.0000  File::Glob::GLOB_ALPHASORT
 0.00       - -0.000      1        -      -  L2hos::Unix::Link
 0.00       - -0.000      1        -      -  L2hos::Unix::pathd
 0.00       - -0.000      1        -      -  L2hos::Unix::dd
 0.00       - -0.000      1        -      -  L2hos::Unix::home
 0.00       - -0.000      1        -      -  L2hos::Unix::fullname
 0.00       - -0.000      1        -      -  Getopt::Long::FindOption
 0.00       - -0.000      1        -      -  L2hos::Unix::syswait
```

```
 0.00       - -0.000      1        -      -  L2hos::Unix::path2latex
 0.00       - -0.000      1        -      -  warnings::BEGIN
 0.00       - -0.000      1        -      -  Getopt::Long::ConfigDefaults
 0.00       - -0.000      1        -      -  Getopt::Long::Configure
 0.00       - -0.000      1        -      -  Fcntl::BEGIN
 0.00       - -0.000      1        -      -  Fcntl::bootstrap
 0.00       - -0.000      1        -      -  AnyDBM_File::BEGIN
 0.00       - -0.000      1        -      -  NDBM_File::bootstrap
 0.00       - -0.000      1        -      -  warnings::unimport
 0.00       - -0.000      1        -      -  Cwd::bootstrap
 0.00       - -0.000      1        -      -  Config::TIEHASH
 0.00       - -0.000      1        -      -  Config::import

ENVIRONMENT
    The environment variable DPROFPP_OPTS can be set to a string containing
    options for dprofpp. You might use this if you prefer -I over -E or if
    you want -F on all the time.

    This was added fairly lazily, so there are some undesirable side
    effects. Options on the commandline should override options in
    DPROFPP_OPTS--but don't count on that in this version.
```

En este enlace encontrará un fichero latex para que pueda repetir el análisis con `Devel::DProf`: http://nereida.deioc.ull.es/ pp2/performancebook/proflatex2html.tgz

```
$perl -d:DProf /usr/local/bin/latex2html perlexamples
....
$ dprofpp
File::Glob::GLOB_QUOTE has 1 unstacked calls in outer
File::Glob::GLOB_TILDE has 1 unstacked calls in outer
Exporter::Heavy::heavy_export has 2 unstacked calls in outer
File::Glob::GLOB_ALPHASORT has 1 unstacked calls in outer
File::Glob::GLOB_BRACE has 1 unstacked calls in outer
Exporter::export has -2 unstacked calls in outer
File::Glob::GLOB_NOMAGIC has 1 unstacked calls in outer
File::Glob::AUTOLOAD has -5 unstacked calls in outer
Total Elapsed Time = 34.27311 Seconds
  User+System Time = 19.94311 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 10.1   2.015  7.150   9734   0.0002 0.0007  main::translate_environments
 8.69   1.733  6.484  14121   0.0001 0.0005  main::process_command
 6.88   1.373  1.373  11469   0.0001 0.0001  main::iso_map
 5.46   1.088  1.088   1270   0.0009 0.0009  main::text_cleanup
 4.70   0.937  6.624   9286   0.0001 0.0007  main::translate_commands
 3.08   0.615  0.968    389   0.0016 0.0025  main::add_real_child_links
 3.01   0.600  0.600  33355   0.0000 0.0000  NDBM_File::FETCH
 2.67   0.532  1.994   6286   0.0001 0.0003  main::accent_safe_for_ij
 2.58   0.515  0.561    311   0.0017 0.0018  main::replace_verb_marks
 2.51   0.500  0.567   4362   0.0001 0.0001  main::revert_to_raw_tex
 2.47   0.492  0.492  12168   0.0000 0.0000  NDBM_File::STORE
 2.35   0.468  0.526  14121   0.0000 0.0000  main::convert_iso_latin_chars
 2.08   0.414 19.990      1   0.4142 19.990  main::translate
 2.06   0.411  0.482   6276   0.0001 0.0001  main::get_next_pair_or_char_pr
 2.04   0.407  5.046      1   0.4071 5.0464  main::post_process
```

9.5. `Devel::SmallProf`

```
$perl -d:SmallProf /usr/local/bin/latex2html lhp
$ ls -ltr | tail -2
-rw-r--r--  1 lhp lhp 1542213 2005-04-27 11:54 smallprof.out
drwxr-xr-x  2 lhp lhp    4096 2005-04-27 11:54 lhp
```

O bien:

```
SMALLPROF_CONFIG=zg perl -d:SmallProf /usr/local/bin/latex2html lhp
$ cat -n smallprof.out | head -4
1  * file name : line number : line count : time (ms) : ctime (ms) : line source
2  /usr/local/bin/latex2html:1797:1:180:169:       waitpid($pid,0);
3  /usr/local/lib/latex2html/L2hos/Unix.pm:268:1:76:79:     $status = waitpid($child_pid, 0);
4  /usr/local/bin/latex2html:15591:1:32:30:            require $_ || die "\n*** Could not load $_ ***\n"
```

Ejercicio 9.5.1. *Escriba el comando que muestra las líneas de* `smallprof.out` *ordenadas por el campo* `count`.

9.6. Hilos en Perl: ithreads

Para ver si nuestro intérprete ha sido compilado con `ithreads`:

```
$ perl -MConfig -e 'print "$Config{useithreads}\n"'
define
```

El área bajo la curva $y = \frac{1}{1+x^2}$ entre 0 y 1 nos proporciona un método para calcular π:

$$\int_0^1 \frac{4}{(1+x^2)}dx = 4\arctan(x)|_0^1 \ = 4(\frac{\pi}{4} - 0) = \pi$$

```
$ cat -n pi.pl
     1  #!/usr/bin/perl -w
     2  # el número Pi es 3.
     3  # 1415926535 8979323846 2643383279 5028841971 6939937510 5820974944 5923078164
     4  # 0628620899 8628034825 3421170679 8214808651 3282306647 0938446095 5058223172
     5  # 5359408128 4811174502 8410270193 8521105559 6446229489 5493038196 4428810975
     6  # 6659334461 2847564823 3786783165 2712019091 4564856692 3460348610 4543266482
     7  # 1339360726 0249141273 7245870066 0631558817 4881520920 9628292540 9171536436
     8
     9  use strict;
    10  use threads;
    11  use threads::shared;
    12
    13  our $N;
    14  our $numthreads;
    15  our $pi = 0;
    16
    17  sub chunk {
    18    my $N = shift;
    19    my $numthreads = shift;
    20
    21    my ($i, $x, $sum, $w);
    22    my $id = threads->self()->tid();
    23    $w = 1/$N;
```

```
  $sum = 0;
  for ($i = $id; $i < $N; $i += $numthreads) {
    $x = ($i + 0.5)*$w;
    $sum += 4.0/(1.0 + $x*$x);
  }
  print "thread $id: $sum\n";
  return $sum;
}

sub par {
  my $nt = shift();
  my $task = shift;
  my @t; # array of tasks
  my $i;
  my @results;

  for($i=1; $i < $nt; $i++) {
    $t[$i] = threads->new($task, @_);
  }
  @{$results[0]} = $task->( @_);
  for($i=1; $i < $nt; $i++) {
    @{$results[$i]} = $t[$i]->join();
  }
  @results;
}

### main ###
$numthreads = (shift || 2);
$N = (shift || 10000);

my @results = par($numthreads, \&chunk, $N, $numthreads);
for (@results) { $pi += $_->[0] }
$pi /= $N;

print "PI = $pi\n";
for (@results) {
  print @$_," ";
}
print "\n";
```

Ejecución:

```
$ time ./pi.pl 1 1000000
thread 0: 3141592.65358976
PI = 3.14159265358976
3141592.65358976

real    0m0.976s
user    0m0.960s
sys     0m0.010s
$ time ./pi.pl 2 1000000
thread 1: 1570795.82679499
thread 0: 1570796.82679495
PI = 3.14159265358994
```

```
1570796.82679495 1570795.82679499

real    0m0.584s
user    0m1.080s
sys     0m0.000s
```

La máquina:

```
$ perl -ne 'print if m/processor|model_name|cache|MHz/' /proc/cpuinfo
processor       : 0
cpu MHz         : 2657.850
cache size      : 512 KB
processor       : 1
cpu MHz         : 2657.850
cache size      : 512 KB
processor       : 2
cpu MHz         : 2657.850
cache size      : 512 KB
processor       : 3
cpu MHz         : 2657.850
cache size      : 512 KB
```

Realmente en la máquina usada sólo hay dos procesadores, cada uno de los cuales virtualiza dos procesadores.

Ejercicio 9.6.1. *Reescriba el ejemplo anterior usando una planificación por bloques. ¿Son mejores los tiempos obtenidos? ¿Mejora la precisión en el cálculo de pi?*

Ejercicio 9.6.2. *Las asignaciones cíclica y por bloques son un caso particular de la asignación cíclica con tamaño de bloque de ciclo fijo. Escriba una subrutina* `parfor` *que implante este tipo de asignación.*

Ejercicio 9.6.3. *En una política de planificación dinámica los procesadores ejecutan los trozos o bloques de iteraciones de tamaño especificado b dinámicamente en exclusión mutua. El iterador se convierte en un recurso compartido. Cada hilo intenta ganar acceso al iterador y si lo consigue obtiene b iteraciones consecutivas que ejecuta. ¿Cuando puede resultar ventajosa una planificación dinámica? Escriba una rutina genérica que provea planificación dinámica de un bucle.*

Vea también `perldoc perlthrtut` o bien http://search.cpan.org/ñwclark/perl-5.8.6/pod/perlthrtut.pod

9.7. Tuberías y Pipes

9.7.1. El Cálculo de los Números Primos

Secuencial:

```
$ cat -n primes-without-threads.pl
#!/usr/bin/perl -w
use strict;

my @primes = (2);

NEW_NUMBER:
for my $num (3 .. 1000) {
        foreach (@primes) { next NEW_NUMBER if $num % $_ == 0 }
        print "Found prime $num\n";
        push @primes, $num;
}
```

Paralelo:

```
$ cat -n primes-using-threads.pl
#!/usr/bin/perl -w

use strict;

use threads;
use Thread::Queue;

my $stream = new Thread::Queue;
my $kid    = new threads(\&check_num, $stream, 2);

for my $i (3 .. 1000) {
        $stream->enqueue($i);
}

$stream->enqueue(undef);
$kid->join;

sub check_num {
        my ($upstream, $cur_prime) = @_;
        my $kid = undef;
        my $downstream = new Thread::Queue;
        while (my $num = $upstream->dequeue) {
                next unless $num % $cur_prime;
                if ($kid) {
                        $downstream->enqueue($num);
                } else {
                        print "Found prime $num\n";
                        $kid = new threads(\&check_num, $downstream, $num);
                }
        }
        $downstream->enqueue(undef) if $kid;
        $kid->join if $kid;
}
```

9.7.2. Asignación de trabajo a threads en un pipe

```
package Threads::Pipe;
use 5.008004;
use strict;
use warnings;
use threads;
use Thread::Queue;
use Carp;
require Exporter;
our @ISA = qw(Exporter);
our @EXPORT_OK = ( 'pipe' );
our $VERSION = '0.01';

# Virtualización de un pipe.
# Se supone que el número de procesos virtuales requeridos
# es conocido de antemano.
```

```
sub pipe { # crea el anillo que virtualiza el pipe de tamaño N
  my $nt = shift(); # número de threads
  my $N = shift(); # número de procesadores virtuales
  my $task = shift(); # subrutina a ejecutar

  my @t; # array de tareas
  my @channel; # array de colas
  # El resto de argumentos son argumentos de $task

  my $id; # identificador de procesador físico

  # creamos canales ...
  for($id=0; $id < $nt; $id++) {
    $channel[$id] = new Thread::Queue; # canal comunicando procesos $id y ($id+1)%$nt
  }

  # creamos threads ...
  for($id=0; $id < $nt; $id++) {
    my $wrap = sub { # clausura que envuelve a la función de usuario $task
      my $i; # identificador de proceso virtual
      my @results;

      for($i = $id; $i < $N; $i += $nt) {
        my $result_i = $task->($i,  $channel[($id+$nt-1)%$nt], $channel[$id], @_);
        push @results, $result_i;
      }
      return \@results;
    }; # end wrap

    $t[$id] = threads->new($wrap, @_); # contexto escalar: retorna escalar
  }
  # La thread 0 no trabaja, sólo espera  ...
  my @result;
  for($id=0; $id < $nt; $id++) {
    $result[$id] = $t[$id]->join(); # join debe ser ejecutado por el padre
  }
  # shuffle
  my @R;
  for($id=0; $id < $nt; $id++) {
    my @aux = @{$result[$id]};
    for my $i (0..$#aux) {
      $R[$id+$nt*$i] = $aux[$i];
    }
  }
  return @R;
}

1;
```

Programa:

```
$ cat -n pipe3.pl
 1  #!/usr/bin/perl -w -I../lib
```

```
# File: pipe3.pl

use strict;
use threads;
use Thread::Queue;
use Threads::Pipe qw(pipe);

our $numthreads = (shift || 2); # número de threads "físicas"
our $numvirtual = (shift || 8); # número de threads "virtuales"
our $nummessages = (shift || 4); # número de mensajes por etapa

### main ###
&pipe($numthreads, $numvirtual, \&job, $nummessages);

sub job {
  my $vid = shift;
  my $from_left = shift;
  my $to_right = shift;
  ##############
  my $nummessages = shift;
  my $id = threads->self->tid(); # identificado de ithread Perl

  my ($i, $num);
  if ($vid) {
    for $i (1..$nummessages) {
      $num = $from_left->dequeue;
      # procesar número ...
      my $processed = $num*$vid;
      print "id=$id vid=$vid: num=$num processed=$processed\n";
      $to_right->enqueue($num);
    }
  }
  else {
    for $i (1..$nummessages) {
      print "id=$id vid=$vid: num=$i\n";
      $to_right->enqueue($i);
    }
  }
}
```

Ejecución:

```
$ pipe3.pl
id=1 vid=0: num=1
id=1 vid=0: num=2
id=1 vid=0: num=3
id=1 vid=0: num=4
id=2 vid=1: num=1 processed=1
id=1 vid=2: num=1 processed=2
id=2 vid=1: num=2 processed=2
id=1 vid=2: num=2 processed=4
id=2 vid=1: num=3 processed=3
id=1 vid=2: num=3 processed=6
id=2 vid=1: num=4 processed=4
```

```
id=1 vid=2: num=4 processed=8
id=2 vid=3: num=1 processed=3
id=1 vid=4: num=1 processed=4
id=2 vid=3: num=2 processed=6
id=2 vid=3: num=3 processed=9
id=1 vid=4: num=2 processed=8
id=2 vid=3: num=4 processed=12
id=1 vid=4: num=3 processed=12
id=2 vid=5: num=1 processed=5
id=1 vid=4: num=4 processed=16
id=2 vid=5: num=2 processed=10
id=1 vid=6: num=1 processed=6
id=1 vid=6: num=2 processed=12
id=2 vid=5: num=3 processed=15
id=1 vid=6: num=3 processed=18
id=2 vid=5: num=4 processed=20
id=1 vid=6: num=4 processed=24
id=2 vid=7: num=1 processed=7
id=2 vid=7: num=2 processed=14
id=2 vid=7: num=3 processed=21
id=2 vid=7: num=4 processed=28
```

9.8. El Problema de la mochila 0-1

Utilizando programación dinámica es posible resolver el problema de la mochila 0-1: Se tiene una mochila de de capacidad M y N objetos de pesos w_k y beneficios p_k. Se trata de encontrar la combinación óptima de objetos que caben en la mochila y producen el máximo beneficio. Para ello denotemos por $f_k(c)$ el beneficio óptimo obtenido usando los primeros k objetos y una mochila de capacidad c. Entonces $f_0(c) = p_0$ si $c \geq w_0$ y 0 en otro caso. Además

$f_k(c) = \text{máx}\{f_{k-1}(c), f_k - 1(c - w_k) + p_k\}$ si $c > w_k$ y $f_k(c) = f_{k-1}(c)$ en otro caso.

El esquema del algoritmo secuencial es:

```
for my $c (0..$M) {
  $f[0][$c] = ($w[0] <= $c)? $p[0] : 0;
}

for my $k (1..$N-1) {
  for my $c (0..$M) {
    my $n = $f[$k-1][$c];
    if ($c >= $w[$k]) {
      my $y = $f[$k-1][$c-$w[$k]]+$p[$k];
      $f[$k][$c] = ($n < $y)? $y : $n;
    }
    else { $f[$k][$c] = $n; }
  }
}
```

En la dirección:

http://nereida.deioc.ull.es/~lhp/perlexamples/Algorithm-Knap01DP-0.01.tar.gz.

puede encontrar una distribución de un módulo Perl que proporciona una solución secuencial.

He aqui una solución paralela:

```
$ cat -n pipeknap.pl
 #!/usr/bin/perl -w -I../lib
```

```perl
# File: pipeknap.pl

use strict;
use threads;
use Thread::Queue;
use Carp;
use IO::File;
use Threads::Pipe qw(pipe);

### main
#my ($M, $w, $p) = ReadKnap($ARGV[0]);
srand(27);
my ($M, $w, $p) = GenKnap($ARGV[0]);
my $N = @$w;
my $numthreads = ($ARGV[1] || 2);

my @f = &pipe($numthreads, $N, \&knap, $M, $w, $p);

my @Table = GetSol(@f);
print "Sol = $Table[-1][-1]\n";

sub knap {
  my $k = shift();
  my $from_left = shift();
  my $to_right = shift();
  ##############
  my $M = shift();
  my @w = @{shift()};
  my @p = @{shift()};

  my $N = @w;
  my @f;
  my @left;

  croak "Profits and Weights don't have the same size" unless scalar(@w) == scalar(@p);

  if ($k) {
    for my $c (0..$M) {
      $left[$c] = $from_left->dequeue();
      if ($c >= $w[$k]) {
        my $y = $left[$c-$w[$k]]+$p[$k];
        $f[$c] = ($left[$c] < $y)? $y : $left[$c];
      }
      else {
        $f[$c] = $left[$c];
      }
      $to_right->enqueue($f[$c]);
    }
  }
  else { # thread virtual 0
    for my $c (0..$M) {
      $f[$c] = ($w[0] <= $c)? $p[0] : 0;
    }
```

```
    $to_right->enqueue(@f);
  }
  return (\@f);
}

sub ReadKnap {
  my $filename = shift;

  my $file = IO::File->new("< $filename");
  croak "Can't open $filename" unless defined($file);
  my (@w, @p);

  my ($N, $M);

  chomp($N = <$file>);
  chomp($M = <$file>);
  for (0..$N-1) {
    $w[$_] = <$file>;
    $p[$_] = <$file>;
  }
  chomp @w;
  chomp @p;
  $file->close();
  return ($M, \@w, \@p);
}

sub GenKnap {
  my $N = (shift() || 10000);
  my $R = (shift() || $N);

  my ($x, $M, @w);
  @w = map { $x = 1 + int(rand($R)); $M += $x; $x } 1..$N;
  $M = int ($M / 2);
  return ($M, \@w, \@w);
}

sub GetSol {
  my @f = @_;

  my $k = 0;
  for my $t (@f) {
    my $c = 0;
    for my $fkc (@$t) {
      $Table[$k][$c] = $fkc;
      $c++;
    }
   $k++;
   }
   return @Table;
 }
```

9.9. Práctica: Aumentando el Grano

Aumente el grano de la solución anterior haciendo que cada thread procese $f_k(c)$ para todo $0 \leq c \leq M$ y un bloque de tamaño g de objetos.

9.10. Práctica: El Problema de Asignación de un Único Recurso

Resuelva utilizando programación dinámica el problema de asignación de un único recurso y diseñe un algoritmo paralelo usando el esqueleto anterior.

9.11. Práctica: La criba de Eratostenes

Adapte el esquema explicado al ejemplo de paralelización del cómputo de los primeros números primos estudiada en la sección 9.7.1

9.12. `Net::SSH::Perl`

9.12.1. Ejemplo de uso

El módulo `Net::SSH::Perl` proporciona las funcionalidades de un cliente SSH. Es compatible con los protocolos SSH-1 y SSH-2.

```
 1 #!/usr/bin/perl -w
 2 use Net::SSH::Perl;
 3 use strict;
 4 my ($stdout, $stderr, $exit) = ('', '', '');
 5 my ($user) = ('yourlogin');
 6 my $pass = 'yourPassword';
 7 my $cmd = 'ls -l';
 8 my $host = 'yourmachine';
 9 my $ssh = Net::SSH::Perl->new($host);
10 $ssh->login($user, $pass);
11 ($stdout, $stderr, $exit) = $ssh->cmd($cmd);
12 print "STDOUT =\n$stdout";
13 print "STDERR =\n$stderr" if $stderr;
```

El constructor `new` (línea 9) establece la conexión con `$host`. Entre los parámetros que admite están:

- El protocolo, e cuál puede ser 2 ó 1 ó '2,1' 'o '1,2'. Los dos últimos indican que ambos protocolos pueden ser usados pero especifican un orden de preferencia.
- El cifrado, el cuál puede ser IDEA, DES, DES3 y blowfish en SSH1 y arcfour, blowfish-cbs y 3des-cbs en SSH-2.
- El puerto al que el demonio `sshd` debe conectarse.

Ejecución:

```
$ ./example.pl
Argument "ssh-dss" isn't numeric in numeric eq (==) at /usr/local/share/perl/5.8.4/Net/SSH/Perl/Key/RSA1.pm line 94, <FH> line 20.
STDOUT =
total 24
drwxr-xr-x    3 casiano  ull         4096 Jul 17  2002 ar
drwxr-xr-x    2 casiano  ull         4096 May  6  2002 bin
drwxr-xr-x    2 casiano  ull           52 Dec 17  2001 etc
drwxr-xr-x    2 casiano  ull           45 Mar 15  2000 include
```

```
drwxr-xr-x    4 casiano  ull         4096 Aug  8  2002 papiguide
drwxr-xr-x    4 casiano  ull           43 Apr  4  2002 repository
lrwxr-xr-x    1 casiano  ull           20 Jul 17  2002 scratch -> /scratch_tmp/casiano
lrwxr-xr-x    1 casiano  ull           26 Jul 17  2002 scratch1 -> /scratch/files/ull/casiano
drwxr-xr-x    8 casiano  ull          107 Mar  6  2003 src
drwxr-xr-x    2 casiano  ull           95 Aug  8  2002 tmp
lrwxr-xr-x    1 casiano  ull           21 Mar  7  2003 unstable -> scratch/unstable/CALL
drwxr-xr-x    5 casiano  ull          143 Oct 26  2004 work
STDERR =
MANPATH: Undefined variable.
```

9.12.2. Combinando con threads

El siguiente ejemplo muestra como dos threads generan conexiones ssh con dos máquinas ejecutando el mismo comando en ambas.

```
#!/usr/bin/perl -w
use strict;
use Net::SSH::Perl;
use threads;
use Data::Dumper;

our $numthreads;

my (@pass, @host);

my $user = 'XXXX';

$pass[0] = 'XXXX'; $host[0] = 'XXXX';

$pass[1] = 'XXXX.'; $host[1] = 'XXXX';

sub par {
  my $nt = shift();
  my $task = shift;
  my @t; # array of tasks
  my $i;
  my @results;

  for($i=1; $i < $nt; $i++) {
    $t[$i] = threads->new($task, @_);
  }
  @{$results[0]} = $task->( @_);
  for($i=1; $i < $nt; $i++) {
    @{$results[$i]} = $t[$i]->join();
  }
  @results;
}

sub ssh {
  my $cmd = shift();
  my $id = threads->self()->tid();
  my $pass = $pass[$id];
  my $host = $host[$id];
```

```
  my ($stdout, $stderr, $exit) = ('', '', '');
  my $ssh = Net::SSH::Perl->new($host);
  $ssh->login($user, $pass);
  [ $ssh->cmd($cmd) ];
}

$numthreads = (shift || 2);
my @results = par($numthreads, \&ssh, 'pwd');

print Dumper(\@results);
```

Ejecución:

```
$ ./example2.pl
Argument "ssh-rsa" isn't numeric in numeric eq (==) at /usr/local/share/perl/5.8.4/Net/SSH/Perl/Key/RSA1.pm line 94, <FH> line 6.
Argument "ssh-dss" isn't numeric in numeric eq (==) at /usr/local/share/perl/5.8.4/Net/SSH/Perl/Key/RSA1.pm line 94, <FH> line 20.
$VAR1 = [ [ [ '/user1/uni/ull/casiano ', 'MANPATH: Undefined variable.  ', 0 ] ],
          [ [ '/home/casiano ', undef, 0 ] ]
        ];
```

9.13. Módulos para FTP Seguro

9.13.1. `Net::SFTP`

El módulo `Net::SFTP` es una implementación en Perl del protocolo SFTP construido sobre el protocolo SSH. `Net::SFTP` usa `Net::SSH::Perl` para construir una conexión segura para la transferencia de ficheros.

```
 cat -n examplenetsftp2.pl
     1  #!/usr/bin/perl -w
     2  use Net::SFTP;
     3  use strict;
     4  use Data::Dumper;
     5
     6  our $count = 0;
     7  my $user = 'casiano';
     8  my $host = (shift or die "Dame una máquina!\n");
     9  my $pass = (shift or die "Dame una paspassn");
    10  my $sftp = Net::SFTP->new($host, user=>$user, password=>$pass);
    11  $sftp->put($0, 'borrame.txt');
    12  $sftp->ls('.', sub { print "$_[0]->{filename}\t"; print "\n" if $count++ % 5 == 0 } );
    13  print "\n";
```

En la línea 12 obtenemos la lista de ficheros. La interfaz del método `ls` es:

```
       $sftp->ls($remote [, $subref ])
```

donde `$remote` es el directorio en la máquina remota. Si se especifica `$subref` entonces la subrutina apuntada por `$subref` será llamada para cada entrada del directorio. Se le pasa una referencia a un hash con tres claves:

- `filename`, El nombre de la entrada en el listado del directorio.
- `longname`, una entrada detallada tipo `ls -l`.
- `a`, Un objeto `Net::SFTP::Attributes` conteniendo los atributos del fichero (atime, mtime, permisos, etc.).

Veamos un ejemplo de ejecución:

```
$ ./examplenetsftp2.pl maquina password
.
..      tmp     .screenrc       .bash_logout    .bash_profile
.bashrc .mailcap        .bash_history   .gimp-1.2       .paraver
.viminfo        public_html     lhp     .Xauthority     pl
borrame.txt     .bashrc~        .ssh    examplesftp.pl
```

9.13.2. `Net::SFTP::Foreign`

El módulo `Net::SFTP::Foreign` proporciona un cliente SFTP en Perl. Para ello utiliza un cliente ssh externo. A diferencia de `Net::SFTP` y de `Net::SSH` no permite el uso de claves (passwords) como argumentos.

```
 1 #!/usr/bin/perl -w
 2 use Net::SFTP::Foreign;
 3 use strict;
 4 use Data::Dumper;
 5
 6 our $count = 0;
 7 my ($stdout, $stderr, $exit) = ('', '', '');
 8 my $user = 'XXXX';
 9 my $host = 'YYY';
10 my $sftp = Net::SFTP::Foreign->new(host=>$host, user=>$user);
11 $sftp->put('/home/lhp/Lperl/src/ssh/examplesftp.pl', 'borrame.txt');
12 $sftp->ls('.', sub { print "$_[0]->{filename}\t"; print "\n" if $count++ % 5 == 0 } );
13 print "\n";
```

La conexión sftp se crea en la línea 10. El constructor `new` crea un objeto que representa la conexión abierta. Los métodos `get` y `put` permiten mover ficheros entre las máquinas. El método `ls` nos devuelve una estructura de datos con información específica sobre cada entrada del directorio. Los tres métodos admiten un parámetro que referencia a una subrutina que se ejecutará sobre cada entrada.

Sigue el resultado de la ejecución. La apertura de la conexión en la línea 10 hace que se pida el password. Una alternativa es usar claves RSA o DSA.

```
$ ./examplesftp.pl
DSA host key for IP address '147.83.42.28' not in list of known hosts.
XXXX@YYY's password: # el usuario rellena la password una vez
MANPATH: Undefined variable.
.
..      ar      bin     etc     src
tmp     .ssh    examplesftp.pl  papiguide       work
.pvmcshrc       .DIMEMAS_defalults      .adprc  scratch1        scratch
include .cshrc  unstable        .nanoscshrc     .news_time
.Xauthority     .nqshosts       borrame.txt     .loggg  .login
.sh_history     .exrc   .papi   .ssh2   repository
.profile        .forgex
```

Capítulo 10

Arañas, Análisis de HTML y XML

Este capítulo muestra como escribir un programa que rellene un formulario de una página web, emita la correspondiente petición y analice la página devuelta por el servidor.

10.1. Decargando Páginas

La librería `LWP` (The World-Wide Web library for Perl) proporciona clases y funciones que permiten escribir clientes WWW. La librería también contiene clases y métodos que dan soporte a la construcción de servidores HTTP sencillos.

La versión mas sencilla de esta librería la proporciona el módulo `LWP::Simple`. Como su nombre indica, provee una versión simplificada de la librería. Esto la hace especialmente indicada para ejecutar *one-liners*: programas Perl que pese a ejecutarse en una sóla línea, normalmente desde la línea de comandos, pueden llegar a realizar tareas de considerable complejidad. Por ejemplo para descargar una página web en el fichero `ejercicio.html` podemos ejecutar el siguiente comando:

```
$ perl -MLWP::Simple -e 'getstore("http://nereida.deioc.ull.es/~lhp/perlexamples/node55.html", \
                                   "ejercicio.html")'
$ ls -ltr | tail -3
-rw-r--r--  1 lhp lhp  7957 2005-06-30 10:09 node343.html
-rw-r--r--  1 lhp lhp 18099 2005-06-30 10:09 node211.html
-rw-r--r--  1 lhp lhp  7093 2005-06-30 10:31 ejercicio.html
```

Si lo que queremos es descargar un buen número de páginas puede ser una buena idea paralelizar la descarga para así aumentar el ancho de banda. Para ello podemos usar el módulo de `threads` presentado en el capítulo 9 o cualquier otro que provea las funcionalidades requeridas, por ejemplo el módulo `Parallel::ForkManager`. Veamos un ejemplo de uso:

```
$ cat -n parfork.pl
#!/usr/bin/perl -w
use Parallel::ForkManager;
use LWP::Simple;
my $pm=new Parallel::ForkManager(10);
for my $link (@ARGV) {
  $pm->start and next;
  my ($fn) = $link =~ /^.*\/(.*?)$/; # contexto de lista
  if (!$fn) { # $fn  es $1
    warn "Cannot determine filename from $fn\n";
  } else {
    # Se encontró una cadena despúes del último /
    print "Getting $fn from $link\n";
    my $rc=getstore($link,$fn);
    print "$link downloaded. response code: $rc\n";
```

```
15    };
16    $pm->finish;
17  };
```

En la línea 4 se crea un objeto gestor de procesos para un máximo de 10 procesos. En la línea 6 se hace uso de `$pm->start` para hacer el `fork`. El método devuelve 0 al proceso hijo y el PID del hijo al padre. Asi pues, al ser evaluada la expresión en circuito corto, el `next` hace que el proceso padre se salte la tarea de descarga (líneas 7-14). La llamada `pm->start` produce la muerte prematura del programa si falla el `fork`. El valor retornado por la llamada a `getstore` es el código HTTP de respuesta proporcionado por el servidor. Por último la llamada a `finish` en la línea 16 termina el proceso hijo. Es posible usar el método `wait_all_children` para esperar por la terminación de todos los procesos que han sido arrancados por el objeto `$pm`.

Sigue un ejemplo de ejecución:

```
$ parfork.pl 'http://nereida.deioc.ull.es/~pl/pl0405/node24.html'\
              'http://nereida.deioc.ull.es/~lhp/perlexamples/node343.html'
Getting node24.html from http://nereida.deioc.ull.es/~pl/pl0405/node24.html
Getting node343.html from http://nereida.deioc.ull.es/~lhp/perlexamples/node343.html
http://nereida.deioc.ull.es/~lhp/perlexamples/node343.html downloaded. response code: 200
http://nereida.deioc.ull.es/~pl/pl0405/node24.html downloaded. response code: 200
$ ls -ltr | tail -4
drwxr-xr-x  2 lhp lhp  4096 2005-06-02 14:22 05
-rwxr-xr-x  1 lhp lhp   420 2005-06-30 10:55 parfork.pl
-rw-r--r--  1 lhp lhp  7957 2005-06-30 11:05 node343.html
-rw-r--r--  1 lhp lhp  7744 2005-06-30 11:05 node24.html
$
```

10.2. Búsqueda en formularios con `get`

Nuestro primer ejemplo considera el buscador de la Universidad de La Laguna (ULL), el cual permite la búsqueda de personal vinculado a la universidad.

En primer lugar, visite la página en la que esta ubicado: http://www.ccti.ull.es/bd/. La figura 10.1 muestra la interfase que percibe un usuario humano.

La estructura de la página que contiene al formulario es como sigue:

```
<form method="GET" action="index.php3">
<font face="Verdana" size="2"><span class="normal">Buscar </span></font>
<input class="ObjetosTexto" type="text" name="cadena" size="20" maxlength="20"
       value ="" title="Cadena que desea buscar">
<font face="Verdana" size="2"><span class="normal"> en </span></font>
<font class="ObjetosSelect">
<select title="Concepto por el que se busca la información"
        class ="ObjetosSelect" name="donde" size="1">
  <option  value="1">el correo eléctrónico</option>
  <option selected value="2">el nombre y apellidos</option>
  <option  value="3">la dirección de trabajo</option>
  <option  value="4">la unidad organizativa</option>
  <option  value="5">la relación con la unidad organizativa</option>
</select>
</font>
<input type="submit" value="Buscar" name="BotonBuscar" class="ObjetosBoton"></p>
</form>
```

El atributo `method` indica si usamos GET o POST al enviar el formulario. El atributo `action` determina la URL que recibirá los datos. Los componentes de un formulario son text-fields, listas

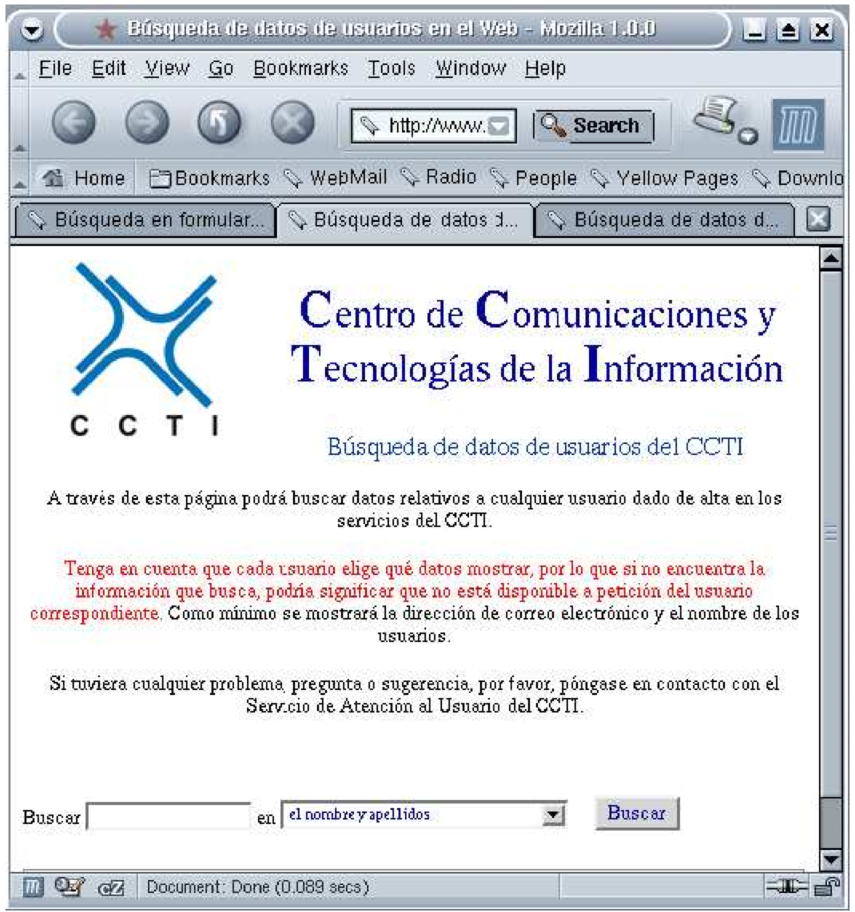

Figura 10.1: El buscador de personas de la ULL

drop-down, checkboxes, etc. cada uno identificado con un `name`. Aqui el campo `input` define la `cadena` a buscar cuando se emita el formulario al seleccionar el botón `Buscar`. La etiqueta `select` permite crear un menú desplegable utilizando la lista de elementos etiquetados con `option`. Nos concentraremos en la selección 2 del menú `donde` para nuestra búsqueda: buscar por nombre y apellidos.

Una vez realizada la búsqueda, la estructura de la página de salida es tal que contiene al menos tres tablas cuando se encuentran resultados. La tercera tabla es la que contiene los resultados. Una fila tiene un aspecto parecido al siguiente:

```
<table border="0" width="100%">
<tr>
  <td>
  <a class="all" href="mailto:xxxx@ull.es" title="Póngase en contacto con el usuario">
  xxxx@ull.es</a>
  </td>
  <td> María Teresa .....  ....... </td>
  <td> ** Dirección de trabajo ** </font>
  </td>
  <td> Departamento de Anatomía, Anatomía Patológica e Histología </td>
  <td> Docentes y personal investigador </td>
  <td> <a name="L1724"> </a><a href="#L1724"> Teléfonos:</a><br>
    1er tlfn contacto: 922319444<br>
    Fax trabajo: 922319999<br>
  </td>
</tr>
```

Solución:

```
#! /usr/bin/perl -w

use URI;
use LWP::UserAgent;
use HTML::TreeBuilder 3;

die "Uso $0 \"nombre\"\n" unless @ARGV == 1;
my $name = $ARGV[0];
my $url = URI->new('http://www.ccti.ull.es/bd/index.php3');
$url->query_form('cadena'=>$name, 'donde'=>'2', 'BotonBuscar'=>'Buscar');
#
#print $url, "\n";
```

Un URL nos dice como obtener algo: *utiliza HTTP con este servidor y pídele esto* o bien *conéctate por ftp a esta máquina y carga este fichero*. La llamada anterior

`$url = URI->new('http://www.ccti.ull.es/bd/index.php3')`

crea un objeto URI representando una URL.

La forma en la que enviamos el formulario usando LWP depende de si se trata de una acción GET o POST. Si es un GET se procede como en el ejemplo: se crea una URL con los datos codíficados usando el método `$url->query_form('cadena'=>$name, 'donde'=>'2', 'BotonBuscar'=>'Buscar')`. y después se procede a llamar al método `get`. En el código que nos ocupa, se crea el agente y se llama al método `get` en una sóla expresión:

```
my $response = LWP::UserAgent->new->get( $url );
die "Error: ", $response->status_line unless $response->is_success;
```

El constructor `new` de la clase `LWP::UserAgent` nos permite utilizar un objeto de la clase agente el cual nos permite gestionar conexiones HTTP y realizar peticiones a los servidores HTTP.

Si el método es POST se debe llamar al método `$browser->post()` pasándole como argumento una referencia al vector de parámetros del formulario.

De hecho, si estuvieramos buscando por una cadena fija, digamos `Casiano`, la línea anterior podría ser sustituida por:

```
$response = LWP::UserAgent->new->get(http://www.ccti.ull.es/bd/index.php3?cadena=Casiano&donde=2)
```

Acto seguido construimos el árbol de análisis sintáctico:

```
my $root = HTML::TreeBuilder->new_from_content($response->content);

#$root->dump;

my @tables = $root->find_by_tag_name('table');
```

La llamada `HTML::TreeBuilder->new_from_content($response->content)` crea el árbol sintáctico del HTML proporcionado por el objeto agente y que fué dejado en el objeto `$response`. El objeto `$response` es de la clase `HTTP::Response`.

En general la sintáxis es:

`HTML::TreeBuilder->new_from_content([ string, ... ])`

Si usamos en vez `new_from_file` construiríamos el árbol para el fichero pasado como argumento.

El método `find_by_tag_name` devuelve la lista de nodos etiquetados con el correspondiente *tag* pasado como argumento. En el ejemplo, la llamada `$root->find_by_tag_name('table')` devuelve una lista con los nodos del árbol que estan etiquetados con la marca `table`.

Si se descomenta la línea `#$root->dump;` se produce un volcado del árbol sintáctico del HTML. La estructura del volcado (abreviada) es similar a esto:

```
<html> @0
  <head> @0.0
    <title> @0.0.0
     ......
    <link href="styles.css" name="style1" rel="stylesheet" type="text/css"> @0.0.1
  <body alink="#800000" link="#000080" vlink="#800000"> @0.1
     ......
```

Cada nodo aparece etiquetado con su nivel de anidamiento.

a continuación el programa comprueba que existen tres tablas en el árbol y procede a recorrer el subárbol de la segunda tabla:

```
if (@tables > 2) {
  my $result = $tables[2];

  my @stack = ($result);
  while (@stack) {
    my $node = shift @stack;
    if (ref $node) {
      unshift @stack, $node->content_list;
    } else {
      print $node,"\n";
    }
  }
} elsif ($root->as_text =~ /Consulta demasiado/) {
  print "Consulta demasiado extensa para $name\n";
}
else {
  print "No se encontró $name";
}
$root->delete;
```

Como se ha dicho la tercera tabla (`my $result = $tables[2]`) es la que contiene los resultados. La llamada al método `$node->content_list` devuelve la lista de los hijos del nodo. Asi pues, la orden `unshift` coloca dichos nodos al comienzo de `@stack` (por cierto, que debería haberse llamado `@queue` ya que es usada como una cola). Si el nodo `$node` no es una referencia es que se trata de una hoja, en cuyo caso contiene información y la mostramos.

Ejemplo de ejecución:

```
$ ./search_ull.pl Coromoto
xxxxx@ull.es
Lourdes Coromoto Pérez González
** Dirección de trabajo **
Departamento de Filos. de la Memoria  y Propología.
Docentes y personal investigador
Teléfonos:
1er tlfn contacto: 922444444
Fax trabajo: 922555555


yyyyy@ull.es
(Visite su página)
Coromoto González Pérez
c/Astrofísico Paco. Sánchez s/n.
Departamento de FitoPato, Postulo. y Pomputación
```

```
Docentes y personal investigador
Teléfonos:
1er tlfn contacto: 922406090
Fax trabajo: 922789341
```

Sigue el código completo del programa:

```perl
#! /usr/bin/perl -w

use URI;
use LWP::UserAgent;
use HTML::TreeBuilder 3;

die "Uso $0 \"nombre\"\n" unless @ARGV == 1;
my $name = $ARGV[0];
my $url = URI->new(’http://www.ccti.ull.es/bd/index.php3’);
$url->query_form(’cadena’=>$name, ’donde’=>’2’, ’BotonBuscar’=>’Buscar’);
#
#print $url, "\n";

my $response = LWP::UserAgent->new->get( $url );
die "Error: ", $response->status_line unless $response->is_success;

my $root = HTML::TreeBuilder->new_from_content($response->content);

#$root->dump;

my @tables = $root->find_by_tag_name(’table’);

if (@tables > 2) {
  my $result = $tables[2];

  my @stack = ($result);
  while (@stack) {
    my $node = shift @stack;
    if (ref $node) {
      unshift @stack, $node->content_list;
    } else {
      print $node,"\n";
    }
  }
} elsif ($root->as_text =~ /Consulta demasiado/) {
  print "Consulta demasiado extensa para $name\n";
}
else {
  print "No se encontró $name";
}
$root->delete;
```

10.3. Búsqueda en formularios con `post` y autentificación

El siguiente ejemplo muestra como acceder a una página protegida bajo autentificación (básica), hacer una búsqueda a través de un formulario `post` y analizar los resultados.

Como ejemplo usaremos las páginas de búsqueda en el servidor de gestión integral de la Escuela Superior de Ingeniería Informática de la Universidad de La Laguna: http://w4.csi.ull.es/personal/cabecera.php3.

Lo primero es que nos pedirá autentificarnos. Observe bien el `realm` (`Autentificacion`).

Una vez atravesada la fase de autentificación, el formulario a rellenar es como sigue:

```
<form method="POST" action="busqueda.php3" target="pie" name="formu">
  <table width=50% cellspacing="0" cellpadding="0" border=0 bgcolor="#DEDEDE">
    <tr><td>Texto a buscar : <input type=text name=TEXTO size=10></td>
    <td><input type=submit value="Buscar"> </td></tr>
  </table>
</form>
```

Y la salida tiene este aspecto:

```
<table  cellspacing=3  cellpadding=3 width=90% align=center>
  <tr>
     <td bgcolor="#CBCCEB" width=60%><font size=+1><b>Nombre</b></font></td>
     <td bgcolor="#CBCCEB"><font size=+1><b>Username</b></font></td>
     <td bgcolor="#CBCCEB"><font size=+1><b>Departamento</b></font></td>
  </tr>
  <tr>
     <td><a href="info.php3?USER=fabiano" >RODRIGUEZ GIL, FABIANO</a></td>
     <td>fabiano</td><td>ESTADÍSTICA, I.O. y C.</td>
  </tr>
</table>
<table  cellspacing=3  cellpadding=3 width=70% align=center>
  <tr bgcolor="#CBCCEB">
     <td><font size=+1><b>Username</b></font></td>
     <td width=65%><font size=+1><b>Nombre</b></font></td></tr>
  <tr>
     <td><a href="mailto:fabiano@etsii.ull.es">fabiano</a></td>
     <td>Casiano Rodriguez Gil</td>
   </tr>
   <tr>
     <td><a href="mailto:call@etsii.ull.es">call</a></td>
     <td>Proyecto de German con Fabiano</td></tr>
</table>
```

Obsérvese que contiene dos tablas (no siempre).

Sigue una solución:

```
#! /usr/bin/perl -w

use strict;
use LWP::UserAgent;
use HTML::TreeBuilder 3;

die "Uso $0 \"nombre\"\n" unless @ARGV == 1;
my $name = $ARGV[0];

my $browser = LWP::UserAgent->new;

$browser->credentials(
'w4.etsii.ull.es:80',
'Autentificacion',
```

```
'alu999'=>'alu999password'
);

my $response = $browser->post(
  'http://w4.etsii.ull.es/personal/busqueda.php3',
  ['TEXTO'=>$name]
);
die "Error: ", $response->status_line unless $response->is_success;

my $root = HTML::TreeBuilder->new_from_content($response->content);
my @rows = $root->find_by_tag_name('tr');

foreach my $n (@rows) {
  my @fields = $n->find_by_tag_name('td');
  foreach my $f (@fields) {
    print $f->as_trimmed_text(),"\t";
  }
  print "\n";
}

$root->delete;
```

10.4. Buscando en Amazon

El siguiente ejemplo es una modificación del que aparece en el libro de Bausch *Amazon Hacks, 100 Industrial-Strength Tips and Tools* [15].

```
$ cat ./amazon_http.pl
#!/usr/local/bin/perl5.8.0
# amazon_http.pl
# A typical Amazon Web API Perl script using the XML/HTTP interface
# Usage: amazon_http.pl <keyword>

#Your Amazon developer's token
my $dev_key='insert developer token';

#Your Amazon affiliate code
my $af_tag='insert associate tag';

#Take the keyword from the command-line
my $keyword =shift @ARGV or die "Usage:perl amazon_http.pl <keyword>\n";

#Assemble the URL
my $url = "http://xml.amazon.co.uk/onca/xml3?t=" . $af_tag .
    "&dev-t=" . $dev_key .
    "&type=lite&f=xml&mode=books&" .
    "KeywordSearch=" . $keyword;

use strict;

#Use the XML::Parser  and LWP::Simple Perl modules
use XML::Simple;
use LWP::Simple;
```

```
my $content = get($url);
die "Could not retrieve $url" unless $content;

my $xmlsimple = XML::Simple->new( );
my $response = $xmlsimple->XMLin($content);

my $details = $response->{Details};
if (ref($details) eq 'ARRAY') {
  foreach my $result (@$details){
      #Print out the main bits of each result
      print
      join "\n",
      $result->{ProductName}||"no title",
      "ASIN: " . $result->{Asin} . ", " .
      $result->{OurPrice} . "\n\n";
  }
} # sigue la modificación al del libro
elsif (ref($details) eq 'HASH') {
  print
  join "\n",
  $details->{ProductName}||"no title",
  "ASIN: " . $details->{Asin} . ", " .
  $details->{OurPrice} . "\n\n";
}
else {
  print "No encontré nada\n";
}
```

Véase un ejemplo de ejecución con una sóla respuesta:

```
$ ./amazon_http.pl 'Perl Medic'
Perl Medic: Transforming Legacy Code
ASIN: 0201795264, $24.49
```

Un ejemplo con ninguna respuesta:

```
$ ./amazon_http.pl 'Perl toto'
No encontré nada
```

y un ejemplo de ejecución con múltiples respuestas:

```
$ ./amazon_http.pl 'Perl CGI'
Programming Perl (3rd Edition)
ASIN: 0596000278, $34.96

JavaScript: The Definitive Guide
ASIN: 0596000480, $31.46

Learning Perl, Third Edition
ASIN: 0596001320, $23.77

JavaScript Bible, Fifth Edition
ASIN: 0764557432, $33.99

Perl Cookbook, Second Edition
ASIN: 0596003137, $33.97
```

```
Mastering Regular Expressions, Second Edition
ASIN: 0596002890, $27.17

Dive Into Python
ASIN: 1590593561, $27.99

JavaScript & DHTML Cookbook
ASIN: 0596004672, $27.17

JavaScript Bible, 4th Edition
ASIN: 0764533428, $33.99

Perl CD Bookshelf, Version 4.0
ASIN: 0596006225, $67.97
```

Parte III

Análisis Léxico y Sintáctico

Capítulo 11

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 11.1 comenzamos haciendo un breve repaso a como construir un módulo en Perl. Si quieres tener un conocimiento mas profundo lee el capítulo 6. La sección 11.2 describe las fases en las que -al menos conceptualmente- se divide un compilador. A continuación la sección 11.3 presenta la primera de dichas fases, el análisis léxico. En la sección 11.5 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 11.7 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 11.6 introducimos la que posiblemente sea la mas sencilla de todas: el análisis descendente predictivo recursivo. En la sección 11.8 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 11.9 trata de la construcción de los árboles de análisis abstractos.

11.1. 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 6.

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.
```

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 sobe el esquema de nombres de los módulos y la forma en la que estos se asigna a ficheros del sistema operativo, lea la sección 6.4.

```
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
```

Después de esto tenemos un módulo "funcional" que no hace nada. Lo podríamos instalar como si lo hubieramos descargado desde CPAN (Véase 6.11.1).

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 la sección 6.12.

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 la sección 6.13. 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 11.1.

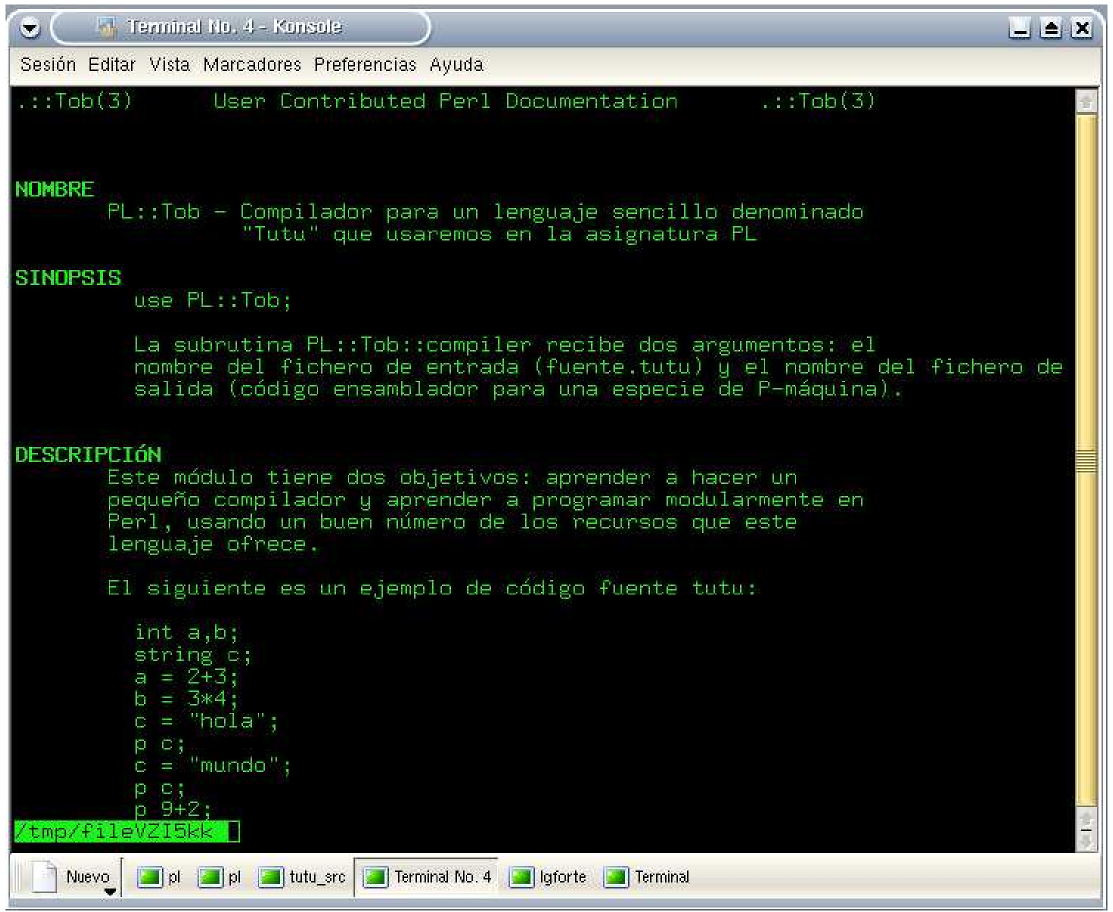

Figura 11.1: El resultado de usar `perldoc Tutu`

11.1.1. 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. ¿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 la sección 6.13.2 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?

11.1.2. Práctica: Crear y documentar el Módulo `PL::Tutu`

Reproduzca los pasos explicados en la sección 11.1 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 la sección 6.11.1.

11.2. Las Fases de un Compilador

La estructura del compilador, descompuesto en fases, queda explicitada en el código de la subrutina `compile`:

```perl
package PL::Tutu;
use 5.008004;
use strict;
use warnings;
use IO::File;
use Carp;

require Exporter;

our @ISA = qw(Exporter);
our @EXPORT = qw( compile compile_from_file);
our $VERSION = '0.01';

our %symbol_table;
our $data;
our $target;
our @tokens;
our $errorflag;
our ($lookahead, $value);
our $tree;
our $global_address;
```

```

# 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) = @_;
  local %symbol_table = ();
  local $data = ""; # Contiene todas las cadenas en el programa fuente
  local $target = ""; # target code
  local @tokens =();
  local $errorflag = 0;
  local ($lookahead, $value) = ();
  local $tree = undef; # abstract syntax tree
  local $global_address = 0;

  ########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;
}

sub compile_from_file {
```

```
  my ($input_name, $output_name) = @_; # Nombres de ficheros
  my $fhi;
  my $targetref;

  if ($input_name and (-r $input_name)) {
    $fhi = IO::File->new("< $input_name");
  }
  else { $fhi = 'STDIN'; }

  ########source input
  local $/ = undef;
  my $input = <$fhi>;

  $targetref = compile($input);

  ########code output
  my $fh = $output_name? IO::File->new("> $output_name") : 'STDOUT';
  $fh->print($$targetref);
  $fh->close;
  1;
}

1;
# Sigue la documentación ...
```

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);
```

El programa `tutu` ilustra otra forma de conseguir que el intérprete Perl busque por la librearía 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);
```

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
```

Ahora tenemos que añadir estos ficheros en `MANIFEST` para que formen parte del proyecto. En vez de eso lo que hacemos es crear un fichero `MANIFEST.SKIP`:

```
lhp@nereida:~/Lperl/src/topdown/PL0506/02fases/PL-Tutu$ cat MANIFEST.SKIP
\.o$
^\.cvsignore$
/\.cvsignore$
\.cvsignore$
CVS/[^/]+$
^Makefile$
/Makefile$
^blib/
\.swp$
\.bak$
\.pdf$
\.ps$
\.sal$
pm_to_blib
\.pdf$
\.tar.gz$
\.tgz$
^META.yml$
```

Para saber mas sobre `MANIFEST` léa la sección 6.16.4.

Es necesario indicarle a Perl que los ficheros añadidos son ejecutables:

```
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.

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', 'blib/arch')" t/*.t
t/PL-Tutu....ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.08 cusr +  0.00 csys =  0.08 CPU)
```

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
```

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
```

11.2.1. 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 = @_;
    ```

19. ¿Que es `\$target`?. Repase la sección 5.1.

20. Toda referencia es un escalar: ¿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 la sección 7.6.

23. ¿Que hace la línea `our @EXPORT = qw( compile compile_from_file)`?

24. ¿Que diferencia hay entre `EXPORT` Y `EXPORT_OK`?. Repase la sección 6.9.

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 la sección 5.4)

28. ¿Que es un hash anónimo? (Repase la sección 5.4)

29. ¿Que hace el operador `=>`?. Repase la sección 1.8.3.

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. ¿Cómo se crea una distribución?

35. ¿Que devuelve `-r $input_name` en la línea 79? Repase la sección 2.2.11.

36. ¿Cuál es la función de la variable mágica `$/`? ¿Que se leerá en la línea 86

    ```
    85   local $/ = undef;
    86   my $input = <$fhi>;
    ```

37. 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);
    ```

11.2.2. Práctica: Fases de un Compilador

Reproduzca los pasos explicados en la sección 11.2 extendiendo el módulo `PL::Tutu` con las funciones de compilación y los correspondientes guiones de compilación.

11.3. 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 el emparejamiento. Sin embargo, en el análisis léxico, el problema es encontrar la subcadena a partir de la última posición “casada” que casa con 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 uno de los patrones/lexemas del lenguaje y añadiéndolo al final de la lista de terminales.

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.

```
...
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 la sección 1.5.

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 casamiento.

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 las secciones 4.3 y 4.4).

La opción `c` y la opción `g` son especialmente útiles para la construcción del analizador. 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 4.12 para mas información sobre la opción `g`).

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 casamiento, obtenemos que la combinación

```
m{\G\s*(...)}gc
```

logra el efecto deseado: digamos que 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.

Por último, la opción `i` permite ignorar el tipo de letra (mayúsculas o minúsculas).

Repase la sección 4.9 para ver algunas de las opciones mas usadas.

```
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 (/\G\s*(.)/gc) {
      Error::fatal "Caracter invalido: $1\n";
    }
    else {
      last;
    }
    redo;
  }
}
```

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 la sección 1.7.3.

11.3.1. 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?

11.3.2. 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?

11.3.3. 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?

11.3.4. Práctica: 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.
- 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 4.15) y la sección 4.13. 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.
- 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`?.

11.4. Haciendo mas Modular el Analizador Léxico

Queremos separar/aislar las diferentes fases del compilador en diferentes módulos. 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
package PL::Tutu;

use 5.008004;
use strict;
use warnings;
use IO::File;
use Carp;
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`.

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`:

```
 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;
 9 use PL::Lexical::Analysis;
10 ...
11
12 sub compile {
13   my ($input) = @_;
14   local %symbol_table = ();
15   local $data = ""; # Contiene todas las cadenas en el programa fuente
16   local $target = ""; # target code
17   my @tokens = ();
18   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
```

```
package PL::Lexical::Analysis;
use strict;
use warnings;
use Carp;
use PL::Error;

require Exporter;

our @ISA = qw(Exporter);
our @EXPORT = qw( scanner );
our $VERSION = '0.01';

sub scanner {
  local $_ = shift;
  my @tokens;

  { # Con el redo del final hacemos un bucle "infinito"
    if (m|\G\s*(\d+)|gc) {
      push @tokens, 'NUM', $1;
    }
    ...
    elsif (m|\G\s*([+*()=;,])|gc) {
      push @tokens, 'PUN', $1;
    }
    elsif (/\G\s*(.)/gc) {
      Error::fatal "Caracter invalido: $1\n";
    }
    else {
      return @tokens;
    }
    redo;
  }
}
```

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);
```

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
```

11.4.1. Comprobando el Analizador Léxico

Queremos comprobar si nuestro código funciona. ¿Cómo hacerlo?. Lo adecuado es llevar una aproximación sistemática que permita validar el código. Esa es la función del programa `test.pl` que se generó automáticamente con `h2xs`.

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 [12]:

- 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

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 [12].

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 ficherojerarquía de ficheros 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
.
|-- Changes
|-- MANIFEST
|-- MANIFEST.SKIP
|-- META.yml
```

```
|-- Makefile.PL
|-- PL-Tutu-0.01.tar.gz
|-- README
|-- bb.txt
|-- lib
|   `-- PL
|       |-- Error.pm
|       |-- Lexical
|       |   `-- Analysis.pm
|       `-- Tutu.pm
|-- scripts
|   |-- test01.sal
|   |-- test01.tutu
|   |-- tutu
|   `-- tutu.pl
`-- t
    `-- 01Lexical.t

5 directories, 15 files
```

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
 6  # change 'tests => 1' to 'tests => last_test_to_print';
 7
 8  use Test::More tests => 2;
 9  BEGIN { use_ok('PL::Lexical::Analysis') };
10  #BEGIN { use_ok('PL::Tutu') };
11
12  #########################
13
14  # Insert your test code below, the Test::More module is use()ed here so read
15  # its man page ( perldoc Test::More ) for help writing this test script.
16
17  my $a = 'int a,b; string c; c = "hello"; a = 4; b = a +1; p b';
18  my @tokens = scanner($a);
19  ok("@tokens" eq
20  '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');
```

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

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', 'blib/arch')" t/*.t
t/01Lexical....ok
All tests successful.
Files=1, Tests=2,  0 wallclock secs ( 0.05 cusr +  0.03 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 -I../lib -v 01Lexical.t
01Lexical....1..2
ok 1 - use PL::Lexical::Analysis;
ok 2 - 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
ok
All tests successful.
Files=1, Tests=2,  0 wallclock secs ( 0.05 cusr +  0.00 csys =  0.05 CPU)
```

Repáse la sección 6.16.7 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`?

11.4.2. Práctica: Pruebas en el Análisis Léxico

Extienda su compilador para modularizar el analizador léxico tal y como se explicó en la sección 11.4.

1. Incluya la estrategia de pruebas de no regresión explicada en esa sección.
2. Incluya pruebas para las extensiones realizadas: comentarios, cadenas y números en punto flotante.
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

```
    elsif (/\G\s*(.)/gc) {
      Error::fatal "Caracter invalido: $1\n";
    }
```

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 utilice `eval` y la variable especial `$@` para controlar que el programa `.t` no termine prematuramente. Repase la sección 3.10.3, el capítulo 6.16 y mas específicamente la sección 6.16.7 del capítulo sobre construcción de módulos. Además lea el capítulo Test Now, test Forever ” del libro de Scott [12].

11.4.3. Práctica: Mas Pruebas y Extensiones al Análizador Léxico

Extienda la práctica con las siguientes características:

1. Cuando la entrada al compilador tiene blancos al final:

```
PL-Tutu/scripts$ cat test02.tutu
int a,b;
a = 4



```

   El analizador léxico produce una salida errónea:

```
PL-Tutu/scripts$ tutu test02.tutu test01.sal
Error: Caracter invalido:

 at ../lib/PL/Lexical/Analysis.pm line 19
```

   Corriga el fallo y convierta el fallo en una prueba.

2. Mejore la práctica anterior para que distinga entre el terminal `NUMINT` y el terminal `NUMFLOAT`. Observe que la expresión regular del segundo contiene a la del primero. Por supuesto es un caso TIMTOWTDI. Por ejemplo, una manera es: una vez capturado un `NUMFLOAT` podemos saber si se trata de un `NUMINT` atendiendo al índice del último paréntesis que casó (véase sección 4.3).

3. Modularize las diferentes fases del compilador (tal y como se explicaron en la sección 11.2) e introduzca pruebas `TODO` usando `can_ok` para las funciones que están por escribir (`parser`, `Optimize`, `code_generator`, `transform`). Repáse la sección 6.16.7.

4. 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`.

Introduzca una prueba `SKIP`: Si existen ficheros `skip.tutu` y `skip.sal` en el directorio `t` realize una prueba que compile `skip.tutu` y compruebe que la salida coincide con la de `skip.sal`. Si los ficheros no existen la prueba no se ejecuta.

Lea `perldoc Test::Tutorial` para estudiar el tutorial de Michael Schwern sobre pruebas.

5. Use el módulo `Test::Pod` para introducir un test que compruebe que la documentación esta bien escrita. Estudie la documentación del módulo `Test::Pod`.

6. Para mejorar la calidad de los mensajes de error extienda el par (terminal, valor) devuelto por el `scanner` a una terna (terminal, valor, número de línea) cuya tercera componente es el número de línea en el que aparece el terminal.

   Hay muchas formas de hacerlo. Una posibilidad es apoyarse en el operador `tr` (descrito en la sección 4.27) el cuál ademas de reemplazar devuelve el número de carácteres reeemplazados o suprimidos:

   ```
   $cuenta = $texto =~ tr/\n/\n/; # cuenta los retornos de carro en texto
   ```

   Modifique las pruebas para que se adapten a la nueva interfaz de `scanner`.

11.4.4. 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 `$+`?
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`?

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{^"([^"]|\\")*"}
"Focho \"
  DB<4> print $& if $a =~ m{^"(\\"||[^"])*"}
"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)
```

11.5. 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 11.5.1. *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 11.5.2. *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 $A \rightarrow \alpha$. Un elemento de P se denomina producción. Por último, S es un símbolo del conjunto V que se denomina símbolo de arranque.*

Definición 11.5.3. *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 $A \rightarrow \gamma$ 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 $\mu \stackrel{*}{\Longrightarrow} \delta$. Una cadena deriva en 0 pasos en si misma.*

Definición 11.5.4. *Dada una gramática $G = (\Sigma, V, P, S)$ se denota por $L(G)$ o lenguaje generado por G al lenguaje:*

$$L(G) = \{x \in \Sigma^* : S \stackrel{*}{\Longrightarrow} x\}$$

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 11.5.5. *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 $\mu \stackrel{*}{\Longrightarrow} \delta$ en la cual en cada paso αAx la regla de producción aplicada $A \rightarrow \gamma$ se aplica en la variable sintáctica mas a la derecha se dice una derivación a derechas

Una derivación $\mu \stackrel{*}{\Longrightarrow} \delta$ en la cual en cada paso $xA\alpha$ la regla de producción aplicada $A \rightarrow \gamma$ se aplica en la variable sintáctica mas a la izquierda se dice una derivación a izquierdas

Definición 11.5.6. *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 $A \rightarrow \gamma$ se traduce en asignar como hijos del nodo etiquetado con A a los nodos etiquetados con los símbolos $X_1 \dots X_n$ que constituyen la frase $\gamma = X_1 \dots X_n$.*

Definición 11.5.7. *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 $x_1 \dots x_n$ 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 11.5.8. *Una gramática G se dice ambigua si existe alguna frase $x \in L(G)$ con al menos dos árboles sintácticos.*

11.5.1. 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.

11.6. 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 usaremos aqui es uno de los mas sencillos: se denomina método de análisis predictivo descendente recursivo.

11.6.1. 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 un analizador predictivo descendente recursivo (APDR) el símbolo que actualmente esta siendo observado (denotado `lookahead`) permite determinar unívocamente que producción de A hay que aplicar. Para ilustrar el método, simplificaremos la gramática presentada en el ejercicio 11.5.1 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.

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 $A \rightarrow \alpha$ se aplica para a continuación comprobar que la entrada que sigue pertenece al lenguaje generado por α. Para ello, las apariciones de terminales en α son emparejadas con los terminales en la entrada y las apariciones de variables B en α se traducen en llamadas a la correspondiente subrutina asociada `B`.

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 la sección 5.15.

Programa 11.6.1.
```
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"); }
}

sub statements {
  statement;
  if ($lookahead eq ';') { match(';'); statements; }
}

sub parser {
  ($lookahead, $value) = splice @tokens,0,2;
  statements; match('EOI');
}
```

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 11.6.1. *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:

$$N(\alpha) = \begin{cases} \{\epsilon\} & si\ \alpha \stackrel{*}{\Longrightarrow} \epsilon \\ \emptyset & en\ otro\ caso \end{cases}$$

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

11.6.2. 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?

11.6.3. Ejercicio: Factores Comunes

En el programa 11.6.1 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?

11.6.4. 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 $A \to \gamma$ 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 11.6.2. *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

$$E(A) = \begin{cases} \{\$\} & si\ S \stackrel{*}{\Longrightarrow} \alpha A \\ \emptyset & en\ otro\ caso \end{cases}$$

Aqui \$ denota el final de la entrada (que se corresponde en el código Perl anterior con el terminal `EOI`).

Si $A \to \gamma_1 \mid \ldots \mid \gamma_n$ 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.

11.6.5. Construcción de los conjuntos de Primeros y Siguientes

Algoritmo 11.6.1. *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 $X \to \epsilon$ 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*

$$\begin{array}{l} i = 1; \\ hacer \\ \quad FIRST(X) = FIRST(X) \cup FIRST^*(Y_i); \\ \quad i++; \\ mientras\ (i \leq k\ y\ \epsilon \in FIRST(Y_i)) \end{array}$$

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 11.6.2. *Construcción del conjunto* $FIRST(\alpha)$

Repita siguiente conjunto de reglas hasta que no se puedan añadir mas símbolos terminales o a ningún conjunto $FIRST(\alpha)$*:*

$$
\begin{array}{l}
i = 1; \\
FIRST(\alpha) = \emptyset; \\
hacer \\
\quad FIRST(\alpha) = FIRST(\alpha) \cup FIRST^*(X_i); \\
\quad i++; \\
mientras\ (i \leq n\ y\ \epsilon \in FIRST(X_i))
\end{array}
$$

Algoritmo 11.6.3. *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* $A \to \alpha B \beta$ *entonces*

$$FOLLOW(B) = FOLLOW(B) \cup (FIRST(\beta) - \{\epsilon\})$$

3. *Si* $A \to \alpha B$ *o* $A \to \alpha B \beta$ *y* $\epsilon \in FIRST(\beta)$ *entonces*

$$FOLLOW(B) = FOLLOW(B) \cup FOLLOW(A)$$

11.6.6. Ejercicio: Construir los $FIRST$

Construya los conjuntos $FIRST$ de las partes derechas de las reglas de producción de la gramática presentada en el ejercicio 11.5.1.

11.6.7. Ejercicio: Calcular los $FOLLOW$

Modificamos la gramática de la sección 11.6.1 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.

11.6.8. 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 `yapp` 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 $/ = undef;
open FILE, "$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
 1  %%
 2  S:
 3      |   'a' S 'b'
 4  ;
 5  %%
```

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`.

11.6.9. 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 $A \to \alpha$ y $A \to \beta$ 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.

11.6.10. 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 $FOLLOW(A) = \emptyset$?

11.6.11. Ejercicio: Ambiguedad y LL(1)

¿Puede una gramática LL(1) ser ambigua?. Razone su respuesta.

11.6.12. 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 (11.3 y 11.4).

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

11.6.13. Práctica: Generación Automática de Analizadores Predictivos

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. Dicha estructura de datos se explicó en la práctica 11.6.8. La estructura de datos generada por `Grammar::Parse` 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 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 }, 'NULL' => { 'S' => 1 },
  'RULES' => [
               [ 'S', [] ],
               [ 'S', [ '\'a\'', 'S', '\'b\'' ] ]
             ],
  'START' => 'S',
  'TERM' => [ '\'b\'', '\'a\'' ], 'NTERM' => { 'S' => [ 0, 1 ] }
  'TAIL' => [ '

  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 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`. 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. 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.pm;

&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.

Para la escritura de `GAP.pm` haga uso del módulo que calcula los $FIRST$ y los $FOLLOW$ que desarrolló durante la práctica 11.6.8.

11.7. Esquemas de Traducción

Definición 11.7.1. *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 $A \to \alpha\beta$ insertamos un fragmento de código:

$$A \to \alpha\{action\}\beta$$

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:

$expr \to expr_1 - NUM$ `{ $expr{TRA} = $expr_1{TRA}." ".$NUM{VAL}." - "}`
$expr \to NUM$ `{ $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 11.7.1. *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 11.7.2. *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 11.7.1. *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} }`

Definición 11.7.3. *Un* atributo heredado *es aquel cuyo valor se computa a partir de los valores de sus hermanos y de su padre.*

Ejercicio 11.7.2. *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.*

11.8. Recursión por la Izquierda

Definición 11.8.1. *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.

11.8.1. 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:

$$\begin{array}{l} A \rightarrow \beta R \\ R \rightarrow \alpha R \quad | \; \epsilon \end{array}$$

eliminando así la recursión por la izquierda.

Definición 11.8.2. *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 11.8.1. *Elimine la recursión por la izquierda de la gramática*

$$\begin{array}{l} expr \rightarrow expr - NUM \\ expr \rightarrow NUM \end{array}$$

Ejercicio 11.8.2. *¿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 11.8.3. *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} = "" }`

11.8.2. 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:

$A \rightarrow A\alpha$ `{ alpha_action }`
$A \rightarrow A\beta$ `{ beta_action }`
$A \rightarrow \gamma$ `{ 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:

$A \rightarrow \gamma$ `{ gamma_action }` R
$R \rightarrow \beta$ `{ beta_action }` R
$R \rightarrow \alpha$ `{ alpha_action }` R
$R \rightarrow \epsilon$

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:

$expr \rightarrow expr_1 - NUM$ `{ $expr{T} = $expr_1{T}." ".$NUM{VAL}." - "}`
$expr \rightarrow NUM$ `{ $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:

$expr \rightarrow NUM$ `{ $r{H} = $NUM{VAL} }` r `{ $expr{T} = $r{T} }`
$r \rightarrow -NUM$ `{ $r_1{H} = $r{H}." ".$NUM{VAL}." - " }` r_1 `{ $r{T} = $r_1{T} }`
$r \rightarrow \epsilon$ `{ $r{T} = $r{H} }`

El atributo `H` es un ejemplo de atributo heredado.

11.8.3. Ejercicio

Calcule los valores de los atributos cuando se aplica el esquema de traducción anterior a la frase `4 - 5 - 7`.

11.8.4. 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:

$expr \rightarrow NUM$ `{ $r{H} = $NUM{VAL} }` r `{ $expr{T} = $r{T} }`
$r \rightarrow -NUM$ `{ $r_1{H} = $r{H}." ".$NUM{VAL}." - " }` r_1 `{ $r{T} = $r_1{T} }`
$r \rightarrow \epsilon$ `{ $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} }
}
```

11.8.5. 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.

11.8.6. 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 11.5.1 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 11.4.1. ¡Note que el árbol que debe producir es el de la gramática inicial, ¡No el de la gramática transformada!

11.9. Árbol de Análisis Abstracto

11.9.1. 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 11.9.1. *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 11.9.1. *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 11.9.2. *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 11.9.2. *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 11.9.3. *Dada una gramática (Σ, N, P, S), se dice que un árbol $t \in B(\Sigma \cup N)$ es del tipo $(X_1, \dots X_k)$ 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 $(X_1, \dots X_n)$, diremos que la producción p es de tipo $(X_1, \dots X_n) \to X$.

Definición 11.9.4. *Consideremos un árbol $t \in B(\Sigma \cup N)$ que sea del tipo $(X_1, \dots X_n)$, esto es las variables sintácticas en el árbol leídas de izquierda a derecha son $(X_1, \dots X_n)$.*

- *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 $t \stackrel{0}{\Longrightarrow} t$.*
- *Decimos que un árbol t deriva en n pasos en el árbol t' y escribimos $t \stackrel{n}{\Longrightarrow} t'$ 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 $t \stackrel{*}{\Longrightarrow} t'$.*

Definición 11.9.5. *Se define el* lenguaje árbol generado por una gramática $G = (\Sigma, N, P, S)$ *como el lenguaje $L(G) = \{t \in B(\Sigma) : \exists S \stackrel{*}{\Longrightarrow} t\}$.*

Ejemplo 11.9.3. *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:

$$L(G) = \{NIL, CONS(A, NIL), CONS(A, CONS(A, NIL)), \dots\}$$

Ejercicio 11.9.1. *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 11.9.4. *Sea $G = (\Sigma, V, P, S)$ con*

$$\Sigma = \{ID, NUM, LEFTVALUE, STR, PLUS, TIMES, ASSIGN, PRINT\}$$
$$\rho(ID) = \rho(NUM) = \rho(LEFTVALUE) = \rho(STR) = 0$$
$$\rho(PRINT) = 1$$
$$\rho(PLUS) = \rho(TIMES) = \rho(ASSIGN) = 2$$
$$V = \{st, expr\}$$

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 11.9.2. *Redefina el concepto de árbol de análisis concreto dado en la definición 11.5.7 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 11.9.3. *¿Cómo son los árboles sintácticos en las derivaciones árbol? Dibuje varios árboles sintácticos para las gramáticas introducidas en los ejemplos 11.9.3 y 11.9.4.*

Intente dar una definición formal del concepto de árbol de análisis sintáctico asociado con una derivación en una gramática árbol

Definición 11.9.6. *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 11.9.4. *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 $t/2.1.2$.

11.9.2. Realización del AAA para Tutu en Perl

En la sección 11.6.1 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 11.9.7. *La gramática árbol extendida que especifica los árboles AAA para el compilador de Tutu es esta:*

$$
\begin{array}{r|ll}
1 & prog & \rightarrow PROGRAM(decls, sts) \\
2 & decls & \rightarrow list\ decl \\
3 & sts & \rightarrow list\ st \\
4 & decl & \rightarrow INT(idlist) \\
5 & decl & \rightarrow STRING(idlist) \\
6 & idlist & \rightarrow list\ SIMPLEID \\
7 & st & \rightarrow ASSIGN(LEFTVALUE, expr) \\
8 & st & \rightarrow PRINT(expr) \\
9 & expr & \rightarrow PLUS(expr, expr) \\
10 & expr & \rightarrow TIMES(expr, expr) \\
11 & expr & \rightarrow NUM \\
12 & expr & \rightarrow ID \\
13 & expr & \rightarrow STR
\end{array}
$$

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
```

```
require Exporter;
our @ISA = qw(Exporter);
our @EXPORT = qw( );
our $VERSION = '0.02';

#######################################################

# Grammar:
# P : DD L      | L
# DD: D ';' DD  | D ';'
# D : int I     | string I
# L : S         | S ; L
# S : ID '=' E  | p E  | epsilon
# E : T '+' E   | T
# T : F '*' T   | F
# F : '(' E ')' | id | num | str
# 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
use 5.008004;
use ExtUtils::MakeMaker;
WriteMakefile(
    NAME              => 'PL::Tutu',
    VERSION_FROM      => 'lib/PL/Tutu.pm', # finds $VERSION
    PREREQ_PM         => {Class::MakeMethods::Emulator::MethodMaker => 0},1
    EXE_FILES         => [ 'scripts/tutu.pl', 'scripts/tutu' ],
    ($] >= 5.005 ?     ## Add these new keywords supported since 5.005
      (ABSTRACT_FROM  => 'lib/PL/Tutu.pm', # retrieve abstract from module
       AUTHOR         => 'Casiano Rodriguez Leon <casiano@ull.es>') : ()),
);
```

Se actualiza también MANIFEST:

```
$ cat -n MANIFEST
Changes
lib/PL/Error.pm
lib/PL/Lexical/Analysis.pm
lib/PL/Syntax/Analysis.pm
lib/PL/Tutu.pm
Makefile.PL
MANIFEST
MANIFEST.SKIP
README
scripts/test01.tutu
scripts/tutu
scripts/tutu.pl
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
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 ($ASSIGN$, $PRINT$, $\ldots$) 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");
  }
}
```

11.9.3. 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 $\rightarrow$ *list decl*	declarations $\rightarrow$ declaration ';' declarations | declaration ';'
sts $\rightarrow$ *list st*	statements $\rightarrow$ statement ';' statements | statement
idlist $\rightarrow$ *list SIMPLEID*	idlist $\rightarrow$ 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;
}
```

11.9.4. 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:

$$
\begin{array}{r|ll}
2 & decls & \rightarrow \text{list } decl \\
4 & decl & \rightarrow INT(idlist) \\
5 & decl & \rightarrow STRING(idlist) \\
6 & idlist & \rightarrow \text{list } SIMPLEID
\end{array}
$$

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.

11.9.5. 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 11.9.7. 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

11.10. 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};
}
```

11.10.1. 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?

11.10.2. Práctica: Análisis Semántico

Extienda el código de la práctica 11.9.5 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.

11.11. 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;
  }
}
```

11.11.1. 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 interfase 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 la sección 7.6.9 de estos apuntes para saber más sobre clases abstractas.

11.12. Patrones Árbol y Transformaciones Árbol

En la fase de optimización presentada en la sección 11.11 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 11.9.1 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 11.12.1. *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 11.12.2. *Se dice que un patrón es un* patrón lineal *si ninguna variable se repite.*

Definición 11.12.3. *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 11.12.1. *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]

[1] Repase la notación de Dewey introducida en la definición 11.9.6

Ejercicio 11.12.1. *Dado la gramática árbol:*

$$S \rightarrow S_1(a, S, b)$$
$$S \rightarrow S_2(NIL)$$

la cuál genera los árboles concretos para la gramática

$$S \rightarrow aSb \mid \epsilon$$

¿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 11.12.2. *Ejemplos de patrones para el AAA definido en el ejemplo 11.9.2 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 11.12.4. *Una* sustitución *es una aplicación θ que asigna variables a patrones $\theta : V \rightarrow B(V \cup \Sigma)$.*

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.

$$\theta : B(V \cup \Sigma) \rightarrow B(V \cup \Sigma)$$
$$t\theta = \begin{cases} x\theta & si\ t = x \in V \\ a(t_1\theta, \ldots, t_k\theta) & si\ t = a(t_1, \ldots, t_k) \end{cases}$$

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 11.12.3. *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 11.12.4. *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 11.12.5. *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 11.12.5. *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 $t.\,1\{x/A\}$.*

$$t = \tau\{x/CONS(A, NIL)\}$$
$$t.\,1 = \tau\{x/A\}$$

Ejercicio 11.12.2. *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 11.11 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.

11.12.1. 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 11.12.

Ademas del plegado de constantes use las nuevas subrutinas para aplicar simultáneamente las siguientes transformaciones algebraicas:

$PLUS(NUM, x)$	$\wedge$ `{ $NUM{VAL} == 0 }`	$\Longrightarrow x$
$PLUS(x, NUM)$	$\wedge$ `{ $NUM{VAL} == 0 }`	$\Longrightarrow x$
$TIMES(NUM, x)$	$\wedge$ `{ $NUM{VAL} == 1 }`	$\Longrightarrow x$
$TIMES(x, NUM)$	$\wedge$ `{ $NUM{VAL} == 1 }`	$\Longrightarrow x$

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?

11.13. 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 11.13.1.

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 11.13.1 propone una mejora.

11.13.1. 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 11.10.2.

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"
#              012345678901234567890123456789012
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).

11.14. 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
```

11.15. Generación de Código: Máquina Basada en Registros

La máquina orientada a pila para la que generamos código en la sección 11.14 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 $R_0 \dots R_{k-1}$. 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 11.15.1. *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 11.15.2. *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?.*

11.15.1. 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 $\text{máx}\{2, r_t\}$ 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.

11.16. 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).

11.16.1. 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 4.23. 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 11.11) 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 12

Construcción de Analizadores Léxicos

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 el emparejamiento. Sin embargo, en el análisis léxico, el problema es encontrar la subcadena a partir de la última posición "casada" que casa con una de las expresiones regulares que definen los lexemas del lenguaje dado.

12.1. Encontrando los terminales mediante sustitución

En esta sección construiremos una clase que provee una familia sencilla de analizadores léxicos. El constructor de la clase recibe como entrada la familia de parejas (expresión-regular, identificador-de-terminal) y devuelve como resultado una referencia a una subrutina, dinámicamente creada, que hace el análisis léxico. Esto es, se devuelve una subrutina que cuando se la llama con una cadena de entrada que se le pasa como parámetro devuelve la siguiente pareja (cadena, identificador-de-terminal).

Por ejemplo, después de la llamada:

```
my $lex = Lexer->new('\s*', #espacios
                     '\d+(\.\d+)?'=>'NUMBER',
                     '#.*'=>'COMMENT',
                     '"[^"]*"'=>'STRING',
                     '\$[a-zA-Z_]\w*'=>'VAR',
                     '[a-zA-Z_]\w*'=>'ID',
                     '.'=>'OTHER');
```

la variable `$lex` contendrá una referencia a una subrutina que se creará dinámicamente y cuyo código es similar al siguiente:

```
sub {
  $_[0] =~ s/\A\s*//;
  $_[0] =~ s/\A(\d+(\.\d+)?)// and return ("$1", 'NUMBER');
  $_[0] =~ s/\A(#.*)// and return ("$1", 'COMMENT');
  $_[0] =~ s/\A("[^"]*")// and return ("$1", 'STRING');
  $_[0] =~ s/\A(\$[a-zA-Z_]\w*)// and return ("$1", 'VAR');
  $_[0] =~ s/\A([a-zA-Z_]\w*)// and return ("$1", 'ID');
  $_[0] =~ s/\A(.)// and return ("$1", 'OTHER');
  $_[0] =~ s/(.|\n)// and return ("$1", ''); # 'tragamos' el resto
  return; # la cadena es vacía
}
```

Recordemos que el ancla `\A` casa con el comienzo de la cadena, incluso si esta es multilínea. Esta metodología es descrita por Conway en [14]. Obsérvese que una vez mas, Perl se aparta de la ortodoxia:

en otros lenguajes un objeto suele adoptar la forma de un `struct` o hash. En Perl cualquier cosa que se bendiga puede ser promocionada a la categoría de objeto. En este caso es una subrutina.

Veamos primero el código del constructor, situado en el módulo `Lexer.pm`:

```
package Lexer;
$VERSION = 1.00;
use strict;

sub new {
  my $class = shift;  # El nombre del paquete
  my $spaces = shift; # Los espacios en blanco a "tragar"
  my $end_sub =<<EOS;
  \$_[0] =~ s/(.|\\n)// and return ("\$1", ''); # 'tragamos' el resto
  return; # la cadena es vacía
}
EOS
  my $code = "sub {\n  \$_[0] =~ s/\\A$spaces//;\n";

  while (my ($regexp, $token) = splice @_, 0, 2) {
    my $lexer_line =<<EOL;
  \$_[0] =~ s/\\A($regexp)// and return ("\$1", '$token');
EOL

    $code .= $lexer_line;
  }
  $code .= $end_sub;

  my $sub = eval $code or die "Error en la definición de los terminales\n";
  bless $sub, $class;
}
```

Obsérvese que, al bendecirla, estamos elevando la subrutina `$sub` a la categoría de objeto, facilitando de este modo

- la coexistencia de diversos analizadores léxicos en el mismo programa,
- el adjudicarle métodos al objeto y
- el uso de la extensión de la sintáxis flecha para objetos.

Así podemos tener diversos métodos asociados con el analizador léxico que modifiquen su conducta por defecto. Por ejemplo, podemos disponer de un método `extract_next` que recibiendo como entrada la cadena a analizar elimine de la misma el siguiente terminal:

```
($val, $token) =  $lex->extract_next($input);
```

El código de dicho método será:

```
sub extract_next {
  &{$_[0]}($_[1]); # $lex->extract_next($data) <=> $lex($data)
}
```

pero es posible tener también un método `lookahead` que devuelva la misma información sin eliminar el terminal (aunque en el siguiente código si que se eliminan los blancos iniciales) de la cadena de entrada:

```
sub lookahead {
  my ($val, $token) = &{$_[0]}($_[1]);
  $_[1] = $val.$_[1];
  return ($val, $token);
}
```

y asi, cuantos métodos hagan falta. Por ejemplo, podemos introducir un método `extract_all` que produzca la lista completa de parejas (cadena, identificador-de-terminal) sin destruir la entrada:

```
sub extract_all {
  my ($self, $input) = @_; # no destructivo
  my @tokens = ();

  while ($input) {
    push @tokens, &{$self}($input);
  }

  return @tokens;
}
```

Veamos un ejemplo de uso. El programa `uselexer.pl` hace uso del package/clase, creando primero un objeto de la clase `Lexer` y utilizando después dos de sus métodos:

```
#!/usr/bin/perl -w
use Lexer;

my ($val, $token);
my $lex = Lexer->new('\s*', #espacios
                     '\d+(\.\d+)?'=>'NUMBER',
                     '#.*'=>'COMMENT',
                     '"[^"]*"'=>'STRING',
                     '\$[a-zA-Z_]\w*'=>'VAR',
                     '[a-zA-Z_]\w*'=>'ID',
                     '.'=>'OTHER');
undef($/);

#token a token; lectura destructiva
my $input = <>;
while (($val, $token) =  $lex->extract_next($input)) {
  print "$token -> $val\n";
}

# todos a la vez
$input = <>;
my @tokens =  $lex->extract_all($input);
print "@tokens\n";
```

Ejemplo de ejecución:

```
$ ./uselexer.pl
$a = 34; # comentario
```

pulsamos `CTRl-D` y obtenemos la salida:

```
VAR -> $a
OTHER -> =
NUMBER -> 34
```

```
OTHER -> ;
COMMENT -> # comentario
```

De nuevo damos la misma entrada, pero ahora utilizaremos el método no destructivo `extract_all`:

```
$a = 34; #comentario
```

pulsamos `CTRl-D` de nuevo y obtenemos la salida:

```
$a VAR = OTHER 34 NUMBER ; OTHER #comentario COMMENT
```

12.2. Construcción usando la opción `g` y el ancla `G`

Tomemos como ejemplo el análisis léxico de una sencilla calculadora que admite constantes numéricas enteras, los paréntesis `(`, `)` y los operadores `+`, `-`, `*` y `/`. Se trata de producir a partir de la cadena de entrada una lista con los *tokens* y sus valores. Por ejemplo, la cadena `3*4-2` generará la lista: `('num', 3, 'op', '*', 'num', 4, 'op', '-', 'num', 2)`.

Las versiones mas recientes de Perl disponen de una opción `/c` que afecta a las operaciones de emparejamiento con `/g` en el contexto escalar. Normalmente, cuando una búsqueda global escalar tiene lugar y no ocurre casamiento, la posición `\G` 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. combinando `\G` y `/c` es posible escribir con cierta comodidad un analizador léxico:

```
{ # Con el redo del final hacemos un bucle "infinito"
  if (\G(\d+)/gc) {
    push @tokens, 'num', $1;
  } elsif (m|\G([-+*()/])|gc) {
    push @tokens, 'pun', $1;
  } elsif (/\G./gc) {
    die 'Caracter invalido';
  }
  else {
    last;
  }
  redo;
}
```

12.3. La clase `Parse::Lex`

La clase `Parse::Lex` nos permite crear un analizador léxico. La estrategia seguida es mover el puntero de búsqueda dentro de la cadena a analizar utilizando conjuntamente el operador `pos()` y el ancla `\G`.

```
> cat -n tokenizer.pl
#!/usr/local/bin/perl

require 5.004;
#BEGIN {  unshift @INC, "../lib"; }

$^W = 0;
use Parse::Lex;
print STDERR "Version $Parse::ALex::VERSION\n";

@token = (
          qw(
```

```
12              ADDOP    [-+]
13              LEFTP    [\(]
14              RIGHTP   [\)]
15              INTEGER  [1-9][0-9]*
16              NEWLINE  \n
17             ),
18           qw(STRING),   [qw(" (?:[^"]+|"")* ")],
19           qw(ERROR  .*), sub {
20             die qq!can\'t analyze: "$_[1]"\n!;
21           }
22          );
23
24  Parse::Lex->trace;
25  $lexer = Parse::Lex->new(@token);
26
27  $lexer->from(\*DATA);
28  print "Tokenization of DATA:\n";
29
30  TOKEN:while (1) {
31    $token = $lexer->next;
32    if (not $lexer->eoi) {
33      print "Record number: ", $lexer->line, "\n";
34      print "Type: ", $token->name, "\t";
35      print "Content:->", $token->text, "<-\n";
36    } else {
37      last TOKEN;
38    }
39  }
40
41  __END__
42  1+2-5
43  "This is a multiline
44  string with an embedded "" in it"
45  this is an invalid string with a "" in it"
```

La línea 3 usa la palabra reservada `require` la cual normalmente indica la existencia de una dependencia. Cuando se usa como:

```
require "file.pl";
```

simplemente carga el fichero en nuestro programa. Si se omite el sufijo, se asume `.pm`. Si el argumento, como es el caso, es un número o un número de versión, se compara el número de versión de perl (como se guarda en `$]`) y se compara con el argumento. si es mas pequeño, se aborta la ejecución.

La línea 4, aunque comentada, contiene una definición del incializador `BEGIN`. Si aparece, este código será ejecutado tan pronto como sea posible. el intérprete Perl busca los módulos en uno de varios directorios estándar contenidos en la variable de entorno `PERL5LIB`. Esa lista esta disponible en un programa Perl a través de la variable `@INC`. Recuerda que el compilador sustituye cada `::` por el separador de caminos. Asi la orden: `unshift @INC, "../lib";` coloca este directorio entre los que Perl realiza la búsqueda por módulos. En mi máquina los mósulos de análisis léxico están en:

```
> locate Lex.pm
/usr/local/lib/perl5/site_perl/5.8.0/Parse/ALex.pm
/usr/local/lib/perl5/site_perl/5.8.0/Parse/CLex.pm
/usr/local/lib/perl5/site_perl/5.8.0/Parse/Lex.pm
/usr/local/lib/perl5/site_perl/5.8.0/Parse/YYLex.pm
```

La línea 6 establece a 0 la variable `$^W` o `$WARNING` la cual guarda el valora actual de la opción de *warning*, si es cierto o falso.

La línea 7 indica que hacemos uso de la clase `Parse::Lex` y la línea 8 muestra el número de versión. La clase `ALex` es la clase abstracta desde la que heredan las otras. Si editas el módulo `ALex.pm` encontrarás el siguiente comentario que aclara la jerarquía de clases:

```
# Architecture:
#           Parse::ALex - Abstract Lexer
#                |
#           +----------+
#           |          |
#           |     Parse::Tokenizer
#           |        |     |
#        LexEvent   Lex  CLex  ...       - Concrete lexers
```

en ese mismo fichero encontrarás la declaración de la variable:

```
$Parse::ALex::VERSION = '2.15';
```

Las líneas de la 10 a la 22 definen el *array* `@token`:

```
10  @token = (
11           qw(
12              ADDOP    [-+]
13              LEFTP    [\(]
14              RIGHTP   [\)]
15              INTEGER  [1-9][0-9]*
16              NEWLINE  \n
17             ),
18           qw(STRING),   [qw(" (?:[^"]+|"")* ")],
19           qw(ERROR  .*), sub {
20             die qq!can\'t analyze: "$_[1]"\n!;
21           }
22          );
```

Esto da lugar a un *array* de la forma:

```
"ADDOP", "[-+]",
"LEFTP", "[(]",
"RIGHTP", "[)]",
"INTEGER", "[1-9][0-9]*",
"NEWLINE", "\n",
"STRING", ARRAY(0x811f55c),
"ERROR", ".*", CODE(0x82534b8)
```

Observa que los elementos 11 y 14 son referencias. El primero a un *array* anónimo conteniendo la expresión regular `"(?:[^"]+)*`"— para las `STRING` y el segundo es una referencia a la subrutina anónima que muestra el mensaje de error.

La línea 24 llama al método `trace` como método de la clase, el cual activa el modo "traza". La activación del modo traza debe establecerse antes de la creación del analizador léxico. Es por esto que precede a la llamada `$lexer = Parse::Lex->new(@token)` en la línea 25. Se puede desactivar el modo traza sin mas que hacer una segunda llamada a este método. El método admite la forma `trace OUTPUT`, siendo `OUTPUT` un nombre de fichero o un manipulador ed ficheros. En este caso la traza se redirije a dicho fichero.

La llamada al método `from` en la línea 27 establece la fuente de los datos a ser analizados. el argumento de este método puede ser una cadena o una lista de cadenas o una referencia a un manipulador de fichero. En este caso la llamada `$lexer->from(\*DATA)` establece que la entrada se produce dede el manipulador de ficheros especial `DATA`, el cual se refiere a todo el texto que sigue al *token* `__END__` en el fichero que contiene el guión Perl.

La llamada al método `next` en la línea 31 fuerza la búsqueda por el siguiente *token*. Devuelve un objeto de la clase `Parse::Token`. Existe un *token* especial, `Token::EOI`— que indica el final de los datos. En el ejemplo, el final se detecta a través del método `eoi`.

El método `line` es usado para devolver el número de línea del registro actual.

Los objetos *token* son definidos mediante la clase `Parse::Token`, la cual viene con `Parse::Lex`. la definición de un *token* normalmente conlleva un nombre simbólico (por ejemplo, `INTEGER`), seguido de una expresión regular. Se puede dar como tercer argumento una referencia a una subrutina anónima, en cuyo caso la subrutina es llamada cada vez que se reconoce el *token*. Los argumentos que recibe la subrutina son la referencia al objeto *token* y la cadena reconocida por la expresión regular. El valor retornado por la subrutina es usado como los nuevos contenidos de la cadena asociada con el objeto *token* en cuestión. Un objeto *token* tiene diversos métodos asociados. Asi el método `name` devuelve el nombre del *token*. El método `text` devuelve lac adena de caracteres reconocida por medio del *token*. Se le puede pasar un parámetro `text EXPR` en cuyo caso `EXPR` define la cadena de caracteres asociada con el lexema.

```
> tokenizer.pl
Version 2.15
Trace is ON in class Parse::Lex
Tokenization of DATA:
[main::lexer|Parse::Lex] Token read (INTEGER, [1-9][0-9]*): 1
Record number: 1
Type: INTEGER   Content:->1<-
[main::lexer|Parse::Lex] Token read (ADDOP, [-+]): +
Record number: 1
Type: ADDOP     Content:->+<-
[main::lexer|Parse::Lex] Token read (INTEGER, [1-9][0-9]*): 2
Record number: 1
Type: INTEGER   Content:->2<-
[main::lexer|Parse::Lex] Token read (ADDOP, [-+]): -
Record number: 1
Type: ADDOP     Content:->-<-
[main::lexer|Parse::Lex] Token read (INTEGER, [1-9][0-9]*): 5
Record number: 1
Type: INTEGER   Content:->5<-
[main::lexer|Parse::Lex] Token read (NEWLINE, \n):

Record number: 1
Type: NEWLINE   Content:->
<-
[main::lexer|Parse::Lex] Token read (STRING, \"(?:[^\"]+|\"\")*\"): "This is a multiline
string with an embedded "" in it"
Record number: 3
Type: STRING    Content:->"This is a multiline
string with an embedded "" in it"<-
[main::lexer|Parse::Lex] Token read (NEWLINE, \n):

Record number: 3
Type: NEWLINE   Content:->
<-
[main::lexer|Parse::Lex] Token read (ERROR, .*): this is an invalid string with a "" in it"
can't analyze: "this is an invalid string with a "" in it""
```

12.3.1. Condiciones de arranque

Los *tokens* pueden ser prefijados por “condiciones de arranque” o “estados”. Por ejemplo:

```
qw(C1:T1 er1), sub { # acción },
qw(T2 er2), sub { # acción },
```

el símbolo `T1` será reconocido únicamente si el estado `T1` está activo. Las condiciones de arranque se activan utilizando el método `start(condicion)` y se desactivan mediante `end(condicion)`. La llamada `start('INITIAL')` nos devuelve a la condición inicial.

```
> cat -n expectfloats.pl
#!/usr/bin/perl -w
 use Parse::Lex;
 @token = (
          'EXPECT', 'expect-floats', sub {
            $lexer->start('expect');
            $_[1]
          },
          'expect:FLOAT', '\d+\.\d+',
          'expect:NEWLINE', '\n', sub { $lexer->end('expect') ; $_[1] },
          'expect:SIGN','[+-]',
          'NEWLINE2', '\n',
          'INT', '\d+',
          'DOT', '\.',
          'SIGN2','[+-]'
         );

 Parse::Lex->exclusive('expect');
 $lexer = Parse::Lex->new(@token);

 $lexer->from(\*DATA);

 TOKEN:while (1) {
   $token = $lexer->next;
   if (not $lexer->eoi) {
     print $token->name," ";
     print "\n" if ($token->text eq "\n");
   } else {
     last TOKEN;
   }
 }

__END__
1.4+2-5
expect-floats 1.4+2.3-5.9
1.5
```

Ejecución:

```
> expectfloats.pl
INT DOT INT SIGN2 INT SIGN2 INT NEWLINE2
EXPECT FLOAT SIGN FLOAT SIGN FLOAT NEWLINE
INT DOT INT NEWLINE2
```

El siguiente ejemplo elimina los comentarios anidados en un programa C:

```
> cat -n nestedcom.pl
#!/usr/local/bin/perl  -w

require 5.004;
#BEGIN {  unshift @INC, "../lib"; }

$^W = 0;
use Parse::Lex;

@token = (
             'STRING',         '"([^\"]|\\.)*"', sub { print "$_[1]"; $_[1]; },
             'CI',             '\/\*', sub { $lexer->start('comment'); $c++; $_[1]; },
             'CF',             '\*\/', sub { die "Error, comentario no finalizado!"; },
             'OTHER',          '(.|\n)', sub { print "$_[1]"; $_[1]; },
             'comment:CCI',    '\/\*', sub { $c++; $_[1]; },
             'comment:CCF',    '\*\/', sub { $c--; $lexer->end('comment') if ($c == 0); $_[1]; },
             'comment:COTHER',       '(.|\n)'
         );

 #Parse::Lex->trace;
 Parse::Lex->exclusive('comment');
 Parse::Lex->skip('');

 $lexer = Parse::Lex->new(@token);

 $lexer->from(\*DATA);

 $lexer = Parse::Lex->new(@token);

 $lexer->from(\*DATA);

 TOKEN:while (1) {
   $token = $lexer->next;
   last TOKEN if ($lexer->eoi);
 }

__END__
main()  {  /* comment */
  printf("hi! /* \"this\" is not a comment */"); /* another /*nested*/
                  comment */
}
```

Al ejecutarlo, produce la salida:

```
> nestedcom.pl
main()  {
  printf("hi! /* \"this\" is not a comment */");
}
```

12.4. La Clase `Parse::CLex`

Relacionada con `Parse::Lex` está la clase `Parse::CLex`, la cual avanza consumiendo la cadena analizada mediante el uso del operador de sustitución (`s///`). Los analizadores producidos mediante esta segunda clase no permiten el uso de anclas en las expresiones regulares. Tampoco disponen de acceso a la subclase `Parse::Token`. He aqui el mismo ejemplo, usando la clase `Parse::CLex`:

```
> cat -n ctokenizer.pl
#!/usr/local/bin/perl -w

require 5.000;
BEGIN {  unshift @INC, "../lib"; }
use Parse::CLex;

@token = (
          qw(
             ADDOP    [-+]
             LEFTP    [\(]
             RIGHTP   [\)]
             INTEGER  [1-9][0-9]*
             NEWLINE  \n
            ),
          qw(STRING),   [qw(" (?:[^"]+|"")* ")],
          qw(ERROR  .*), sub {
            die qq!can\'t analyze: "$_[1]"!;
          }
         );

Parse::CLex->trace;
$lexer = Parse::CLex->new(@token);

$lexer->from(\*DATA);
print "Tokenization of DATA:\n";

TOKEN:while (1) {
  $token = $lexer->next;
  if (not $lexer->eoi) {
    print "Record number: ", $lexer->line, "\n";
    print "Type: ", $token->name, "\t";
    print "Content:->", $token->getText, "<-\n";
  } else {
    last TOKEN;
  }
}

__END__
1+2-5
"This is a multiline
string with an embedded "" in it"
this is an invalid string with a "" in it"


> ctokenizer.pl
Trace is ON in class Parse::CLex
Tokenization of DATA:
[main::lexer|Parse::CLex] Token read (INTEGER, [1-9][0-9]*): 1
Record number: 1
```

```
Type: INTEGER   Content:->1<-
[main::lexer|Parse::CLex] Token read (ADDOP, [-+]): +
Record number: 1
Type: ADDOP     Content:->+<-
[main::lexer|Parse::CLex] Token read (INTEGER, [1-9][0-9]*): 2
Record number: 1
Type: INTEGER   Content:->2<-
[main::lexer|Parse::CLex] Token read (ADDOP, [-+]): -
Record number: 1
Type: ADDOP     Content:->-<-
[main::lexer|Parse::CLex] Token read (INTEGER, [1-9][0-9]*): 5
Record number: 1
Type: INTEGER   Content:->5<-
[main::lexer|Parse::CLex] Token read (NEWLINE, \n):

Record number: 1
Type: NEWLINE   Content:->
<-
[main::lexer|Parse::CLex] Token read (STRING, \"(?:[^\"]+|\"\")*\"): "This is a multiline
string with an embedded "" in it"
Record number: 3
Type: STRING    Content:->"This is a multiline
string with an embedded "" in it"<-
[main::lexer|Parse::CLex] Token read (NEWLINE, \n):

Record number: 3
Type: NEWLINE   Content:->
<-
[main::lexer|Parse::CLex] Token read (ERROR, .*): this is an invalid string with a "" in it"
can't analyze: "this is an invalid string with a "" in it"" at ctokenizer.pl line 17, <DATA> line 4.
```

12.5. Usando `Text::Balanced`

La función `extract_multiple` del módulo `Text::Balanced` puede ser considerada un generador de analizadores léxicos. El primer argumento es la cadena a ser procesada y el segundo la lista de *extractores* a aplicar a dicha cadena. Un extractor puede ser una subrutina, pero también una expresión regular o una cadena.

En un contexto de lista devuelve la lista con las subcadenas de la cadena original, según han sido producidas por los extractores. En un contexto escalar devuelve la primera subcadena que pudo ser extraída de la cadena original. En cualquier caso, `extract_multiple` empieza en la posición actual de búsqueda indicada por `pos` y actualiza el valor de `pos` de manera adecuada.

Si el extractor es una referencia a un hash, deberá contener exactamente un elemento. La clave actúa como nombre del terminal o *token* y el valor es un extractor. La clave es utilizada para bendecir la cadena casada por el extractor.

Veamos un programa de ejemplo:

```
$ cat extract_variable.pl
#!/usr/local/bin/perl5.8.0 -w

use strict;
use Text::Balanced qw( extract_variable extract_quotelike
                       extract_codeblock extract_multiple);

my $text = <<'EOS'
```

```
#!/usr/local/bin/perl5.8.0 -w

$pattern = shift || '';
$pattern =~ s{::}{/}g;
$pattern =~ s{$}{.pm};
@dirs = qw(
/usr/local/lib/perl5/5.8.0/i686-linux
/usr/local/lib/perl5/5.8.0
/usr/local/lib/perl5/site_perl/5.8.0/i686-linux
/usr/local/lib/perl5/site_perl/5.8.0
/usr/local/lib/perl5/site_perl
/usr/lib/perl5
/usr/share/perl5
/usr/lib/perl/5.6.1
/usr/share/perl/5.6.1
);

for (@dirs) {
  my $file = $_."/".$pattern;
  print "$file\n" if (-e $file);
}

EOS
;
my @tokens = &extract_multiple(
  $text,
  [
   {variable => sub { extract_variable($_[0], '') }},
   {quote => sub { extract_quotelike($_[0],'') }},
   {code => sub { extract_codeblock($_[0],'{}','') }},
   {comment => qr/#.*/}
  ],
  undef,
  0
)
;

for (@tokens) {
  if (ref($_)) {
    print ref($_),"=> $$_\n";
  }
  else {
    print "other things: $_\n";
  }
}
```

La subrutina `extract_variable` extrae cualqueir variable Perl válida. El primer argumento es la cadena a ser procesada, el segundo es una cadena que especifica un patrón indicando el prefijo a saltarse. Si se omite como en el ejemplo, se usarán blancos. La subrutina `extract_quotelike` extrae cadenas Perl en cualqueira de sus múltiples formas de representación. La subrutina `extract_codeblock` extrae un bloque de código.

Ejecución del programa anterior:

```
$ ./extract_variable.pl
```

```
comment=> #!/usr/local/bin/perl5.8.0 -w
other things:


variable=> $pattern
other things:  = shift ||
quote=> ''
other things: ;

variable=> $pattern
other things:  =~
quote=> s{::}{/}g
other things: ;

variable=> $pattern
other things:  =~
quote=> s{$}{.pm}
other things: ;

variable=> @dirs
other things:  =
quote=> qw(
/usr/local/lib/perl5/5.8.0/i686-linux
/usr/local/lib/perl5/5.8.0
/usr/local/lib/perl5/site_perl/5.8.0/i686-linux
/usr/local/lib/perl5/site_perl/5.8.0
/usr/local/lib/perl5/site_perl
/usr/lib/perl5
/usr/share/perl5
/usr/lib/perl/5.6.1
/usr/share/perl/5.6.1
)
other things: ;

for (
variable=> @dirs
other things: )
code=> {
  my $file = $_."/".$pattern;
  print "$file\n" if (-e $file);
}
other things:
```

Capítulo 13

RecDescent

13.1. Introducción

`Parse::RecDescent` es un módulo, escrito por Damian Conway, que permite construir analizadores sintácticos descendentes con vuelta atrás. No forma parte de la distribución estándar de Perl, por lo que es necesario descargarlo desde CPAN (consulte la sección 6.11.3 para una descripción de como hacerlo). En lo que sigue, y por brevedad, nos referiremos a el como RD.

```
 1 #!/usr/local/bin/perl5.8.0 -w
 2 use strict;
 3 use warnings;
 4 
 5 use Parse::RecDescent;
 6 
 7 
 8 my $grammar = q {
 9 
10   start:  seq_1 seq_2
11 
12   seq_1    :   'A' 'B' 'C' 'D'
13                { print "seq_1: " . join (" ", @item[1..$#item]) . "\n" }
14              | 'A' 'B' 'C' 'D' 'E' 'F'
15                { print "seq_1: " . join (" ", @item[1..$#item]) . "\n" }
16 
17   seq_2    : character(s)
18 
19   character: /\w/
20                { print "character: $item[1]\n" }
21 
22 };
23 
24 my $parser=Parse::RecDescent->new($grammar);
25 
26 $parser->start("A B C D E F");
```

En la línea 5 se declara el uso del módulo. En la línea 8 se guarda la gramática en la variable escalar `$grammar`. La variable sintáctica `start` es considerada de arranque de la gramática. Las partes derechas de las producciones se separan mediante la barra `|`, y las acciones se insertan entre llaves. El contenido de las acciones es código Perl. Dentro de esas acciones es posible hacer alusión a los atributos de los símbolos de la producción a través del array `@item`. Así `$item[1]` contendrá el atributo asociado al primer símbolo (el carácter `A` en este caso). Para que una producción tenga éxito es necesario que la acción devuelva un valor definido (no necesariamente verdadero). El valor asociado con una producción es el valor retornado por la acción, o bien el valor asociado con la variable `$return` (si aparece en la acción) o con el item asociado con el último símbolo que casó con éxito.

Es posible indicar una o mas repeticiones de una forma sentencial como en la línea 17. La regla:

```
seq_2 : character(s)
```

indica que el lenguaje `seq_2` se forma por la repetición de uno o mas caracteres (`character`). El valor asociado con una pluralidad de este tipo es una referencia a un array conteniendo los valores individuales de cada uno de los elementos que la componen, en este caso de los `character`. Es posible introducir un *patrón de separación* que indica la forma en la que se deben separar los elementos:

```
seq_2 : character(s /;/)
```

lo cual especifica que entre `character` y `character` el analizador debe saltarse un punto y coma.

Como se ve en la línea 19 es legal usar expresiones regulares en la parte derecha. La llamada al constructor en la línea 24 deja en `$parser` un analizador sintáctico recursivo descendente. El analizador así construido dispone de un método por cada una de las variables sintácticas de la gramática. El método asociado con una variable sintáctica reconoce el lenguaje generado por esa variable sintáctica. Así, la llamada de la línea 26 al método `start` reconoce el lenguaje generado por `start`. Si la variable sintáctica tiene éxito el método retorna el valor asociado. En caso contrario se devuelve `undef`. Esto hace posible retornar valores como `0, "0", or ""`.

La siguiente ejecución muestra el orden de recorrido de las producciones.

```
$ ./firstprodnolongest.pl
seq_1: A B C D
character: E
character: F
```

¿Que ha ocurrido? RD es perezoso en la evaluacion de las producciones. Como la primera regla tiene éxito y es evaluada antes que la segunda, retorna el control a la aprte derecha de la regla de `start`. A continuación se llama a `seq2`.

En este otro ejemplo invertimos el orden de las reglas de producción de `seq_1`:

```
#!/usr/local/bin/perl5.8.0 -w

use strict;
use warnings;

use Parse::RecDescent;

my $grammar = q {

  start:  seq_1 seq_2

  seq_1    :  'A' 'B' 'C' 'D' 'E' 'F'
               { print "seq_1: " . join (" ", @item[1..$#item]) . "\n" }
           | 'A' 'B' 'C' 'D'
               { print "seq_1: " . join (" ", @item[1..$#item]) . "\n" }


  seq_2    : character(s)

  character: /\w/
               { print "character: $item[1]\n" }
```

```
};

my $parser=Parse::RecDescent->new($grammar);

$parser->start("A B C D E F");
```

La ejecución resultante da lugar al éxito de la primera regla, que absorbe toda la entrada:

```
$ ./firstprodnolongest2.pl
seq_1: A B C D E F
```

13.2. Orden de Recorrido del Árbol de Análisis Sintáctico

RD es un analizador sintáctico descendente recursivo pero no es predictivo. Cuando estamos en la subrutina asociada con una variable sintáctica, va probando las reglas una a una hasta encontrar una que tiene éxito. En ese momento la rutina termina retornando el valor asociado. Consideremos la siguiente gramática (véase el clásico libro de Aho et al. [11], página 182):

$$\begin{array}{l} S \rightarrow cAd \\ A \rightarrow ab \mid a \end{array}$$

Claramente la gramática no es LL(1). Codifiquemos la gramática usando RD:

```
$ cat -n aho182.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use warnings;
 4
 5  use Parse::RecDescent;
 6
 7  $::RD_TRACE = 1;
 8  $::RD_AUTOACTION =  q{ 1 };
 9  my $grammar = q {
10    S : 'c' A 'd'  { print "Éxito!\n" }
11    A : 'a' 'b'
12      | 'a'
13  };
14
15  my $parser=Parse::RecDescent->new($grammar);
16  my $input = <>;
17  $parser->S($input);
```

en este ejemplo usamos dos nuevas variables (líneas 7 y 8) que gobiernan la conducta de un analizador generado por RD. Las variables `$::RD_TRACE` y `$::RD_AUTOACTION`. La primera hace que el analizador generado vuelque en `stderr` un informe del análisis. La asignación en la línea 8 de una cadena a `$::RD_AUTOACTION` hace que todas las producciones con las cuáles no se asocie una acción explícitamente, tengan como acción asociada la cadena que figura en la variable `$::RD_AUTOACTION` (en este caso, devolver `1`). Así las producciones en las líneas 11 y 12 tienen asociadas la acción `{ 1 }`, mientras que la acción asociada con la producción en la línea 10 es `{ print "Éxito!\n" }`.

Obsérvese que, dado que el analizador se ejecuta dentro de su propio espacio de nombres, las variables que pertenzcan al paquete `Main` como es el caso de `RD_AUTOACTION` deben ser explicitadas prefijándolas con el nombre del paquete.

Veamos el comportamiento del programa con la entrada `c a d`. Al ejecutar el programa, lo primero que ocurre, al construir el objeto analizador en la línea 15 es que RD nos informa de su interpretación de los diferentes símbolos en la gramática:

```
$ ./aho182.pl
    Parse::RecDescent: Treating "S :" as a rule declaration
    Parse::RecDescent: Treating "c" as a literal terminal
    Parse::RecDescent: Treating "A" as a subrule match
    Parse::RecDescent: Treating "d" as a literal terminal
    Parse::RecDescent: Treating "{ print "Éxito!\n" }" as an action
    Parse::RecDescent: Treating "A :" as a rule declaration
    Parse::RecDescent: Treating "a" as a literal terminal
    Parse::RecDescent: Treating "b" as a literal terminal
    Parse::RecDescent: Treating "|" as a new production
    Parse::RecDescent: Treating "a" as a literal terminal
printing code (12078) to RD_TRACE
```

Ahora se ejecuta la entrada en la línea 16, tecleamos `c a d` y sigue una información detallada de la cosntrucción del árbol sintáctico concreto:

```
c a d
 1|    S     |Trying rule: [S]                      |
 1|    S     |                                      |"c a d\n"
 1|    S     |Trying production: ['c' A 'd']        |
 1|    S     |Trying terminal: ['c']                |
 1|    S     |>>Matched terminal<< (return value:   |
  |          |[c])                                  |
```

La salida aparece en cuatro columnas. El número de línea en el texto de la gramática, la variable sintáctica (que en la jerga RD se denomina regla o subregla), una descripción verbal de lo que se está haciendo y por último, la cadena de entrada que queda por leer. Se acaba de casar `c` y por tanto se pasa a llamar al método asociado con `A`. en la entrada queda `a d\n`:

```
 1|    S     |                                      |" a d\n"
 1|    S     |Trying subrule: [A]                   |
 2|    A     |Trying rule: [A]                      |
 2|    A     |Trying production: ['a' 'b']          |
 2|    A     |Trying terminal: ['a']                |
 2|    A     |>>Matched terminal<< (return value:   |
  |          |[a])                                  |
 2|    A     |                                      |" d\n"
 2|    A     |Trying terminal: ['b']                |
 2|    A     |<<Didn't match terminal>>             |
```

RD intenta infructuosamente la primera producción

$$A \rightarrow a\ b$$

por tanto va a probar con la segunda producción de A:

```
 2|    A     |                                      |"d\n"
 2|    A     |Trying production: ['a']              |
 2|    A     |                                      |" a d\n"
 2|    A     |Trying terminal: ['a']                |
 2|    A     |>>Matched terminal<< (return value:   |
  |          |[a])                                  |
 2|    A     |                                      |" d\n"
 2|    A     |Trying action                         |
 2|    A     |>>Matched action<< (return value: [1])|
 2|    A     |>>Matched production: ['a']<<         |
```

```
 2|    A     |>>Matched rule<< (return value: [1])  |
 2|    A     |(consumed: [ a])                      |
 1|    S     |>>Matched subrule: [A]<< (return      |
  |          |value: [1]                            |
 1|    S     |Trying terminal: ['d']                |
 1|    S     |>>Matched terminal<< (return value:   |
  |          |[d])                                  |
 1|    S     |                                      |"\n"
 1|    S     |Trying action                         |
Éxito!
 1|    S     |>>Matched action<< (return value: [1])|
 1|    S     |>>Matched production: ['c' A 'd']<<   |
 1|    S     |>>Matched rule<< (return value: [1])  |
 1|    S     |(consumed: [c a d])                   |
```

De este modo la frase es aceptada. Sin embargo, ¿que ocurriría si invertimos el orden de las producciones de `A` y damos como entrada la cadena `c a b d`? Puesto que RD es ocioso, la ahora primera regla `A : a` tendrá éxito y el control retornará al método asociado con `S`:

```
$ ./aho182_reverse.pl
c a b d
 1|    S     |Trying rule: [S]                      |
 1|    S     |                                      |"c a b d\n"
 1|    S     |Trying production: ['c' A 'd']        |
 1|    S     |Trying terminal: ['c']                |
 1|    S     |>>Matched terminal<< (return value:   |
  |          |[c])                                  |
 1|    S     |                                      |" a b d\n"
 1|    S     |Trying subrule: [A]                   |
 2|    A     |Trying rule: [A]                      |
 2|    A     |Trying production: ['a']              |
 2|    A     |Trying terminal: ['a']                |
 2|    A     |>>Matched terminal<< (return value:   |
  |          |[a])                                  |
 2|    A     |                                      |" b d\n"
 2|    A     |Trying action                         |
 2|    A     |>>Matched action<< (return value: [1])|
 2|    A     |>>Matched production: ['a']<<         |
 2|    A     |>>Matched rule<< (return value: [1])  |
 2|    A     |(consumed: [ a])                      |
```

Ahora RD va a rechazar la cadena. No intenta llamar de nuevo al método asociado con `A` para que haga nuevos intentos:

```
 1|    S     |>>Matched subrule: [A]<< (return      |
  |          |value: [1]                            |
 1|    S     |Trying terminal: ['d']                |
 1|    S     |<<Didn't match terminal>>             |
 1|    S     |                                      |"b d\n"
 1|    S     |<<Didn't match rule>>                 |
```

Debes, por tanto, ser cuidadoso con el orden en el que escribes las producciones. El orden en que las escribas tiene una importante incidencia en la conducta del analizador. *En general el analizador generado por RD no acepta el lenguaje generado por la gramática*, en el sentido formal de la expresión *lenguaje generado*, tal y como fué definida en la definición 11.5.4. En este sentido, hay una separación semántica similar a la que ocurre en las expresiones regulares. El *or* en las expresiones regulares de Perl también es perezoso, a diferencia de la convención adoptada en el estudio teórico de los lenguajes regulares y los autómatas.

Ejercicio 13.2.1. *Dado el siguiente programa Perl*

```
#!/usr/local/bin/perl5.8.0 -w
use strict;
use warnings;
use Parse::RecDescent;

$::RD_TRACE = 1;
my $grammar = q {

  start:  seq_1 ';'
  seq_1     : 'A' 'B' 'C' 'D'
               { print "seq_1: " . join (" ", @item[1..$#item]) . "\n" }
            |  'A' 'B' 'C' 'D' 'E' 'F'
               { print "seq_1: " . join (" ", @item[1..$#item]) . "\n" }
};

my $parser=Parse::RecDescent->new($grammar);
my $result = $parser->start("A B C D E F ;");
if (defined($result)) {
  print "Valida\n";
} else { print "No reconocida\n"; }
```

¿Cuál es la salida?

Indica que ocurre si cambiamos el lugar en el que aparece el separador punto y coma, poniéndolo en las producciones de `seq_1` *como sigue:*

```
#!/usr/local/bin/perl5.8.0 -w
use strict;
use warnings;
use Parse::RecDescent;

$::RD_TRACE = 1;
my $grammar = q {

  start:  seq_1
  seq_1     : 'A' 'B' 'C' 'D' ';'
               { print "seq_1: " . join (" ", @item[1..$#item]) . "\n" }
            |  'A' 'B' 'C' 'D' 'E' 'F' ';'
               { print "seq_1: " . join (" ", @item[1..$#item]) . "\n" }
};

my $parser=Parse::RecDescent->new($grammar);

my $result = $parser->start("A B C D E F ;");
if (defined($result)) {
  print "Valida\n";
} else { print "No reconocida\n"; }
```

¿Podrías resumir en palabras la forma en la que RD explora el árbol de análisis concreto? Intenta escribir un seudocódigo que resuma el comportamiento de RD.

13.3. La ambiguedad de las sentencias `if-then-else`

Existen lenguajes que son *intrínsecamente ambiguos*, esto lenguajes independientes del contexto tales que, cualquier gramática que los genere es ambigua. Sin embargo, lo normal es que siempre se pueda encontrar una gramática que no sea ambigua y que genere el mismo lenguaje.

Existe una ambiguedad en la parte de las sentencias condicionales de un lenguaje. Por ejemplo, consideremos la gramática:

$$
\begin{array}{l}
\text{SENTENCIA} \rightarrow \text{if EXPRESION then SENTENCIA} \\
\text{SENTENCIA} \rightarrow \text{if EXPRESION then SENTENCIA else SENTENCIA} \\
\text{SENTENCIA} \rightarrow \text{OTRASSENTENCIAS}
\end{array}
$$

Aqui OTRASSENTENCIAS es un terminal/comodín para aludir a cualesquiera otras sentencias que el lenguaje peuda tener (bucles, asignaciones, llamadas a subrutinas, etc.)

La gramática 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:

$$\text{if } E_1 \text{ then (if } E_2 \text{ then } S_1) \text{ else } S_2$$

El siguiente ejemplo considera una aproximación a la resolución del problema usando `Parse::RecDescent`. Para simplificar, la única variable sintáctica que permanece del ejemplo anterior es SENTENCIA (que ha sido renombrada `st`). Las demás han sido convertidas en terminales y se han abreviado. En esta gramática colocamos primero la regla mas larga `st : 'iEt' st 'e' st`.

```
#!/usr/local/bin/perl5.8.0 -w
use strict;
use Parse::RecDescent;
use Data::Dumper;

$::RD_TRACE = 1;
$::RD_AUTOACTION =  q{ [@item] };
my $grammar = q{
  prog : st ';'
  st : 'iEt' st 'e' st
     | 'iEt' st
     | 'o' { 'o' }
};

my $parse = Parse::RecDescent->new($grammar);

my $line;
while ($line = <>) {
  print "$line\n";
  my $result = $parse->prog($line);
  if (defined($result)) { print Dumper($result); }
  else { print "Cadena no válida\n"; }
}
```

En este caso, la variable `$::RD_AUTOACTION` ha sido establecida a `q{ [@item] }`. Esto significa que cada vez que una producción tenga éxito se devolverá una referencia al array conteniendo los atributos de los símbolos en la producción. El contenido de esta variable es evaluado siempre que la producción en cuestión no tenga una acción asociada explícitamente: Por tanto, se aplica para todas las reglas salvo para la regla en la línea 12, en la que se devuelve el carácter `'o'`.

El programa anterior da lugar a la siguiente ejecución:

```
bash-2.05b$ ./ifthenelse.pl file4.txt
```

El fichero `file4.txt` contiene una sóla línea:

```
$ cat file4.txt
iEt iEt o e o ;
```

que es una frase con dos posibles árboles. RD intentará primero reiteradamente la primera de las producciones de `st`:

```
 1|   prog   |Trying rule: [prog]                 |
 1|   prog   |                                    |"iEt iEt o e o ;\n"
 1|   prog   |Trying production: [st ';']         |
 1|   prog   |Trying subrule: [st]                |
 2|    st    |Trying rule: [st]                   |
 2|    st    |Trying production: ['iEt' st 'e' st]  |
 2|    st    |Trying terminal: ['iEt']            |
 2|    st    |>>Matched terminal<< (return value:  |
  |          |[iEt])                              |
 2|    st    |                                    |" iEt o e o ;\n"
 2|    st    |Trying subrule: [st]                |
 3|    st    |Trying rule: [st]                   |
 3|    st    |Trying production: ['iEt' st 'e' st]  |
 3|    st    |Trying terminal: ['iEt']            |
 3|    st    |>>Matched terminal<< (return value:  |
  |          |[iEt])                              |
 3|    st    |                                    |" o e o ;\n"
```

Esta opción acabará fracasando, ya que sólo hay un `else`.

```
 3|    st    |Trying subrule: [st]                |
 4|    st    |Trying rule: [st]                   |
 4|    st    |Trying production: ['iEt' st 'e' st]  |
 4|    st    |Trying terminal: ['iEt']            |
 4|    st    |<<Didn't match terminal>>           |
```

Estamos delante del terminal `o` y obviamente la primera producción de `st` no casa, se intenta con la segunda, la cual naturalmente fracasrá:

```
 4|    st    |                                    |"o e o ;\n"
 4|    st    |Trying production: ['iEt' st]       |
 4|    st    |                                    |" o e o ;\n"
 4|    st    |Trying terminal: ['iEt']            |
 4|    st    |<<Didn't match terminal>>           |
 4|    st    |                                    |"o e o ;\n"
 4|    st    |Trying production: ['o']            |
```

Seguimos delante del terminal `o` y la segunda producción de `st` tampoco casa, se intenta con la tercera producción:

```
 4|    st    |                                    |" o e o ;\n"
 4|    st    |Trying terminal: ['o']              |
 4|    st    |>>Matched terminal<< (return value: |
  |          |[o])                                |
 4|    st    |                                    |" e o ;\n"
 4|    st    |Trying action                       |
 4|    st    |>>Matched action<< (return value: [o])|
```

La `o` por fin ha sido emparejada. Se ha ejecutado la acción no automática.

```
 4|    st    |>>Matched production: ['o']<<       |
 4|    st    |>>Matched rule<< (return value: [o]) |
 4|    st    |(consumed: [ o])                    |
 3|    st    |>>Matched subrule: [st]<< (return   |
  |          |value: [o]                          |
```

Recuérdese por donde íbamos conjeturando. Se ha construido el árbol (erróneo):

```
st
 |
 ` 'iEt'  st    'e'  st
           |
           ` 'iEt'  st    'e'  st
                    |      ^
                    ` 'o'
```

Por ello, el `e` va a ser consumido sin problema:

```
 3|    st    |Trying terminal: ['e']              |
 3|    st    |>>Matched terminal<< (return value: |
  |          |[e])                                |
 3|    st    |                                    |" o ;\n"
 3|    st    |Trying subrule: [st]                |
 4|    st    |Trying rule: [st]                   |
 4|    st    |Trying production: ['iEt' st 'e' st] |
 4|    st    |Trying terminal: ['iEt']            |
 4|    st    |<<Didn't match terminal>>           |
 4|    st    |                                    |"o ;\n"
 4|    st    |Trying production: ['iEt' st]       |
 4|    st    |                                    |" o ;\n"
 4|    st    |Trying terminal: ['iEt']            |
 4|    st    |<<Didn't match terminal>>           |
 4|    st    |                                    |"o ;\n"
 4|    st    |Trying production: ['o']            |
 4|    st    |                                    |" o ;\n"
 4|    st    |Trying terminal: ['o']              |
 4|    st    |>>Matched terminal<< (return value: |
  |          |[o])                                |
 4|    st    |                                    |" ;\n"
 4|    st    |Trying action                       |
 4|    st    |>>Matched action<< (return value: [o])|
 4|    st    |>>Matched production: ['o']<<       |
 4|    st    |>>Matched rule<< (return value: [o]) |
 4|    st    |(consumed: [ o])                    |
 3|    st    |>>Matched subrule: [st]<< (return   |
```

```
  |          |value: [o]                         |
 3|    st    |Trying action                      |
 3|    st    |>>Matched action<< (return value:  |
  |          |[ARRAY(0x830cc04)])                |
```

Hemos construido el árbol:

```
st
 |
 ` 'iEt'  st    'e'  st
          |     ^
          ` 'iEt'  st    'e'  st
                   |          |
                   ` 'o'      `o
```

Asi pues la segunda regla conjeturada `st : 'iEt' st 'e' st` ha tenido éxito. Después de esto esperamos ver `e`, pero lo que hay en la entrada es un punto y coma.

```
 3|    st    |>>Matched production: ['iEt' st 'e'   |
  |          |st]<<                              |
 3|    st    |>>Matched rule<< (return value:    |
  |          |[ARRAY(0x830cc04)])                |
 3|    st    |(consumed: [ iEt o e o])           |
 2|    st    |>>Matched subrule: [st]<< (return  |
  |          |value: [ARRAY(0x830cc04)]          |
 2|    st    |Trying terminal: ['e']             |
 2|    st    |<<Didn't match terminal>>          |
 2|    st    |                                   |";\n"
```

Se ha fracasado. Se probará a este nivel con la siguiente regla `st : 'iEt' st` (que acabará produciendo un árbol de acuerdo con la regla del "if" mas cercano). Mágicamente, la entrada consumida es devuelta para ser procesada de nuevo (obsérvese la tercera columna). Sin embargo, los efectos laterales que las acciones ejecutadas por las acciones asociadas con los falsos éxitos permanecen. En este caso, por la forma en la que hemos escrito las acciones, no hay ningún efecto lateral.

```
 2|    st    |Trying production: ['iEt' st]      |
 2|    st    |                                   |"iEt iEt o e o ;\n"
 2|    st    |Trying terminal: ['iEt']           |
 2|    st    |>>Matched terminal<< (return value:  |
  |          |[iEt])                             |
 2|    st    |                                   |" iEt o e o ;\n"
 2|    st    |Trying subrule: [st]               |
 3|    st    |Trying rule: [st]                  |
 3|    st    |Trying production: ['iEt' st 'e' st]  |
```

Ahora el analizador va a intentar el árbol:

```
st
 |
 ` 'iEt'  st
          |
          ` 'iEt'  st    'e'  st
```

que se corresponde con la interpretación clásica: *El* `else` *casa con el* `if` *mas cercano.*

```
 3|    st    |Trying terminal: ['iEt']           |
 3|    st    |>>Matched terminal<< (return value:  |
  |          |[iEt])                             |
 3|    st    |                                   |" o e o ;\n"
 3|    st    |Trying subrule: [st]               |
```

La ineficiencia de RD es clara aqui. Va a intentar de nuevo las diferentes reglas hasta dar con la del 'o'

```
 4|    st    |Trying rule: [st]                     |
 4|    st    |Trying production: ['iEt' st 'e' st]  |
 4|    st    |Trying terminal: ['iEt']              |
 4|    st    |<<Didn't match terminal>>             |
 4|    st    |                                      |"o e o ;\n"
 4|    st    |Trying production: ['iEt' st]         |
 4|    st    |                                      |" o e o ;\n"
 4|    st    |Trying terminal: ['iEt']              |
 4|    st    |<<Didn't match terminal>>             |
 4|    st    |                                      |"o e o ;\n"
 4|    st    |Trying production: ['o']              |
 4|    st    |                                      |" o e o ;\n"
 4|    st    |Trying terminal: ['o']                |
 4|    st    |>>Matched terminal<< (return value:   |
  |          |[o])                                  |
 4|    st    |                                      |" e o ;\n"
 4|    st    |Trying action                         |
 4|    st    |>>Matched action<< (return value: [o])|
 4|    st    |>>Matched production: ['o']<<         |
 4|    st    |>>Matched rule<< (return value: [o])  |
 4|    st    |(consumed: [ o])                      |
 3|    st    |>>Matched subrule: [st]<< (return     |
  |          |value: [o]                            |
```

Ahora el "else" será consumido por la sentencia condicional anidada:

```
 3|    st    |Trying terminal: ['e']                |
 3|    st    |>>Matched terminal<< (return value:   |
  |          |[e])                                  |
 3|    st    |                                      |" o ;\n"
```

y de nuevo debemos pasar por el calvario de todas las reglas, ya que la `'o'` es la última de las reglas:

```
 3|    st    |Trying subrule: [st]                  |
 4|    st    |Trying rule: [st]                     |
 4|    st    |Trying production: ['iEt' st 'e' st]  |
 4|    st    |Trying terminal: ['iEt']              |
 4|    st    |<<Didn't match terminal>>             |
 4|    st    |                                      |"o ;\n"
 4|    st    |Trying production: ['iEt' st]         |
 4|    st    |                                      |" o ;\n"
 4|    st    |Trying terminal: ['iEt']              |
 4|    st    |<<Didn't match terminal>>             |
 4|    st    |                                      |"o ;\n"
 4|    st    |Trying production: ['o']              |
 4|    st    |                                      |" o ;\n"
 4|    st    |Trying terminal: ['o']                |
 4|    st    |>>Matched terminal<< (return value:   |
  |          |[o])                                  |
 4|    st    |                                      |" ;\n"
 4|    st    |Trying action                         |
 4|    st    |>>Matched action<< (return value: [o])|
```

```
 4|    st    |>>Matched production: ['o']<<         |
 4|    st    |>>Matched rule<< (return value: [o])  |
 4|    st    |(consumed: [ o])                      |
```

A partir de aqui todo encaja. Nótese la ejecución de la acción automática:

```
 3|    st    |>>Matched subrule: [st]<< (return     |
  |          |value: [o]                            |
 3|    st    |Trying action                         |
 3|    st    |>>Matched action<< (return value:     |
  |          |[ARRAY(0x82f2774)])                   |
 3|    st    |>>Matched production: ['iEt' st 'e'   |
  |          |st]<<                                 |
 3|    st    |>>Matched rule<< (return value:       |
  |          |[ARRAY(0x82f2774)])                   |
 3|    st    |(consumed: [ iEt o e o])              |
 2|    st    |>>Matched subrule: [st]<< (return     |
  |          |value: [ARRAY(0x82f2774)]             |
 2|    st    |Trying action                         |
 2|    st    |>>Matched action<< (return value:     |
  |          |[ARRAY(0x830c41c)])                   |
 2|    st    |>>Matched production: ['iEt' st]<<    |
 2|    st    |>>Matched rule<< (return value:       |
  |          |[ARRAY(0x830c41c)])                   |
 2|    st    |(consumed: [iEt iEt o e o])           |
 1|   prog   |>>Matched subrule: [st]<< (return     |
  |          |value: [ARRAY(0x830c41c)]             |
```

A estas alturas hemos construido el árbol:

```
st
 |
` 'iEt'  st
          |
         ` 'iEt'  st    'e'  st
                  |          |
                  ` 'o'      `o
```

y el punto y coma nos espera en la entrada:

```
 1|   prog   |Trying terminal: [';']                |
 1|   prog   |>>Matched terminal<< (return value:   |
  |          |[;])                                  |
 1|   prog   |                                      |"\n"
 1|   prog   |Trying action                         |
 1|   prog   |>>Matched action<< (return value:     |
  |          |[ARRAY(0x830c314)])                   |
 1|   prog   |>>Matched production: [st ';']<<      |
 1|   prog   |>>Matched rule<< (return value:       |
  |          |[ARRAY(0x830c314)])                   |
 1|   prog   |(consumed: [iEt iEt o e o ;])         |
```

Ahora ya se ejecuta la línea `print Dumper($result)` en el programa principal, volcando la estructura de datos construida durante el análisis:

```
$VAR1 = [ 'prog',
          [ 'st',
```

```
            'iEt',
            [ 'st',
              'iEt', 'o', 'e', 'o'
            ]
          ],
          ';'
        ];
```

Ejercicio 13.3.1. *Si cambiamos el orden de las producciones y no forzamos a que la sentencia acabe en punto y coma:*

```
my $grammar = q{
  st : 'iEt' st { [ @item ] }
     | 'iEt' st 'e' st { [ @item ] }
     | 'o'
};
```

¿Que árbol se obtendrá al darle la entrada `iEt iEt o e o ;`*? ¿Que ocurre si, manteniendo este orden, forzamos a que el programa termine en punto y coma?*

```
my $grammar = q{
  prog : st ';'
  st : 'iEt' st { [ @item ] }
     | 'iEt' st 'e' st { [ @item ] }
     | 'o'
};
```

¿Es aceptada la cadena?

Ejercicio 13.3.2. *La regla "tod else casa con el if mas cercano" puede hacerse explícita usando esta otra gramática:*

SENTENCIA	→	*EQUI*	| *NOEQUI*
EQUI	→	*if EXPRESION then EQUI else EQUI*	| *OTRASSENTENCIAS*
NOEQUI	→	*if EXPRESION then SENTENCIA*	|
		if EXPRESION then EQUI else NOEQUI	

Escriba un analizador sintáctico para la gramática usando RD y analice su comportamiento.

13.4. La directiva `commit`

El excesivo costo de RD puede aliviarse haciendo uso de las directivas `<commit>` y `<uncommit>`, las cuales permiten podar el árbol de búsqueda. Una directiva `<commit>` le indica al analizador que debe ignorar las producciones subsiguientes si la producción actual fracasa. Es posible cambiar esta conducta usando posteriormente la directiva `<uncommit>`, la cual revoca el estatus del último `<commit>`. Así la gramática del if-then-else puede reescribirse como sigue:

```
$ cat -n ifthenelse_commit.pl
#!/usr/bin/perl -w
use strict;
use Parse::RecDescent;
use Data::Dumper;

$::RD_TRACE = 1;
$::RD_AUTOACTION =  q{ [@item] };
```

```
 8  my $grammar = q{
 9    prog : st ';'
10    st : 'iEt' <commit> st <uncommit> 'e' st
11       | 'iEt' <commit> st
12       | 'o' { 'o' }
13  };
14  ...
```

en este caso, si no se encuentra una sentencia después de `iEt` se producirá directamente el fracaso de `st` sin intentar las otras dos reglas. Sin embargo, si no se encuentra un `e` después de la sentencia si que se va a intentar la regla de la línea 11, ya que la directiva `<uncommit>` revoca el `<commit>` previo.

Observe el resultado de la ejecución: la acción automática de construcción del árbol da lugar a la inserción de los valores devueltos por las directivas `<commit>`:

```
./ifthenelse_commit.pl file4.txt

iEt iEt o e o ;

$VAR1 = [ 'prog',
          [ 'st',
            'iEt', 1, [ 'st',
                        'iEt', 1, 'o', 1, 'e', 'o'
                      ]
          ],
          ';'
        ];
```

Ejercicio 13.4.1. *¿Cuanto ahorro produce en este caso, el uso de las directivas* `<commit>` *en la gramática anterior? (supuesto que la entrada es* `iEt iEt o e o ;`*) ¿Se obtiene algún beneficio?*

13.5. Las Directivas `skip` y `leftop`

Consideremos una extensión al conocido problema del reconocimietno del lenguaje de *valores separados por comas* (véase la sección 4.21). El fichero de entrada contiene líneas como esta:

```
"tierra, luna",1,"marte, deimos",9.374
```

la extensión consiste en admitir que el separador puede cambiar entre líneas y en lso campos de una línea, pudiendo ser no sólo la coma sino también un tabulador o el carácter dos puntos. El problema está en que los separadores dentro de las cadenas no deben confundirnos. Por ejemplo, la coma en `"tierra, luna"` no debe ser interpretada como un separador.

He aqui una solución usando RD:

```
#!/usr/local/bin/perl5.8.0 -w
use strict;
use Parse::RecDescent;
use Data::Dumper;

my $grammar = q{
  line : <leftop: value sep value>
  value: /"([^"]*)"/ { $1; }
  sep : <skip:""> /[:,\t]/
};

```

```
12 my $parse = Parse::RecDescent->new($grammar);
13
14 my $line;
15 while ($line = <>) {
16   print "$line\n";
17   my $result = $parse->line($line);
18   print Dumper($result);
19 }
```

La directiva <skip> usada en la línea 9 permite redifinir lo que el analizador entiende por “blancos”: los símbolos sobre los que saltar. Por defecto el tabulador es un “blanco”. En este ejemplo, cuando estamos reconociendo un separador no queremos que sea así. La directiva <skip:""> hace que ningún símbolo sea considerado como blanco.

La directiva usada en la línea 7 <leftop: value sep value> especifica una lista asociativa a izquierdas de elementos de tipo value separados por elementos del lenguaje denotado por sep. El valor retornado es un array anónimo conteniendo la lista de elementos. En general, si el separador es especificado como una cadena, la directiva no retorna el separador como parte de la lista. Por ejemplo, en el código:

```
list: '(' <leftop: list_item ',' list_item> ')' { $return = $item[2] }
```

el valor devuelto es una referencia a la lista, la cual no incluye las comas. Sin embargo, cuando el separador viene dado por una variable sintáctica o bien por medio de una expresión regular, las distintas apariciones del separador se incluyen en la lista devuelta. Este es el caso en nuestro ejemplo.

Veamos primero el fichero de entrada para la ejecución:

```
$ cat file.txt
"xx":"yyy":"zzz"
"xx","yyy","zzz"
"xx"    "yyy"   "zzz"
"xx"    "yyy","zzz"
"xx":"yyy","zzz"
"x:x":"y,y","z  zz"
"x:x":"y,y","z  zz"."ttt"
```

La última fila usa el punto como separador del último campo. La ejecución de Dumper($result) en la línea 18 nos muestra las listas formadas:

```
$ ./commaseparated.pl file.txt
"xx":"yyy":"zzz"
$VAR1 = [ 'xx', ':', 'yyy', ':', 'zzz' ];
"xx","yyy","zzz"
$VAR1 = [ 'xx', ',', 'yyy', ',', 'zzz' ];
"xx"    "yyy"   "zzz"
$VAR1 = [ 'xx', '     ', 'yyy', '     ', 'zzz' ];
"xx"    "yyy","zzz"
$VAR1 = [ 'xx', '     ', 'yyy', ',', 'zzz' ];
"xx":"yyy","zzz"
$VAR1 = [ 'xx', ':', 'yyy', ',', 'zzz' ];
"x:x":"y,y","z  zz"
$VAR1 = [ 'x:x', ':', 'y,y', ',', 'z  zz' ];
"x:x":"y,y","z  zz"."ttt"
$VAR1 = [ 'x:x', ':', 'y,y', ',', 'z  zz' ];
```

La última entrada "x:x":"y,y","z zz"."ttt" muestra como el analizador se detiene en la cadena máximal "x:x":"y,y","z zz" que pertenece al lenguaje.

Ejercicio 13.5.1. *La última salida del programa anterior no produce un rechazo de la cadena de entrada. Se limita a detenerse en la cadena máximal* `"x:x":"y,y","z  zz"` *que puede aceptar. Si queremos que la conducta sea de rechazo, se puede hacer que sea la línea completa la que se empareje. Escriba una nueva versión del programa anterior que recoja esta especificación.*

Ejercicio 13.5.2. *Escriba una variante del ejercicio anterior en la que se fuerze a que todos los separadores en una misma línea sean iguales (aunque pueden ser diferentes en diferentes líneas).*

13.6. Las directivas `rulevar` y `reject`

En los apuntes de Conway del "Advanced Perl Parsing" realizado durante el Deutscher Perl-Workshop 3.0 en el año 2001 se plantea la siguiente variante del problema de los valores separados por comas: los campos no están delimitados por dobles comillas y el separador es aquél que separa la línea en un mayor número de elementos, o lo que es lo mismo, aquél separador (de entre un grupo fijo `[:,\t]`) que se repite mas veces en la línea. Por ejemplo, la línea:

```
a:1,b:2,c:3,d
```

se separará por comas produciendo la lista `["a:1", "b:2","c:3","d"]`, pero la línea

```
a:1,b:2,c:3:d
```

se separará mediante el carácter dos puntos produciendo la lista `["a","1,b", "2,c","3","d"]`.

Para resolver el problema se declaran dos variables `max` y `$maxcount` locales al método asociado con la variable sintáctica `line` que reconoce la categoría gramatical línea. Para declarar tales variables se usa la directiva `rulevar`.

La otra directiva que se usa es `reject`, la cuál hace que la producción falle (exactamente igual que si la acción hubiese retornado `undef`). La estrategia consiste en almacenar en, separador por separador llevar en `$maxcount` el máximo número de items en que se descompone la línea y en `$max` la referencia al array anónimo "ganador".

```
#!/usr/local/bin/perl5.8.0 -w
use strict;
use Parse::RecDescent;
use Data::Dumper;

#$::RD_TRACE = 1;
my $grammar = q{
  line : <rulevar: $max>
  line : <rulevar: $maxcount = 0>
  line :  <leftop: value ',' value>
           { $maxcount = @{$item[1]}; $max = $item[1];
             print "maxcount[,] = $maxcount\n"; }
           <reject>
       | <leftop: datum ':' datum>
           { if (@{$item[1]} > $maxcount) {
               $maxcount = @{$item[1]}; $max = $item[1]; }
             print "maxcount[,:] = $maxcount\n";
           }
           <reject>
       |  <leftop: field ";" field>
           { if (@{$item[1]} > $maxcount) {
               $maxcount = @{$item[1]}; $max = $item[1]; }
             print "maxcount[,:;] = $maxcount\n";
           }
```

```
            <reject>
        | { $return = $max; }

    value: /[^,]*/
    datum: /[^:]*/
    field: /[^;]*/
  };

  my $parse = Parse::RecDescent->new($grammar);
  my $line;
  while ($line = <>) {
    print "$line\n";
    my $result = $parse->line($line);
    if (defined($result)) { print Dumper($result); }
    else { print "Cadena no válida\n"; }
  }
```

Probemos el código con el siguiente fichero de entrada:

```
 cat file3.txt
1:2:3,3:4;44
1,2:3,3:4;44
1;2:3;3:4;44
```

He aqui una ejecución:

```
$ ./maxseparator.pl file3.txt
1:2:3,3:4;44

maxcount[,] = 2
maxcount[,:] = 4
maxcount[,:;] = 4
$VAR1 = [ '1', '2', '3,3', '4;44 ' ];
1,2:3,3:4;44

maxcount[,] = 3
maxcount[,:] = 3
maxcount[,:;] = 3
$VAR1 = [ '1', '2:3', '3:4;44 ' ];
1;2:3;3:4;44

maxcount[,] = 1
maxcount[,:] = 3
maxcount[,:;] = 4
$VAR1 = [ '1', '2:3', '3:4', '44 ' ];
```

13.7. Utilizando `score`

La directiva `<score: ...>` toma como argumento una expresión numérica que indica la prioridad de la producción. La directiva lleva un `<reject>` asociado. La producción con puntuación mas alta es elegida (salvo que hubiera otra que hubiera aceptado directamente). El problema anterior puede ser resuelto como sigue:

```
#!/usr/local/bin/perl5.8.0 -w
use strict;
```

```
use Parse::RecDescent;
use Data::Dumper;

#$::RD_TRACE = 1;
my $grammar = q{
  line : <leftop: value ',' value>
           <score: @{$item[1]}>
       | <leftop: datum ':' datum>
           <score: @{$item[1]}>
       | <leftop: field ";" field>
           <score: @{$item[1]}>

  value: /[^,]*/
  datum: /[^:]*/
  field: /[^;]*/
};

my $parse = Parse::RecDescent->new($grammar);

my $line;
while ($line = <>) {
  print "$line\n";
  my $result = $parse->line($line);
  if (defined($result)) { print Dumper($result); }
  else { print "Cadena no válida\n"; }
}
```

Al darle como entrada el fichero

```
$ cat file3.txt
1:2:3,3:4;44
1,2:2,3:3;33
1;2:2;3:3;44
```

Obtenemos la ejecución:

```
$ ./score.pl file3.txt
1:2:3,3:4;44
$VAR1 = [ '1', '2', '3,3', '4;44 ' ];
1,2:2,3:3;33
$VAR1 = [ '1', '2:2', '3:3;33 ' ];
1;2:2;3:3;44
$VAR1 = [ '1', '2:2', '3:3', '44 ' ];
```

13.8. Usando `autoscore`

La solución anterior puede simplificarse usando `autoscore`. Si una directiva `<autoscore>` aparece en cualquier producción de una variable sintáctica, el código especificado por la misma es utilizado como código para la puntuación de la producción dentro de las producciones de la variable sintáctica. Con la excepción, claro está, de aquellas reglas que tuvieran ya una directiva `score` explícitamente asignada. El código de la gramática queda como sigue:

```
my $grammar = q{
  line : <autoscore: @{$item[1]}>
       | <leftop: value ',' value>
```

```
       | <leftop: datum ':' datum>
       | <leftop: field ";" field>

  value: /[^,]*/
  datum: /[^:]*/
  field: /[^;]*/
};
```

Por defecto, el analizador generado por `parse::RecDescent` siempre acepta la primera producción que reconoce un prefijo de la entrada. Como hemos visto, se puede cambiar esta conducta usando la directiva `<score: ...>`. Las diferentes producciones son intentadas y su puntuación (`score`) evaluada, considerándose vencedora aquella con mayor puntuación. Esto puede usarse para hacer que la regla vencedora sea "la mas larga", en el sentido de ser la que mas elementos en `@item` deja. En este caso tomamos ventaja de la directiva `<autoscore>`, la cual permite asociar una directiva `<score>` con cada producción que no tenga ya una directiva `<score>` explícita.

```
#!/usr/local/bin/perl5.8.0 -w
use strict;
use warnings;
use Parse::RecDescent;

$::RD_TRACE = 1;
#$::RD_HINT = 1;
my $grammar = q {

  start:  seq_1 ';'
  seq_1     : <autoscore: @item>
            | 'A' 'B' 'C' 'D'
               { local $^W = 0;
                 print "seq_1: " . join (" ", @item[1..$#item]) . " score: $score\n"
               }
            |  'A' 'B' 'C' 'D' 'E' 'F'
               { print "seq_1: " . join (" ", @item[1..$#item]) . " score: $score\n" }
};

my $parser;

{
  local $^W = 0; # Avoid warning "uninitialized value in (m//) at RecDescent.pm line 626."
  $parser=Parse::RecDescent->new($grammar);
}

my $result = $parser->start("A B C D E F ;");
if (defined($result)) {
  print "Valida\n";
} else { print "No reconocida\n"; }
```

13.9. El Hash `%item`

En el contexto de una regla de producción, el hash `%item` proporciona un acceso indexado en los símbolos a los correspondientes atributos. Así, si tenemos la regla `A : B C D`, el elemento `$item{B}` hace alusión al atributo de `B` e `$item{C}` hace alusión al atributo de `C`. Los símbolos correspondientes a cadenas, expresiones regulares, directivas y acciones se guardan respectivamente bajo una clave de la forma `__STRINGn__`, `__PATTERNn__`, `__DIRECTIVEn__`, `__ACTIONn__`, donde `n` indica la posición ordinal del elemento dentro de los de su tipo en la regla. Así, `__PATTERN2__` hace alusión a la segunda expresión regular en la parte derecha de la regla.

El elemento especial `$item{__RULE__}` contiene el nombre de la regla actual.

La ventaja de usar `%item` en vez de `@items` es que evita los errores cometidos al introducir o eliminar elementos en la parte derecha de una producción. Cuando esto ocurre, el programador debe cambiar los números de los índices en la acción.

Una limitación del hash `%item` es que, cuando hay varias apariciones de una variable sintáctica sólo guarda el valor de la última aparición. Por ejemplo:

```
range: '(' number '..' number )' { $return = $item{number} }
```

retorna en `$item{number}` el valor correspondiente a la segunda aparición de la variable sintáctica `number`.

13.10. Usando la directiva autotree

La directiva `<autotree>` hace que RD construya automáticamente una representación del árbol sintáctico concreto. Cada nodo del árbol, correspondiendo a una parte derecha de una regla de producción es bendecido (usando `bless`) en una clase cuyo nombre es el de la variable sintáctica que produce: viene a ser equivalente a `{ bless \%item, $item[0] }`.

```
$ cat ./infixautotree.pl
#!/usr/local/bin/perl5.8.0 -w
use strict;
use Parse::RecDescent;
use Data::Dumper;

#$::RD_TRACE = 1;
my $grammar = q{
  <autotree>
  expre  : <leftop: prods '+' prods>
  prods : <leftop: unario '*' unario>
  unario: "(" expre ")" |  numero
  numero : /\d+(\.\d+)?/
};

my $parse = new Parse::RecDescent($grammar);
my $line = shift;
my $result = $parse->expre($line);
if (defined($result)) {
    print "Válida:\n", Dumper($result);
}
  else { print "Cadena no válida\n"; }
```

Las reglas que consisten en un sólo terminal, como la de número, llevan un tratamiento especial. En ese caso el código ejecutado es:

```
{ bless {__VALUE__=>$item[1]}, $item[0] }
```

La estructura resultante es relativamente compleja. Cada nodo es bendecido en la clase con nombre el de la variable sintáctica, esto es, con nombre el de la regla. Podemos sacar ventaja de esta aproximación si asociamos métodos de procesamiento con cada tipo de nodo. Por ejemplo, para hacer un recorrido del árbol sintáctico, podríamos extender el ejemplo anterior como sigue:

```
$ cat infixautotree2.pl
...

sub expre::traverse {
  my $root = shift;
  my $level = shift;

  process($root, $level, '__RULE__');
  foreach (@{$root->{__DIRECTIVE1__}}) {
    $_->traverse($level+1);
  }
}

sub prods::traverse {
  my $root = shift;
  my $level = shift;

  process($root, $level, '__RULE__');
  foreach (@{$root->{__DIRECTIVE1__}}) {
    $_->traverse($level+1);
  }
}

sub unario::traverse {
  my $root = shift;
  my $level = shift;

  process($root, $level, '__RULE__');
  my $child = $root->{numero} || $root->{expre};
  $child->traverse($level+1);
}

sub numero::traverse {
  my $root = shift;
  my $level = shift;

  process($root, $level, '__VALUE__');
}

sub process {
  my $root = shift;
  my $level = shift;
  my $value = shift;

  print " "x($width*$level),$root->{$value},"\n";
}
```

Sigue un ejemplo de ejecución:

```
$ ./infixautotree2.pl '2+3*(5+2)+9' 6
Válida:
expre
      prods
            unario
```

```
                  2
      prods
            unario
                  3
            unario
                  expre
                        prods
                              unario
                                    5
                        prods
                              unario
                                    2
      prods
            unario
                  9
```

13.11. Práctica

Reescriba la fase de análisis sintáctico del compilador para el lenguaje Tutu usando `Parse::RecDescent`. La gramática de Tutu es como sigue:

program → block
block → declarations statements | statements
declarations → declaration ';' declarations | declaration ';'
declaration → INT idlist | STRING idlist
statements → statement ';' statements | statement
statement → ID '=' expression | P expression | '{' block '}'
expression → expression '+' term | expression '-' term | term
term → term '*' factor | term '/' factor | factor
factor → '(' expression ')' | ID | NUM | STR
idlist → ID ',' idlist | ID

Las acciones deberán producir el árbol de análisis abstracto o AST. Recicle todo el código que ha escrito para las restantes fases: análisis léxico, semántico, optimización de código, cálculo de direcciones, generación de código y optimización peephole.

13.12. Construyendo un compilador para Parrot

Las ideas y el código en esta sección están tomadas del artículo de Dan Sugalski *Building a parrot Compiler* que puede encontrarse en http://www.onlamp.com/lpt/a/4725.

```
#!/usr/local/bin/perl5.8.0 -w
#
# This program created 2004, Dan Sugalski. The code in this file is in
# the public domain--go for it, good luck, don't forget to write.
use strict;
use Parse::RecDescent;
use Data::Dumper;

# Take the source and destination files as parameters
my ($source, $destination) = @ARGV;

my %global_vars;
my $tempcount = 0;
```

```
my (%temps) = (P => 0,
         I => 0,
         N => 0,
         S => 0
        );

# AUTOACTION simplifies the creation of a parse tree by specifying an action
# for each production (ie action is { [@item] })
$::RD_AUTOACTION = q{ [@item] };

my $grammar = <<'EOG';
field: /\b\w+\b/

stringconstant: /'[^']*'/ |
    /"[^"]*"/
#"
float: /[+-]?(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?/

constant: float | stringconstant

addop: '+' | '-'
mulop: '*' | '/'
modop: '%'
cmpop: '<>' | '>='| '<=' | '<' | '>' | '='
logop: 'and' | 'or'

parenexpr: '(' expr ')'

simplevalue: parenexpr | constant | field

modval: <leftop: simplevalue modop simplevalue>

mulval: <leftop: modval mulop modval>

addval: <leftop: mulval addop mulval>

cmpval: <leftop: addval cmpop addval>

logval: <leftop: cmpval logop cmpval>

expr: logval

declare: 'declare' field

assign: field '=' expr

print: 'print' expr

statement: assign | print | declare
EOG

# ?? Makes emacs cperl syntax highlighting mode happier
my $parser = Parse::RecDescent->new($grammar);
```

La gramatica categoriza las prioridades de cada una de las operaciones: categorías próximas al símbolo de arranque tienen menos prioridad que aquellas mas lejanas.

```
68  my @nodes;
69  open SOURCE, "<$source" or die "Can't open source program ($!)";
70
71  while (<SOURCE>) {
72      # Strip the trailing newline and leading spaces. If the line is
73      # blank, then skip it
74      chomp;
75      s/^\s+//;
76      next unless $_;
77
78      # Parse the statement and throw an error if something went wrong
79      my $node = $parser->statement($_);
80      die "Bad statement" if !defined $node;
81
82      # put the parsed statement onto our list of nodes for later treatment
83      push @nodes, $node;
84  }
85
86  print Dumper(\@nodes);
87  #exit;
88
89  # At this point we have parsed the program and have a tree of it
90  # ready to process. So lets do so. First we set up our node handlers.
91
```

El programa principal lee una línea del fuente y construye el árbol (línea 79). Los árboles se van guardando en la lista `@nodes`. El paso siguiente es la generación de código:

```
# At this point we have parsed the program and have a tree of it
# ready to process. So lets do so. First we set up our node handlers.

my (%handlers) = (addval => \&handle_generic_val,
      assign => \&handle_assign,
      cmpval => \&handle_generic_val,
      constant => \&delegate,
      declare => \&handle_declare,
      expr => \&delegate,
      field => \&handle_field,
      float => \&handle_float,
      logval => \&handle_generic_val,
      modval => \&handle_generic_val,
      mulval => \&handle_generic_val,
      negfield => \&handle_negfield,
      parenexpr => \&handle_paren_expr,
      print => \&handle_print,
      simplevalue => \&delegate,
      statement => \&delegate,
      stringconstant => \&handle_stringconstant,
     );
```

El autor ha optado por escribir un manipulador para cada tipo de nodo. Es algo similar a lo que hicimos usando métodos y herencia para el compilador de Tutu. Algunos nodos simplemente delegan y otros recurren a un manipulador genérico.

La fase de generación de código comienza por la escritura de un preámbulo y termina con la escritura de un pie requeridos por el intérprete. En medio se sitúa el código correspondiente a la traducción de los nodos provenientes de las diversas líneas del fuente:

```
# Open the output file and emit the preamble
open PIR, ">$destination" or die "Can't open destination ($!)";
print PIR <<HEADER;
.sub __MAIN prototyped
  .param pmc argv
HEADER

foreach my $node (@nodes) {
    my @lines = process_node(@$node);
    print PIR join("", @lines);
}

print PIR <<FOOTER;
  end
.end
FOOTER
```

La subrutina `process_node` hace un recorrido de los árboles de análisis, llamando a los manipuladores de los nodos que están siendo visitados. El elemento 0 del array `elems` identifica la clase de nodo. Así la llamada `$handlers{$elems[0]}->(@elems)` produce una llamada al manipulador correspondiente, pasándole como argumento los hijos del nodo.

```
# The value of the last expression evaluated
sub last_expr_val {
    return $::last_expr;
}

# Setting the last expression evaluated's value
sub set_last_expr_val {
    $::last_expr = $_[0];
}

sub process_node {
  my (@elems) = @_;
  return "\n" unless @elems;
  return "\n" unless defined($elems[0]);
  if (ref $elems[0]) {
    return process_node(@{$elems[0]});
  } elsif (exists($handlers{$elems[0]})) {
    return $handlers{$elems[0]}->(@elems);
  } else {
    return "***", $elems[0], "***\n";
  }
}
```

A continuación siguen los diversos manipuladores para los diferentes tipos de nodo:

```
sub handle_assign {
    my ($nodetype, $destvar, undef, $expr) = @_;
    my @nodes;
    push @nodes, process_node(@$expr);
```

```
    my $rhs = last_expr_val();
    push @nodes, process_node(@$destvar);
    my $lhs = last_expr_val();
    push @nodes, "  $lhs = $rhs\n";
    return @nodes;
}

sub handle_declare {
    my ($nodetype, undef, $var) = @_;
    my @lines;

    my $varname = $var->[1];

    # Does it exist?
    if (defined $global_vars{$varname}) {
      die "Multiple declaration of $varname";
    }
    $global_vars{$varname}++;
    push @lines, "  .local pmc $varname\n";
    push @lines, "  new $varname, .PerlInt\n";
    return @lines;
}

sub handle_field {
    my ($nodetype, $fieldname) = @_;
    if (!exists $global_vars{$fieldname}) {
      die "undeclared field $fieldname used";
    }
    set_last_expr_val($fieldname);
    return;
}

sub handle_float {
    my ($nodetype, $floatval) = @_;
    set_last_expr_val($floatval);
    return;
}

sub handle_generic_val {
  my (undef, $terms) = @_;
  my (@terms) = @$terms;

  # Process the LHS
  my $lhs = shift @terms;
  my @tokens;
  push @tokens, process_node(@$lhs);

  my ($op, $rhs);

  # Now keep processing the RHS as long as we have it
  while (@terms) {
      $op = shift @terms;
      $rhs = shift @terms;
```

```
my $val = last_expr_val();
my $oper = $op->[1];

push @tokens, process_node(@$rhs);
my $other_val = last_expr_val();

my $dest = $temps{P}++;

foreach ($oper) {
  # Simple stuff -- addition, subtraction, multiplication,
  # division, and modulus. Just a quick imcc transform
  /(\+|\-|\*|\/|%)/ && do { push @tokens, "new \$P$dest, .PerlInt\n";
          push @tokens, "\$P$dest = $val $oper $other_val\n";
          set_last_expr_val("\$P$dest");
          last;
        };
  /and/ && do { push @tokens, "new \$P$dest, .PerlInt\n";
    push @tokens, "\$P$dest = $val && $other_val\n";
    set_last_expr_val("\$P$dest");
    last;
        };
  /or/ && do { push @tokens, "new \$P$dest, .PerlInt\n";
         push @tokens, "\$P$dest = $val || $other_val\n";
         set_last_expr_val("\$P$dest");
         last;
       };
  /<>/ && do { my $label = "eqcheck$tempcount";
         $tempcount++;
         push @tokens, "new \$P$dest, .Integer\n";
         push @tokens, "\$P$dest = 1\n";
         push @tokens, "ne $val, $other_val, $label\n";
         push @tokens, "\$P$dest = 0\n";
         push @tokens, "$label:\n";
         set_last_expr_val("\$P$dest");
         last;
       };
  /=/ && do { my $label = "eqcheck$tempcount";
    $tempcount++;
    push @tokens, "new \$P$dest, .Integer\n";
    push @tokens, "\$P$dest = 1\n";
    push @tokens, "eq $val, $other_val, $label\n";
    push @tokens, "\$P$dest = 0\n";
    push @tokens, "$label:\n";
    set_last_expr_val("\$P$dest");
    last;
  };
  /</ && do { my $label = "eqcheck$tempcount";
    $tempcount++;
    push @tokens, "new \$P$dest, .Integer\n";
    push @tokens, "\$P$dest = 1\n";
    push @tokens, "lt $val, $other_val, $label\n";
    push @tokens, "\$P$dest = 0\n";
    push @tokens, "$label:\n";
```

```
          set_last_expr_val("\$P$dest");
          last;
        };
        />/ && do { my $label = "eqcheck$tempcount";
          $tempcount++;
          push @tokens, "new \$P$dest, .Integer\n";
          push @tokens, "\$P$dest = 1\n";
          push @tokens, "gt $val, $other_val, $label\n";
          push @tokens, "\$P$dest = 0\n";
          push @tokens, "$label:\n";
          set_last_expr_val("\$P$dest");
          last;
        };
        die "Can't handle $oper";
      }
  }
  return @tokens;
}

sub handle_paren_expr {
    my ($nodetype, undef, $expr, undef) = @_;
    return process_node(@$expr);
}

sub handle_stringconstant {
    my ($nodetype, $stringval) = @_;
    set_last_expr_val($stringval);
    return;
}

sub handle_print {
    my ($nodetype, undef, $expr) = @_;
    my @nodes;
    push @nodes, process_node(@$expr);
    my $val = last_expr_val();
    push @nodes, "  print $val\n";
    return @nodes;
}

sub delegate {
    my ($nodetype, $nodeval) = @_;
    return process_node(@$nodeval);
}
```

El fichero fuente `foo.len`:

```
declare foo
declare bar

foo = 15
bar = (foo+8)*32-7

print bar
print "\n"
```

```
print foo % 10
print "\n"
```

Compilamos:

```
$ ./compiler.pl foo.len foo.pir
```

Esto produce por pantalla un volcado de los árboles de als diferentes sentencias. Asi para `declare foo`:

```
$VAR1 = [ [ 'statement', [ 'declare', 'declare', [ 'field', 'foo' ] ] ],
```

para la sentencia `foo = 15` el árbol es:

```
[ 'statement',
  [ 'assign',
    [
      'field',
      'foo'
    ],
    '=',
    [ 'expr',
      [ 'logval',
        [
          [ 'cmpval',
            [
              [ 'addval',
                [
                  [ 'mulval',
                    [
                      [ 'modval',
                        [
                          [ 'simplevalue',
                            [ 'constant',
                              [
                                'float',
                                '15'
                              ]
] ] ] ] ] ] ] ] ] ] ] ] ] ] ],
```

Este es el árbol de la sentencia `print bar`:

```
[ 'statement',
  [ 'print',
    'print',
    [ 'expr',
      [ 'logval',
        [
          [ 'cmpval',
            [
              [ 'addval',
                [
                  [ 'mulval',
                    [
                      [ 'modval',
                        [
```

```
                          [ 'simplevalue',
                            [
                              'field',
                              'bar'
] ] ] ] ] ] ] ] ] ] ] ] ] ] ] ],
```

Además de los árboles presentados en la salida estándar, se produce como salida el fichero `foo.pir` conteniendo el código parrot intermedio:

```
$ cat foo.pir
.sub __MAIN prototyped
  .param pmc argv
  .local pmc foo
  new foo, .PerlInt
  .local pmc bar
  new bar, .PerlInt
  foo = 15
new $P0, .PerlInt
$P0 = foo + 8
new $P1, .PerlInt
$P1 = $P0 * 32
new $P2, .PerlInt
$P2 = $P1 - 7
  bar = $P2
  print bar
  print "\n"
new $P3, .PerlInt
$P3 = foo % 10
  print $P3
  print "\n"
  end
.end
```

Antes de ejecutarlo veamos las opciones de `parrot`:

```
$ parrot -h
parrot [Options] <file>
  Options:
    -h --help
    -V --version
   <Run core options>
    -b --bounds-checks|--slow-core
    -C --CGP-core
    -f --fast-core
    -g --computed-goto-core
    -j --jit-core
    -p --profile
    -P --predereferenced-core
    -S --switched-core
    -t --trace
   <VM options>
    -d --debug[=HEXFLAGS]
       --help-debug
    -w --warnings
    -G --no-gc
```

```
        --gc-debug
        --leak-test|--destroy-at-end
    -. --wait    Read a keystroke before starting
   <Compiler options>
    -v --verbose
    -E --pre-process-only
    -o --output=FILE
       --output-pbc
    -O --optimize[=LEVEL]
    -a --pasm
    -c --pbc
    -r --run-pbc
    -y --yydebug
   <Language options>
       --python
see docs/running.pod for more
```

Con la opción `-o` podemos producir un fichero en formato `pbc`:

```
$ parrot -o foo.pbc foo.pir
```

que podemos ejecutar con el depurador `pdb` (para construirlo en el momento de la instalación de Parrot deberás hacer `make pdb`).

```
$ pdb foo.pbc
Parrot Debugger 0.0.2

(pdb) h
List of commands:
    disassemble -- disassemble the bytecode
    load -- load a source code file
    list (l) -- list the source code file
    run (r) -- run the program
    break (b) -- add a breakpoint
    watch (w) -- add a watchpoint
    delete (d) -- delete a breakpoint
    disable -- disable a breakpoint
    enable  -- reenable a disabled breakpoint
    continue (c) -- continue the program execution
    next (n) -- run the next instruction
    eval (e) -- run an instruction
    trace (t) -- trace the next instruction
    print (p) -- print the interpreter registers
    stack (s) -- examine the stack
    info -- print interpreter information
    quit (q) -- exit the debugger
    help (h) -- print this help

Type "help" followed by a command name for full documentation.
```

Veamos el programa traducido:

```
(pdb) list 1 17
1  new_p_ic P16,32
2  new_p_ic P30,32
3  set_p_ic P16,15
```

```
4  new_p_ic P29,32
5  add_p_p_ic P29,P16,8
6  new_p_ic P28,32
7  mul_p_p_ic P28,P29,32
8  new_p_ic P29,32
9  sub_p_p_ic P29,P28,7
10  set_p_p P30,P29
11  print_p P30
12  print_sc "\n"
13  new_p_ic P30,32
14  mod_p_p_ic P30,P16,10
15  print_p P30
16  print_sc "\n"
17  end
```

Procedemos a ejecutarlo:

```
(pdb) n
2  new_p_ic P30,32
(pdb)
3  set_p_ic P16,15
(pdb)
4  new_p_ic P29,32
(pdb)
5  add_p_p_ic P29,P16,8
(pdb) p P16
P16 = [PerlInt]
Stringified: 15
5  add_p_p_ic P29,P16,8
(pdb) c
729
5
Program exited.
(pdb) quit
$
```

13.13. Práctica

Extienda el lenguaje Tutu para que comprenda el tipo lógico, declarado mediante la palabra reservada `bool` y operaciones de comparación como las que figuran en el ejemplo propuesto por Sugalski. Genere código para la máquina basada en registros. Extienda el juego de instrucciones de manera apropiada.

13.14. Práctica

Modifique la fase de generación de código del compilador de Tutu para que produzca código para la máquina Parrot. Modifique el optimizador peephole para que se adapte al código Parrot.

Capítulo 14

Análisis LR

14.1. `Parse::Yapp`: Ejemplo de Uso

El generador de analizadores sintácticos `Parse::Yapp` es un analizador LALR inspirado en `yacc`. El módulo `Parse::Yapp` no viene con la distribución de Perl, por lo que es necesario bajarlo desde CPAN, en la dirección

`\http://search.cpan.org/~fdesar/Parse-Yapp-1.05/lib/Parse/Yapp.p`

o bien en nuestros servidores locales, por ejemplo en el mismo directorio en que se guarda la versión HTML de estos apuntes encontrará una copia de Parse-Yapp-1.05.tar.gz. La versión a la que se refiere este capítulo es la 1.05.

Para ilustrar su uso veamos un ejemplo en el que se genera una sencilla calculadora numérica. Los contenidos del programa `yapp` los hemos guardado en un fichero denominado `Calc.yp` (el código completo figura en el apéndice en la página 467)

```
#
# Calc.yp
#
# Parse::Yapp input grammar example.
#
# This file is PUBLIC DOMAIN
#
#
```

Se pueden poner comentarios tipo Perl o tipo C (`/* ... */`) a lo largo del fichero.

```
%right  '='
%left   '-' '+'
%left   '*' '/'
%left   NEG
%right  '^'

%%
input:  #empty
     |   input line  { push(@{$_[1]},$_[2]); $_[1] }
;
```

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.* Véase la sección 14.7 para mas detalles.

Un programa yapp 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 15 separa la cabeza del cuerpo. 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 yapp 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.

En `Parse::Yapp` las acciones son convertidas en subrutinas anónimas. Mas bien en métodos anónimos. Así pues el primer argumento de la subrutina se identifica con una referencia al analizador (`$_[0]`). Los restantes parámetros se corresponden con los *atributos de los símbolos* en la parte derecha de la regla de producción (`$_[1]` ...). Por ejemplo, el código en la línea 21 imprime el atributo asociado con la variable sintáctica `expr`, que en este caso es su valor numérico.

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. Asi pues el atributo retornado por `input` es una referencia a una lista conteniendo los valores de las expresiones evaluadas.

Para saber mas sobre las estructuras internas de `yapp` para la representación de las acciones asociadas con las reglas véase la sección 14.4.

```
19
20  line:  '\n'       { 0 }
21      |  exp '\n'   { print "$_[1]\n"; $_[1] }
22      |  error '\n' { $_[0]->YYErrok; 0 }
23  ;
```

El terminal `error` en la línea 22 esta asociado con la aparición de un error. El tratamiento es el mismo que en `yacc`. Cuando se produce un error en el análisis, `yapp` emite un mensaje de error y produce "mágicamente" un terminal especial denominado `error`. A partir de ahí permanecerá silencioso, consumiendo terminales hasta encontrar uno de los terminales que le hemos indicado en las reglas de recuperación de errores, en este caso, cuando encuentre un retorno de carro. Como se ha dicho, en `Parse::Yapp` el primer argumento de la acción denota al analizador sintáctico. Así pues el código `$_[0]->YYErrok` es una llamada al método `YYErrok` del analizador. Este método funciona como la macro `yyerrok` de `yacc`, indicando que la presencia del retorno del carro (`\n`) la podemos considerar un signo seguro de que nos hemos recuperado del error. A partir de este momento, `yapp` volverá a emitir mensajes de error. Para saber más sobre la recuperación de errores en `yapp` léase la sección 14.15.

```
24
25  exp:        NUM
```

La acción por defecto es retornar `$_[1]`. Por tanto, en este caso el valor retornado es el asociado a `NUM`.

```
26          |   VAR           { $_[0]->YYData->{VARS}{$_[1]} }
```

El método `YYData` provee acceso a un hash que contiene los datos que están siendo analizados. En este caso creamos una entrada `VARS` que es una referencia a un hash en el que guardamos las variables. Este hash es la tabla de símbolos de la calculadora.

```
27          |   VAR '=' exp   { $_[0]->YYData->{VARS}{$_[1]}=$_[3] }
28          |   exp '+' exp   { $_[1] + $_[3] }
29          |   exp '-' exp   { $_[1] - $_[3] }
30          |   exp '*' exp   { $_[1] * $_[3] }
```

Hay numerosas ambiguedades en esta gramática. Por ejemplo,

- ¿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 las líneas 9-13. Al declarar como asociativo a izquierdas al terminal `-` (línea 10) hemos resuelto este tipo de ambiguedad. 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.

```
31          |   exp '/' exp   {
32                              $_[3]
33                              and return($_[1] / $_[3]);
34                              $_[0]->YYData->{ERRMSG}
35                                =   "Illegal division by zero.\n";
36                              $_[0]->YYError;
37                              undef
38                            }
```

En la regla de la división comprobamos que el divisor es distinto de cero. Si es cero inicializamos el atributo `ERRMSG` en la zona de datos con el mensaje de error apropiado. Este mensaje es aprovechado por la subrutina de tratamiento de errores (véase la subrutina `_Error` en la zona de la cola). La subrutina `_Error` es llamada automáticamente por `yapp` cada vez que ocurre un error sintáctico. Esto es asi por que en la llamada al analizador se especifican quienes son las diferentes rutinas de apoyo:

```
my $result = $self->YYParse( yylex => \&_Lexer,
                             yyerror => \&_Error,
                             yydebug => 0x0 );
```

Por defecto, *una regla de producción tiene la prioridad del último terminal que aparece en su parte derecha*. 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.

```
39          |   '-' exp %prec NEG   { -$_[2] }
```

¿Cual es la ambiguedad que surge con esta regla? Una de las 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 `-` 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 $E \rightarrow -E$ y $E \rightarrow E - E$ tienen, en principio la prioridad del terminal `-`, el cual fué declarado en la línea 10. La prioridad expresada explícitamente para la regla por la declaración `%prec NEG` de la línea 39 hace que la regla tenga la prioridad del terminal `NEG` (línea 12) y por tanto mas prioridad que el terminal `-`. Esto hará que `yapp` finalmente opte por la regla `exp: '-' exp`.

La declaración de `^` como asociativo a derechas y con un nivel de prioridad alto resuelve las ambiguedades relacionadas con este operador:

```
40          |   exp '^' exp         { $_[1] ** $_[3] }
41          |   '(' exp ')'         { $_[2] }
42  ;
```

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. 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;
    my $result = $self->YYParse( yylex => \&_Lexer,
                                 yyerror => \&_Error,
                                 yydebug => 0x0 );
    my @result = @$result;
    print "@result\n";
}
```

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}`.

```
# rutinas de apoyo
%%

sub _Error {
  exists $_[0]->YYData->{ERRMSG}
    and do {
        print $_[0]->YYData->{ERRMSG};
        delete $_[0]->YYData->{ERRMSG};
        return;
    };
  print "Syntax error.\n";
}

```

A continuación sigue el método que implanta el análisis léxico `_Lexer`. En primer lugar se comprueba la existencia de datos en `parser->YYData->{INPUT}`. Si no es el caso, los datos se tomarán de la entrada estándar:

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

  $parser->YYData->{INPUT}
  or  $parser->YYData->{INPUT} = <STDIN>
  or  return('',undef);

```

Cuando el analizador léxico alcanza el final de la entrada debe devolver la pareja `('',undef)`.

Eliminamos los blancos iniciales (lo que en inglés se conoce por *trimming*):

```
  $parser->YYData->{INPUT}=~s/^[ \t]//;

```

A continuación vamos detectando los números, identificadores y los símbolos individuales. El bucle `for ($parser->YYData->{INPUT})` se ejecuta mientras la cadena en `$parser->YYData->{INPUT}` no sea vacía, lo que ocurrirá cuando todos los terminales hayan sido consumidos.

```
65    for ($parser->YYData->{INPUT}) {
66      s/^([0-9]+(?:\.[0-9]+)?)//
67        and return('NUM',$1);
68      s/^([A-Za-z][A-Za-z0-9_]*)//
69        and return('VAR',$1);
70      s/^(.)//s
71        and return($1,$1);
72    }
73  }
```

Ejercicio 14.1.1. *1. ¿Quién es la variable índice en la línea 65?*

2. ¿Sobre quién ocurre el binding en las líneas 66, 68 y 70?

3. ¿cual es la razón por la que `$parser->YYData->{INPUT}` *se ve modificado si no aaprece como variable para el binding en las líneas 66, 68 y 70?*

Construimos el módulo `Calc.pm` a partir del fichero `Calc.yp` especificando la gramática, usando un fichero `Makefile`:

```
> cat Makefile
Calc.pm: Calc.yp
        yapp -m Calc Calc.yp
> make
yapp -m Calc Calc.yp
```

Esta compilación genera el fichero `Calc.pm` conteniendo el analizador:

```
> ls -ltr
total 96
-rw-r-----    1 pl       users        1959 Oct 20  1999 Calc.yp
-rw-r-----    1 pl       users          39 Nov 16 12:26 Makefile
-rwxrwx--x    1 pl       users          78 Nov 16 12:30 usecalc.pl
-rw-rw----    1 pl       users        5254 Nov 16 12:35 Calc.pm
```

El script `yapp` es un *frontend* al módulo `Parse::Yapp`. Admite diversas formas de uso:

- `yapp [options]` *`grammar`*`[.yp]`

  El sufijo `.yp` es opcional.

- `yapp -`*`V`*

  Nos muestra la versión:

  ```
  $ yapp -V
  This is Parse::Yapp version 1.05.
  ```

- `yapp -`*`h`*

  Nos muestra la ayuda:

  ```
  $ yapp -h

  Usage:  yapp [options] grammar[.yp]
    or    yapp -V
    or    yapp -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::Yapp::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::Yapp and gracefully exits
-h           Display this help screen
```

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 podría haberse omitido.

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*.

Veamos los contenidos del ejecutable `usecalc.pl` el cuál utiliza el módulo generado por `yapp`:

```
> cat usecalc.pl
#!/usr/local/bin/perl5.8.0 -w

use Calc;

$parser = new Calc();
$parser->Run;
```

Al ejecutar obtenemos:

```
$ ./usecalc3.pl
2+3
5
4*8
32
^D
5 32
```

Pulsamos al final `Ctrl-D` para generar el final de fichero. El analizador devuelve la lista de valores computados la cual es finalmente impresa.

¿En que orden ejecuta `YYParse` las acciones? La respuesta es que el analizador generado por `yapp` construye una derivación a derechas inversa y ejecuta las acciones asociadas a las reglas de producción que se han aplicado. Así, para la frase `3+2` 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 de la línea 25:

```
25  exp:        NUM { $_[1]; } # acción por defecto
```

Esta instancia de `exp` tiene ahora como atributo `3`.

2. De nuevo la acción de la línea 25:

```
25  exp:        NUM { $_[1]; } # acción por defecto
```

Esta nueva instancia de `exp` tiene como atributo `2`.

3. La acción asociada con $E \rightarrow E + E$, en la línea 28:

```
28          |   exp '+' exp   { $_[1] + $_[3] }
```

La nueva instancia (nodo) `exp` tiene como atributo `5 = 3 + 2`.

Obsérvese que la antiderivación a derechas da lugar a un recorrido ascendente y a izquierdas del árbol:

```
     E(3)
    / | \
(1)E + E(2)
  /     \
NUM     NUM
```

Los números entre paréntesis indican el orden de visita de las producciones.

14.2. Conceptos Básicos

Los analizadores generador por `yapp` 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 14.2.1. *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.*

$$FSD = \left\{ \alpha \in (\Sigma \cup V)* : \exists S \underset{RM}{\overset{*}{\Longrightarrow}} \alpha \right\}$$

*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 14.2.2. *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: $handle(\alpha) = (B \to \gamma, \beta\gamma)$. La función *handle* tiene dos componentes: $handle_1(\alpha) = B \to \gamma$ 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 14.2.3. *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 14.2.4. *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, $(E \to E + E, 2)$ sería un LR(0) item para la gramática de las expresiones.*

  Conjunto de Estados:

  $Q = \{(A \to \alpha, n) : A \to \alpha \in P,\ n \leq |\alpha|\}$

  La notación $\mid \alpha \mid$ denota la longitud de la cadena $\mid \alpha \mid$. En vez de la notación $(A \to \alpha, n)$ escribiremos: $A \to \beta_\uparrow\gamma = \alpha$, donde la flecha ocupa el lugar indicado por el número $n = \mid \beta \mid$:
- *Función de transición:*

  $\delta(A \to \alpha_\uparrow X\beta, X) = A \to \alpha X_\uparrow\beta\ \forall X \in V \cup \Sigma$

  $\delta(A \to \alpha_\uparrow B\beta, \epsilon) = B \to \gamma \forall B \in V$
- *Estado de arranque: Se añade la "superregla" $S' \to S$ a la gramática $G = (\Sigma, V, P, S)$. El LR(0) item $S' \to_\uparrow S$ 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 14.2.1. *Consideremos la gramática:*

$$S \to a\ S\ b$$
$$S \to \epsilon$$

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 14.1 muestra el* autómata finito no determinista con ϵ-transiciones *(*NFA*) que reconoce los prefijos viables de esta gramática, construido de acuerdo con el algoritmo 14.2.4.*

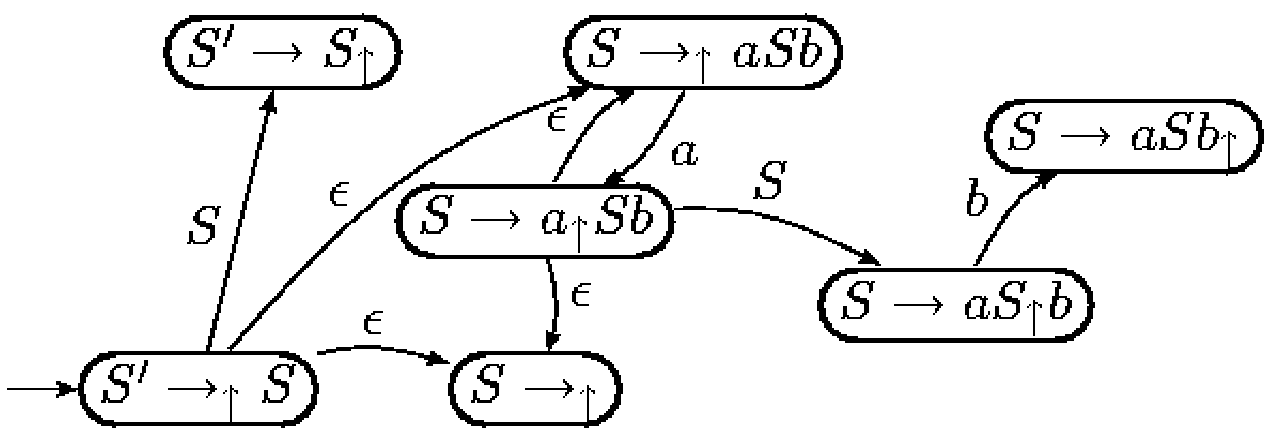

Figura 14.1: NFA que reconoce los prefijos viables

Ejercicio 14.2.1. *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?*

14.3. Construcción de las Tablas para el Análisis SLR

14.3.1. Los conjuntos de Primeros y Siguientes

Repasemos las nociones de conjuntos de *Primeros* y *siguientes*:

Definición 14.3.1. *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:

$N(\alpha) = \left\{ \begin{array}{ll} \{\epsilon\} & si\ \alpha \stackrel{*}{\Longrightarrow} \epsilon \\ \emptyset & en\ otro\ caso \end{array} \right.$

Definición 14.3.2. *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

$$E(A) = \begin{cases} \{\$\} & si\ S \stackrel{*}{\Longrightarrow} \alpha A \\ \emptyset & en\ otro\ caso \end{cases}$$

Algoritmo 14.3.1. *Construcción de los conjuntos* $FIRST(X)$

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*

$$\begin{array}{l} i = 1; \\ do \\ \quad FIRST(X) = FIRST(X) \cup FIRST(Y_i) - \{\epsilon\}; \\ \quad i++; \\ mientras\ (\epsilon \in FIRST(Y_i)\ and\ (i \leq k)) \\ si\ (\epsilon \in FIRST(Y_k)\ and\ i > k)\ FIRST(X) = FIRST(X) \cup \{\epsilon\} \end{array}$$

Este algoritmo puede ser extendido para calcular $FIRST(\alpha)$ para $\alpha = X_1 X_2 \cdots X_n \in (V \cup \Sigma)^*$.

Algoritmo 14.3.2. *Construcción del conjunto* $FIRST(\alpha)$

$$\begin{array}{l} i = 1; \\ FIRST(\alpha) = \emptyset; \\ do \\ \quad FIRST(\alpha) = FIRST(\alpha) \cup FIRST(X_i) - \{\epsilon\}; \\ \quad i++; \\ mientras\ (\epsilon \in FIRST(X_i)\ and\ (i \leq n)) \\ si\ (\epsilon \in FIRST(X_n)\ and\ i > n)\ FIRST(\alpha) = FIRST(X) \cup \{\epsilon\} \end{array}$$

Algoritmo 14.3.3. *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* $A \rightarrow \alpha B \beta$ *entonces*

$$FOLLOW(B) = FOLLOW(B) \cup (FIRST(\beta) - \{\epsilon\})$$

3. *Si* $A \rightarrow \alpha B$ *o bien* $A \rightarrow \alpha B \beta$ *y* $\epsilon \in FIRST(\beta)$ *entonces*

$$FOLLOW(B) = FOLLOW(B) \cup FOLLOW(A)$$

14.3.2. 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 14.2 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

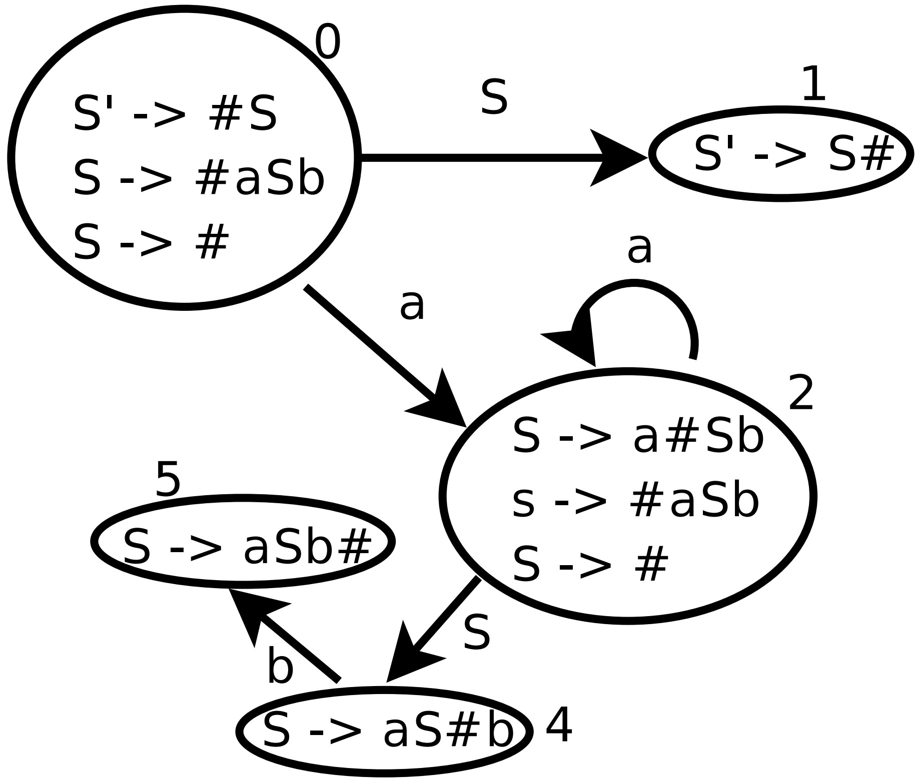

Figura 14.2: DFA equivalente al NFA de la figura 14.1

los que transita. Se repite el proceso con los conjuntos resultantes hasta que no se introducen nuevos conjuntos-estado.

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^*.

$$\hat{\delta}(q, x) = \begin{cases} \delta(\hat{\delta}(q, y), a) & \text{si } x = ya \\ q & \text{si } x = \epsilon \end{cases} \tag{14.1}$$

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 14.1.

Para el NFA mostrado en el ejemplo 14.2.1 el DFA construído mediante esta técnica es el que se muestra en la figura 14.2. Se ha utilizado el símbolo `#` como marcador. Se ha omitido el número 3 para que los estados coincidan en numeración con los generados por `yapp` (véase el cuadro 14.1).

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 14.2.4. De hecho es la tabla de transiciones restringida a V (recuerde que el alfabeto del autómata es $V \cup \Sigma$). Esto es,

$$\delta_{|V\times Q} : V \times Q \to Q.$$
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*):

$$\delta_{|\Sigma\times Q} : \Sigma \times Q \to Q$$
donde se define $action(i, a) = \delta(a, I_i)$

Cuando un estado s contiene un LR(0)-item de la forma $A \to \alpha_\uparrow$, 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 $A \to \alpha$ 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 $A \to \alpha$. 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 $A \to \alpha$ es el *handle* de γ es porque:

$$\exists S \underset{RM}{\overset{*}{\Longrightarrow}} \beta A b x \underset{RM}{\overset{*}{\Longrightarrow}} \beta \alpha b x = \gamma$$

Por tanto, cuando estamos reduciendo por $A \to \alpha$ los únicos terminales legales que cabe esperar en una reducción por $A \to \alpha$ son los terminales $b \in FOLLOW(A)$.

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 14.3.4. *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 14.2.4. 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".*

Definición 14.3.3. *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 14.3.1. *Al aplicar el algoritmo a la gramática 14.2.1*

1	$S \rightarrow a\ S\ b$
2	$S \rightarrow \epsilon$

partiendo del autómata finito determinista que se construyó en la figura 14.2 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.

El método de análisis *LALR* usado por `yapp` 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 `yapp` aplica la construcción del subconjunto a la gramática del ejemplo 14.2.1. Para ello construimos el siguiente programa `yapp`:

```
$ cat -n aSb.yp
     1  %%
     2  S:  # empty
     3      |   'a' S 'b'
     4  ;
     5  %%
     ......
```

y compilamos, haciendo uso de la opción `-v` para que `yapp` 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 14.1. 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 14.1: Tablas generadas por `yapp`. 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 14.27.

Ejercicio 14.3.1. *Compare la tabla 14.1 resultante de aplicar* `yapp` *con la que obtuvo en el ejemplo 14.3.1.*

14.4. El módulo Generado por yapp

La ejecución de la orden `yapp -m Calc Calc.yp` produce como salida el módulo `Calc.pm` el cual contiene las tablas LALR(1) para la gramática descrita en `Calc.yp`. Estas tablas son las que dirigen al analizador LR. Puede ver el código completo del módulo en el apéndice que se encuentra en la página 469. La estructura del módulo `Calc.pm` es como sigue:

```
package Calc;
use vars qw ( @ISA );
use strict;
@ISA= qw ( Parse::Yapp::Driver );
use Parse::Yapp::Driver;

```

```
 7 sub new {
 8    my($class)=shift;
 9    ref($class) and $class=ref($class);
10
11     my($self)=$class->SUPER::new(
12        yyversion => '1.05',
13        yystates => [
..          ...
32        ], # estados
33        yyrules  => [
..        # ... mas reglas
70        ], # final de las reglas
71        @_); # argumentos pasados
72   bless($self,$class);
73 }
```

La clase `Calc` hereda de `Parse::Yapp::Driver`, pero el objeto creado será bendecido en la clase `Calc` (Línea 4, véanse también la figura 14.3 y la línea 72 del fuente). Por tanto, el constructor llamado en la línea 11 es el de `Parse::Yapp::Driver`. Se utiliza la estrategia de llamada con parámetros con nombre. El valor para la clave `yystates` es una referencia anónima al array de estados y el valor para la clave `yyrules` es una referencia anónima a las reglas.

```
10
11     my($self)=$class->SUPER::new(
12        yyversion => '1.05',
13        yystates => [
14         {#State 0
15           DEFAULT => -1, GOTOS => { 'input' => 1 }
16         },
17         {#State 1
18           ACTIONS => {
19             'NUM' => 6, '' => 4, "-" => 2, "(" => 7,
20             'VAR' => 8, "\n" => 5, 'error' => 9
21           },
22           GOTOS => { 'exp' => 3, 'line' => 10 }
23         },
24         # ...  mas estados
25         {#State 27
26           ACTIONS => {
27             "-" => 12, "+" => 13, "/" => 15, "^" => 16,
28             "*" => 17
29           },
30           DEFAULT => -8
31         }
32        ], # estados
```

Se ve que un estado se pasa como un hash anónimo indexado en las acciones y los saltos.

Para consultar los números asociados con las reglas de producción vea el apéndice en la página 477 conteniendo el fichero `Calc.output`.

A continuación vienen las reglas:

```
33        yyrules  => [
34        [#Rule 0
35              '$start', 2, undef ],
36        [#Rule 1
```

```
             'input', 0, undef ],
       [#Rule 2
             'input', 2, sub
#line 17 "Calc.yp"
             { push(@{$_[1]},$_[2]); $_[1] }
       ],
       [#Rule 3
             'line', 1, sub
#line 20 "Calc.yp"
             { $_[1] }
       ],
       [#Rule 4
             'line', 2, sub
#line 21 "Calc.yp"
             { print "$_[1]\n" }
       ],
  # ... mas reglas
    [#Rule 11
            'exp', 3, sub
#line 30 "Calc.yp"
{ $_[1] * $_[3] }
   ],
  [#Rule 12
            'exp', 3, sub
#line 31 "Calc.yp"
  {
    $_[3] and return($_[1] / $_[3]);
    $_[0]->YYData->{ERRMSG} =   "Illegal division by zero.\n";
    $_[0]->YYError;
    undef
  }
    ],
  # ... mas reglas
  ], # final de las reglas
```

Las reglas son arrays anónimos conteniendo el nombre de la regla o variable sintáctica (`exp`), el número de símbolos en la parte derecha y la subrutina anónima con el código asociado.

Vemos como la acción es convertida en una subrutina anónima. Los argumentos de dicha subrutina son los atributos semánticos asociados con los símbolos en la parte derecha de la regla de producción. El valor retornado por la acción/subrutina es el valor asociado con la reducción.

Para hacer que el compilador Perl diagnostique los errores relativos al fuente `Calc.yp` se usa una directiva `#line`.

```
  @_);
  bless($self,$class);
}

```

la bendición con dos argumentos hace que el objeto pertenezca a la clase `Calc`. A continuación siguen las subrutinas de soporte:

```
#line 44 "Calc.yp"


sub _Error {
```

```
  # ...
}

sub _Lexer {
  my($parser)=shift;
  # ...
}

sub Run {
    my($self)=shift;
    $self->YYParse( yylex => \&_Lexer, yyerror => \&_Error );
}

my($calc)=new Calc;
$calc->Run;

1;
```

14.5. 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 *yapp* 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 14.5.1. *Análizador LR*

```
push(s0);
 b = yylex();
 for( ; ; ;) {
   s = top(0); a = b;
   switch (action[s][a]) {
     case "shift t" :
       push(t);
       b = yylex();
       break;
     case "reduce A ->alpha" :
       eval(Sub{A -> alpha}->(top(|alpha|-1).attr, ... , top(0).attr));
       pop(|alpha|);
       push(goto[top(0)][A]);
       break;
     case "accept" : return (1);
     default : yyerror("syntax error");
   }
 }
```

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 `sub_{reduce A -> alpha}` el código de la acción asociada con la regla $A \to \alpha$.

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 `yapp` la construcción de las tablas de *acciones* y *gotos* se realiza mediante el algoritmo *LALR*.

14.6. Depuración en `yapp`

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 un ejemplo de salida para la gramática que se describe en la página 475 cuando se llama con:
`$self->YYParse( yylex => \&_Lexer, yyerror => \&_Error, yydebug => 0x1F )`

```
$ ./use_aSb.pl
----------------------------------------
In state 0:
Stack:[0]
ab  # el usuario ha escrito esto
Need token. Got >a<
Shift and go to state 2.
----------------------------------------
In state 2:
Stack:[0,2]
Need token. Got >b<
Reduce using rule 1 (S,0): S -> epsilon
Back to state 2, then go to state 4.
----------------------------------------
In state 4:
Stack:[0,2,4]
Shift and go to state 5.
----------------------------------------
In state 5:
Stack:[0,2,4,5]
Don't need token.
Reduce using rule 2 (S,3): 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.
```

14.7. 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 *yapp* 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 *tokens* 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 14.1, 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 *yapp* 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 *yapp* 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 `yapp`:

```
$ 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
#!/usr/bin/perl -w
use strict;
use Precedencia;

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/^[ \t]//;

  for ($parser->YYData->{INPUT}) {
      s/^([0-9]+(?:\.[0-9]+)?)//
              and return('NUMBER',$1);
      s/^(.)//s
              and return($1,$1);
  }
}

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 14.6 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:

```
yapp -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 `yapp` 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:

```
%left '@'
%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 14.7.1. *Explique la forma en que* `yapp` *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.

14.8. Generación interactiva de analizadores `Yapp`

En el siguiente código, la subrutina `create_yapp_package` nos muestra como crear un analizador `Yapp` en tiempo de ejecución. Las dos líneas:

```
  my $p = new Parse::Yapp(input => $grammar);
  $p = $p->Output(classname => $name);
```

crean una cadena en `$p` conteniendo el código de la clase que implanta el analizador. Todo el truco está en hacer

```
  eval $p;
```

para tener el paquete a mano:

```
$ cat left.pl
#!/usr/local/bin/perl5.8.0 -w
#use strict;
use Parse::Yapp;

sub lex{
    my($parser)=shift;

    return('',undef) unless $parser->YYData->{INPUT};
```

```
    for ($parser->YYData->{INPUT}) {
        s/^\s*//;
        s/^(.)//;
        my $ret = $1;
        return($ret, $ret);
    }
}

sub yapp {
  my $grammar = shift
     or die "Must specify a grammar as first argument";
  my $name = shift
     or die "Must specify the name of the class as second argument";

  my $p = new Parse::Yapp(input => $grammar) or die "Bad grammar.";
  $p = $p->Output(classname => $name) or die "Can't generate parser.";

  eval $p;
  $@ and die "Error while compiling your parser: $@\n";
}

######## main #########
my $grammar = q {
%left '*'
%%
S:  A
;

A:  A '*' A  { "($_[1] $_[2] $_[3])" }
 |  B
;

B:  'a' | 'b' | 'c' | 'd'
;

%%
};

&yapp($grammar, "Example");
my $p = new Example(yylex => \&lex, yyerror => sub {});

print "Expresion: ";
$p->YYData->{INPUT} = <>;
$p->YYData->{INPUT} =~ s/\s*$//;

my $out=$p->YYParse;
print "out = $out\n";
```

Sigue un ejemplo de ejecución:

```
$ ./left.pl
Expresion: a*b*c*d
out = (((a * b) * c) * d)
```

14.9. Construcción del Árbol Sintáctico

El siguiente ejemplo usa `yapp` para construir el árbol sintáctico de una expresión en infijo:

```
$ cat -n Infixtree_bless.yp
#
# Infixtree.yp
#

%{
use Data::Dumper;
%}
%right  '='
%left   '-' '+'
%left   '*' '/'
%left   NEG

%%
input:  #empty
        |   input line
;

line:     '\n'        { $_[1] }
        | exp '\n'    { print Dumper($_[1]); }
        | error '\n'  { $_[0]->YYErrok }
;

exp:        NUM
        |   VAR                 { $_[1] }
        |   VAR '=' exp         { bless [$_[1], $_[3]], 'ASSIGN' }
        |   exp '+' exp         { bless [$_[1], $_[3] ], 'PLUS'}
        |   exp '-' exp         { bless [$_[1], $_[3] ], 'MINUS'}
        |   exp '*' exp         { bless [$_[1], $_[3]], 'TIMES' }
        |   exp '/' exp         { bless [$_[1], $_[3]], 'DIVIDE' }
        |   '-' exp %prec NEG   { bless [$_[2]], 'NEG' }
        |   '(' exp ')'         { $_[2] }
;

%%

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/^[ \t]//;

    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 );
}
```

Para compilar hacemos:

```
$ yapp -m Infixtree Infixtree_bless.yp
```

El guión que usa el analizador anterior es similar al que vimos en la sección 14.1:

```
$ cat -n ./useinfixtree.pl
     1  #!/usr/bin/perl -w
     2
     3  use Infixtree;
     4
     5  $parser = new Infixtree();
     6  $parser->Run;
```

Veamos un ejemplo de ejecución:

```
$ ./useinfixtree.pl
a = 2+3
$VAR1 = bless( [
                 'a',
                 bless( [
                          '2',
                          '3'
                        ], 'PLUS' )
               ], 'ASSIGN' );
b = a*4+a
$VAR1 = bless( [
                 'b',
                 bless( [
                          bless( [
                                   'a',
                                   '4'
                                 ], 'TIMES' ),
                          'a'
                        ], 'PLUS' )
               ], 'ASSIGN' );
```

14.10. 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 $A \rightarrow \alpha\beta$:

$$A \rightarrow \alpha\,\{action_1\}\,\beta\,\{action_2\}$$

`yapp` 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.

14.11. Esquemas de Traducción

Un *esquema de traducción* es una gramática independiente del contexto en la cuál se han asociado atributos a los símbolos de la gramática. Un atributo queda caracterizado por un identificador o nombre y un tipo o clase. Además se han insertado acciones, esto es, código Perl/Python/C, ... en medio de las partes derechas. En ese código es posible referenciar los atributos de los símbolos de la gramática como variables del lenguaje subyacente.

Recuerde que el orden en que se evalúan los fragmentos de código es el de un recorrido primero-profundo del árbol de análisis sintáctico. Mas específicamente, considerando a las acciones como hijos-hoja del nodo, el recorrido que realiza un esquema de traducción es:

```
1    sub esquema_de_traduccion {
2      my $node = shift;
3
4      for my $child ($node->children) { # de izquierda a derecha
5        if ($child->isa('ACTION') {
6          $child->execute;
7        }
8        else { esquema_de_traduccion($child) }
9      }
10   }
```

Obsérvese que, como el bucle de la línea 4 recorre a los hijos de izquierda a derecha, se debe dar la siguiente condición para que un esquema de traducción funcione:

Para cualquier regla de producción aumentada con acciones, de la forma

$$A \rightarrow X_1 \ldots X_j\texttt{\{ action(\$A\{b\}, \$X}_1\texttt{\{c\}}\ldots\ \texttt{X}_n\texttt{\{d\})\}}X_{j+1} \ldots X_n$$

debe ocurrir que los atributos evaluados en la acción insertada después de X_j dependan de atributos y variables que fueron computadas durante la visita de los hermanos izquierdos o de sus ancestros. En particular no deberían depender de atributos asociados con las variables $X_{j+1} \ldots X_n$. Ello no significa que no sea correcto evaluar atributos de $X_{j+1} \ldots X_n$ en esa acción.

14.12. Definición Dirigida por la Sintáxis

Una *definición dirigida por la sintáxis* es un pariente cercano de los esquemas de traducción. En una definición dirigida por la sintáxis una gramática $G = (V, \Sigma, P, S)$ se aumenta con nuevas características:

- A cada símbolo $S \in V \cup \Sigma$ de la gramática se le asocian cero o mas atributos. Un atributo queda caracterizado por un identificador o nombre y un tipo o clase. A este nivel son *atributos formales*, como los parámetros formales, en el sentido de que su realización se produce cuando el nodo del árbol es creado.
- A cada regla de producción $A \to X_1 X_2 \ldots X_n \in P$ se le asocian un conjunto de *reglas de evaluación de los atributos* o *reglas semánticas* que indican que el atributo en la parte izquierda de la regla semántica depende de los atributos que aparecen en la parte derecha de la regla. El atributo que aparece en la parte izquierda de la regla semántica puede estar asociado con un símbolo en la parte derecha de la regla de producción.
- Los atributos de cada símbolo de la gramática $X \in V \cup \Sigma$ se dividen en dos grupos disjuntos: *atributos sintetizados* y *atributos heredados*. Un atributo de X es un *atributo heredado* si depende de atributos de su padre y hermános en el árbol. Un *atributo sintetizado* es aquél tal que el valor del atributo depende de los valores de los atributos de los hijos, es decir en tal caso X ha de ser una variable sintáctica y los atributos en la parte derecha de la regla semántica deben ser atributos de símbolos en la parte derecha de la regla de producción asociada.
- Los atributos predefinidos se denominán *atributos intrínsecos*. Ejemplos de atributos intrínsecos son los atributos sintetizados de los terminales, los cuáles se han computado durante la fase de análisis léxico. También son atributos intrínsecos los atributos heredados del símbolo de arranque, los cuales son pasados como parámetros al comienzo de la computación.

La diferencia principal con un esquema de traducción está en que no se especifica el orden de ejecución de las reglas semánticas. Se asume que, bien de forma manual o automática, se resolverán las dependencias existentes entre los atributos determinadas por la aplicación de las reglas semánticas, de manera que serán evaluados primero aquellos atributos que no dependen de ningún otro, despues los que dependen de estos, etc. siguiendo un esquema de ejecución que viene guiado por las dependencias existentes entre los datos.

Aunque hay muchas formas de realizar un evaluador de una definición dirigida por la sintáxis, conceptualmente, tal evaluador debe:

1. Construir el árbol de análisis sintáctico para la gramática y la entrada dadas.
2. Analizar las reglas semánticas para determinar los atributos, su clase y las dependencias entre los mismos.
3. Construir el *grafo de dependencias* de los atributos, el cual tiene un nodo para cada ocurrencia de un atributo en el árbol de análisis sintáctico etiquetado con dicho atributo. El grafo tiene una arista entre dos nodos si existe una dependencia entre los dos atributos a través de alguna regla semántica.
4. Supuesto que el grafo de dependencias determina un *orden parcial* (esto es cumple las propiedades reflexiva, antisimétrica y transitiva) construir un *orden topológico* compatible con el orden parcial.
5. Evaluar las reglas semánticas de acuerdo con el orden topológico.

Una definición dirigida por la sintáxis en la que las reglas semánticas no tienen efectos laterales se denomina una *gramática atribuída*.

Si la definición dirigida por la sintáxis puede ser realizada mediante un esquema de traducción se dice que es *L-atribuída*. Para que una definición dirigida por la sintáxis sea L-atribuída deben cumplirse que cualquiera que sea la regla de producción $A \rightarrow X_1 \ldots X_n$, los atributos heredados de X_j pueden depender únicamente de:

1. Los atributos de los símbolos a la izquierda de X_j

2. Los atributos heredados de A

Nótese que las restricciones se refieren a los atributos heredados. El cálculo de los atributos sintetizados no supone problema para un esquema de traducción. Si la gramática es LL(1), resulta fácil realizar una definición L-atribuída en un analizador descendente recursivo predictivo.

Si la definición dirigida por la sintáxis sólo utiliza atributos sintetizados se denomina *S-atribuída*. Una definición S-atribuída puede ser fácilmente trasladada a un programa `yapp`.

14.13. Manejo en `yapp` de Atributos Heredados

Supongamos que `yapp` 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 `yapp` 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:

D →	C T `{ $L{c} = $C{c}; $L{t} = $T{t} }` L
C →	global `{ $C{c} = "global" }`
C →	local `{ $C{c} = "local" }`
T →	integer `{ $T{t} = "integer" }`
T →	float `{ $T{t} = "float" }`
L →	`{ $L`$_1$`{t} = $L{t}; $L`$_1$`{c} = $L{c}; }` L_1 ','
	id `{ set_class($id{v}, $L{c}); set_type($id{v}, $L{t}); }`
L →	id `{ set_class($id{v}, $L{c}); set_type($id{v}, $L{t}); }`

Los atributos `c` y `t` denotan respectivamente la clase y el tipo.

Ejercicio 14.13.1. *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 `yapp`. *¿En que orden construye* `yapp` *el árbol y en que orden ejecutará las acciones?*

A la hora de transformar este esquema de traducción en un programa `yapp` es importante darse cuenta que en cualquier derivación a derechas desde D, cuando se reduce por una de las reglas

$$L \rightarrow \text{id} \mid L_1 \text{ ',' id}$$

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:

$$\begin{array}{c} D \Longrightarrow C\ T\ L \Longrightarrow C\ T\ L, \text{id} \Longrightarrow C\ T\ L, \text{id}, \text{id} \Longrightarrow C\ T\ \text{id}, \text{id}, \text{id} \Longrightarrow \\ \Longrightarrow C\ \text{float id}, \text{id}, \text{id} \Longrightarrow \text{local float id}, \text{id}, \text{id} \end{array}$$

Observe que el orden de recorrido de `yapp` es:

$$\begin{array}{c} \text{local float id}, \text{id}, \text{id} \Longleftarrow C\ \text{float id}, \text{id} \Longleftarrow C\ T\ \text{id}, \text{id}, \text{id} \Longleftarrow \\ \Longleftarrow C\ T\ L, \text{id}, \text{id} \Longleftarrow C\ T\ L, \text{id} \Longleftarrow C\ T\ L \Longleftarrow D \end{array}$$

en la antiderivación, cuando el mango es una de las dos reglas para listas de identificadores, $L \rightarrow \text{id}$ y $L \rightarrow L, \text{id}$ es decir durante las tres ultimas antiderivaciones:

$$C\ T\ L, \text{id}, \text{id} \Longleftarrow C\ T\ L, \text{id} \Longleftarrow C\ T\ L \Longleftarrow D$$

las variables a la izquierda del mango son T y C. Esto ocurre siempre. Estas observaciones nos conducen al siguiente programa `yapp`:

```
$ 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 `yapp`:

```
$ 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, $1);
      }
      elsif ($input =˜ m/\G\s*(.)/gc) {
        die "Caracter invalido: $1\n";
      }
      else {
        return (’’, undef); # end of file
      }
      redo;
    }
  }
}
```

```
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 `$input` (véase la sección 5.16 para mas detalles sobre el uso de clausuras en Perl). Aunque la variable `$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
```

Ejercicio 14.13.2. *El siguiente programa* `yapp` *calcula un árbol de análisis abstracto para la gramática del ejemplo anterior:*

```
%token FLOAT INTEGER
%token GLOBAL
%token LOCAL
%token NAME

%%
declarationlist
  : /* vacio */                     { bless [], 'declarationlist' }
  | declaration ';' declarationlist { push @{$_[3]}, $_[1]; $_[3] }
  ;

declaration
  : class type namelist
      {
        bless {class => $_[1], type => $_[2], namelist => $_[3]}, 'declaration';
      }
  ;

class
  : GLOBAL  { bless { GLOBAL => 0}, 'class' }
  | LOCAL   { bless { LOCAL => 1}, 'class' }
  ;

type
  : FLOAT   { bless { FLOAT => 2}, 'type' }
  | INTEGER { bless { INTEGER => 3}, 'type' }
  ;

namelist
```

```
  : NAME
     { bless [ $_[1]], 'namelist' }
  | namelist ',' NAME
      { push @{$_[1]}, $_[3]; $_[1] }
  ;
%%
```

sigue un ejemplo de ejecución:

```
$ ./useinherited3.pl
Entrada (En Unix, presione CTRL-D para terminar):
global float x,y;
$VAR1 = bless( [
  bless( {
    'namelist' => bless( [ 'x', 'y' ], 'namelist' ),
    'type' => bless( { 'FLOAT' => 2 }, 'type' ),
    'class' => bless( { 'GLOBAL' => 0 }, 'class' )
  }, 'declaration' )
], 'declarationlist' );
```

Extienda el programa del ejemplo para que la gramática incluya las acciones del esquema de traducción. Las acciones se tratarán como un terminal `CODE` *y serán devueltas por el analizador léxico. Su atributo asociado es el texto del código. El programa* `yapp` *deberá devolver el árbol abstracto extendido con las acciones-terminales. La parte mas difícil de este problema consiste en "reconocer" el código Perl incrustado. La estrategia seguir consiste en contar el número de llaves que se abren y se cierran. Cuando el contador alcanza cero es que hemos llegado al final del código Perl incrustado. Esta estrategia tiene una serie de problemas. ¿Sabría decir cuáles? (sugerencia: repase la sección 14.19.3 o vea como* `yapp` *resuelve el problema).*

14.14. Acciones en Medio de una Regla y Atributos Heredados

La estrategia utilizada en la sección 14.13 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 14.14.1. *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 `yapp`:

```
$ cat -n Inherited2.yp
%%
S : 'a' A C
  | 'b' A B  { $_[2]; } C
  ;

C : 'c' { print "Valor: ",$_[0]->YYSemval(0),"\n"; $_[0]->YYSemval(0) }
  ;

A : 'a' { 'a' }
  ;

B : 'b' { 'b' }
  ;

%%
```

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 `yapp`. La regla 2 por la que se reduce es la asociada con la acción intermedia:

```
$ cat -n Inherited2.output
Rules:
------
0:      $start -> S $end
1:      S -> 'a' A C
2:      @1-3 -> /* empty */
3:      S -> 'b' A B @1-3 C
4:      C -> 'c'
5:      A -> 'a'
6:      B -> 'b'
...
```

Obsérvese la notación usada por `yapp` para la *acción en medio de la regla*yapp!variables intermedias: `@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.

14.15. Recuperación de Errores

Las entradas de un traductor pueden contener errores. El lenguaje `yapp` 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.

Consideremos lo que ocurre al ejecutar nuestra calculadora `yapp` con una entrada errónea. Recordemos la gramática:

```
%right  '='
%left   '-' '+'
%left   '*' '/'
%left   NEG
%right  '^'
%%
input:  # empty
        |   input line  { push(@{$_[1]},$_[2]); $_[1] }
;
line:       '\n'       { $_[1] }
        |   exp '\n'   { print "$_[1]\n" }
        |   error '\n' { $_[0]->YYErrok }
;
```

La regla `line` $\rightarrow$ `error '\n'` es una producción de error. La idea general de uso es que, a traves de la misma, el programador le indica a `yapp` 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 al método `YYErrok` el programador anuncia que, si se alcanza este punto, la recuperación puede considerarse "completa" y que `yapp` 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.

El resto de la gramática de la calculadora era como sigue:

```
exp:        NUM
        |   VAR                 { $_[0]->YYData->{VARS}{$_[1]} }
        |   VAR '=' exp         { $_[0]->YYData->{VARS}{$_[1]}=$_[3] }
        |   exp '+' exp         { $_[1] + $_[3] }
        |   exp '-' exp         { $_[1] - $_[3] }
        |   exp '*' exp         { $_[1] * $_[3] }
        |   exp '/' exp         {
                                  $_[3]
```

```
                                 and return($_[1] / $_[3]);
                                 $_[0]->YYData->{ERRMSG}
                                   =   "Illegal division by zero.\n";
                                 $_[0]->YYError;
                                 undef
                               }
        |   '-' exp %prec NEG   { -$_[2] }
        |   exp '^' exp         { $_[1] ** $_[3] }
        |   '(' exp ')'         { $_[2] }
;
```

en la ejecución activamos el flag `yydebug` a `0x10` para obtener información sobre el tratamiento de errores:

```
$self->YYParse( yylex => \&_Lexer, yyerror => \&_Error, yydebug => 0x10 );
```

Pasemos a darle una primera entrada errónea:

```
$ ./usecalc.pl
3-+2
Syntax error.
**Entering Error recovery.
**Pop state 12.
**Pop state 3.
**Shift $error token and go to state 9.
**Dicard invalid token >+<.
**Pop state 9.
**Shift $error token and go to state 9.
**Dicard invalid token >NUM<.
**Pop state 9.
**Shift $error token and go to state 9.
**End of Error recovery.
```

El esquema general del algoritmo de recuperación de errores usado por la versión actual de `yapp` 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`. En el ejemplo anterior el analizador estaba en el estado 12 y lo retira de la pila. Los contenidos del estado 12 son:

   ```
   exp -> exp '-' . exp (Rule 10)
   '('shift 7 '-'shift 2 NUM shift 6 VAR shift 8
   exp go to state 21
   ```

   Obviamente no esperabamos ver un `'+'` aqui. El siguiente estado en la cima de la pila es el 3, el cual tampoco tiene ninguna transición ante el terminal `error`:

   ```
   line -> exp . '\n'(Rule 4)
   exp -> exp . '+' exp (Rule 9)
   exp -> exp . '-' exp (Rule 10)
   exp -> exp . '*' exp (Rule 11)
   exp -> exp . '/' exp (Rule 12)
   exp -> exp . '^' exp (Rule 14)

   '*'shift 17 '+'shift 13 '-'shift 12 '/'shift 15
   '\n'shift 14 '^'shift 16
   ```

El pop sobre el estado 3 deja expuesto en la superficie al estado 1, el cuál "sabe" como manejar el error:

```
$start -> input . $end (Rule 0)
input -> input . line (Rule 2)

$end shift 4 '('shift 7 '-'shift 2 '\n'shift 5
NUM shift 6 VAR shift 8
error shift 9

exp go to state 3
line go to state 10
```

3. En este punto transita al estado correspondiente a desplazar el *token* `error`.

   En consecuencia, con lo dicho, en el ejemplo se va al estado 9:

```
line -> error . '\n'(Rule 5)
'\n'shift, and go to state 20
```

4. Entonces el algoritmo de recuperación va leyendo *tokens* y descartandolos hasta encontrar uno que sea aceptable. En este caso hemos especificado que el terminal que nos da cierta confianza de recuperación es el retorno de carro:

```
**Dicard invalid token >+<.
**Pop state 9.
**Shift $error token and go to state 9.
**Dicard invalid token >NUM<.
**Pop state 9.
**Shift $error token and go to state 9.
**End of Error recovery.
```

5. Sólo se envían nuevos mensajes de error una vez asimilados (desplazados) algunos símbolos terminales. De este modos se intenta evitar la aparición masiva de mensajes de error.

14.16. 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.

En el caso particular de las listas, se recomienda seguir el siguiente esquema:

Ejercicio 14.16.1. *Compruebe el funcionamiento de la metodología para la recuperación de errores en listas presentada en la tabla 14.2 estudie el siguiente programa* `yapp` *siguiendo la traza de estados, generando entradas con todos los tipos de error posibles. ¿Cómo se recupera el analizador en caso de existencia de un segundo error? ¿Que ocurre si dos errores consecutivos están muy próximos? El programa corresponde al tercer caso de la tabla 14.2, el caso x:y{Ty} con x =* `list`*, T =* `','` *e y =* `NUMBER`*:*

Construcción	EBNF	yapp

secuencia opcional	x:{y}	`x : /* null */`
		`  | x y  { $_[0]->YYErrok; }`
		`  | x error`

secuencia	x:y{y}	`x : y`
		`  | xy   { $_[0]->YYErrok; }`
		`  | error`
		`  | x error`

lista	x:y{Ty}	`x : y`
		`  | x T y { $_[0]->YYErrok; }`
		`  | error`
		`  | x error`
		`  | x error y { $_[0]->YYErrok; }`
		`  | x  T error`

Cuadro 14.2: Recuperación de errores en listas

```
%token NUMBER
%%
command
  :
  | command list '\n' { $_[0]->YYErrok; }
  ;

list
    : NUMBER            { put($1); }
    | list ',' NUMBER   { put($3); $_[0]->YYErrok; }
    | error             { err(1); }
    | list error        { err(2); }
    | list error NUMBER { err(3); put($3); $_[0]->YYErrok; }
    | list ',' error    { err(4); }
    ;

%%
sub put { my $x = shift; printf("%2.1lf\n",$x); }
sub err { my $code = shift; printf("err %d\n",$code); }
 ...
```

14.17. Consejos a seguir al escribir un programa yapp

Cuando escriba un programa `yapp` asegurese de seguir los siguientes consejos:

1. Coloque el punto y coma de separación de reglas en una línea aparte. Un punto y coma "pegado" al final de una regla puede confundirse con un terminal de la regla.
2. Si hay una regla que produce vacío, coloquela en primer lugar y acompáñela de un comentario resaltando ese hecho.
3. Nunca escriba dos reglas de producción en la misma línea.
4. Sangre convenientemente todas las partes derechas de las reglas de producción de una variable, de modo que queden alineadas.

5. Ponga nombres representativos a sus variables sintácticas. No llame `Z` a una variable que representa el concepto "lista de parámetros", llámela `ListaDeParametros`.

6. Es conveniente que declare los terminales simbólicos, esto es, aquellos que llevan un identificador asociado. Si no llevan prioridad asociada o no es necesaria, use una declaración `%token`. De esta manera el lector de su programa se dará cuenta rápidamente que dichos identificadores no se corresponden con variables sintácticas. Por la misma razón, si se trata de terminales asociados con caracteres o cadenas no es tan necesario que los declare, a menos que, como en el ejemplo de la calculadora para `'+'` y `'*'`, sea necesario asociarles una precedencia.

7. Es importante que use la opción `-v` para producir el fichero `.output` conteniendo información detallada sobre los conflictos y el autómata. Cuando haya un conflicto shift-reduce no resuelto busque en el fichero el estado implicado y vea que LR(0) items $A \to \alpha_\uparrow$ y $B \to \beta_\uparrow\gamma$ entran en conflicto.

8. Si según el informe de `yapp` el conflicto se produce ante un terminal a, es porque $a \in FOLLOW(A)$ y $a \in FIRST(\gamma)$. Busque las causas por las que esto ocurre y modifique su gramática con vistas a eliminar la presencia del terminal a en uno de los dos conjuntos implicados o bien establezca reglas de prioridad entre los terminales implicados que resuelvan el conflicto.

9. Nótese que cuando existe un conflicto de desplazamiento reducción entre $A \to \alpha_\uparrow$ y $B \to \beta_\uparrow\gamma$, el programa `yapp` contabiliza un error por cada terminal $a \in FOLLOW(A) \cap FIRST(\gamma)$. Por esta razón, si hay 16 elementos en $FOLLOW(A) \cap FIRST(\gamma)$, el analizador `yapp` informará de la existencia de 16 conflictos *shift-reduce*, cuando en realidad se trata de uno sólo. No desespere, los conflictos "auténticos" suelen ser menos de los que `yapp` anuncia.

10. Si necesita declarar variables globales, inicializaciones, etc. que afectan la conducta global del analizador, escriba el código correspondiente en la cabecera del analizador, protegido por los delimitadores `%{` y `%}`. Estos delimitadores deberán aparecer en una línea aparte. Por ejemplo:

```
%{
our contador = 0;
%}

%token NUM
...
%%
```

11. Si tiene problemas en tiempo de ejecución con el comportamiento del análizador sintáctico use la opción `yydebug => 0x1F` en la llamada al analizador.

12. Si trabaja en windows y pasa los ficheros a unix tenga cuidado con la posible introducción de caractéres espúreos en el fichero. Debido a la presencia de caractéres de control invisibles, el analizador `yapp` pasará a rechazar una gramatica aparentemente correcta.

13. Sea consciente de que los analizadores sintáctico y léxico mantienen una relación de corutinas en `yapp`: Cada vez que el analizador sintáctico necesita un nuevo terminal para decidir que regla de producción se aplica, llama al analizador léxico, el cuál deberá retornar el siguiente terminal. La estrategia es diferente de la utilizada en el ejemplo usado para el lenguaje Tutu en el capítulo 11. Allí generabamos en una primera fase la lista de terminales. Aquí los terminales se generan de uno en uno y cada vez que se encuentra uno nuevo se retorna al analizador sintáctico. La ventaja que tiene este método es que permite colaborar al analizador sintáctico y al analizador léxico para "dinámicamente" modificar la conducta del análisis léxico. Por ejemplo en los compiladores del lenguaje C es común hacer que el analizador léxico cuando descubre un identificador que previamente ha sido declarado como identificador de tipo (mediante el uso de `typedef`) retorne un terminal `TYPENAME` diferente del terminal `ID` que caracteriza a los identificadores. Para ello,

el analizador sintáctico, cuando detecta una tal declaración, “avisa” al analizador léxico para que modifique su conducta. El analizador sintáctico volverá a avisarlo cuando la declaración del identificador como identificador de tipo salga de ámbito y pierda su especial condición.

14. En `yapp` el analizador sintáctico espera que el analizador léxico devuelva de cada vez una pareja formada por dos escalares. El primer escalar es la cadena que designa el terminal. A diferencia de la habitual costumbre `yacc` de codificar los terminales como enteros, en `yapp` se suelen codificar como cadenas. La segunda componente de la pareja es el atributo asociado con el terminal. Si el atributo es un atributo complejo que necesitas representar mediante un hash o un vector, lo mejor es hacer que esta componente sea una referencia al objeto describiendo el atributo. El analizador léxico le indica al sintáctico la finalización de la entrada enviándole la pareja `('',undef)` formada por la palabra vacía con atributo `undef`.

15. Hay fundamentalmente dos formas de hacer el analizador léxico: hacerlo destructivo o no destructivo. En los destructivos se usa el operador de sustitución `s` (véase el ejemplo de la sección 14.1), en cuyo caso la entrada procesada es retirada de la cadena leída. En los no destructivos utilizamos el operador de emparejamiento `m`. Véase el ejemplo de analizador léxico en la sección 14.13 (concretamente la subrutina `scanner` en la línea 20 del fichero `useinherited.pl`)

Ejemplo 14.17.1. *Consideremos de nuevo el programa* `yapp` *para producir árboles para las expresiones en infijo. Supongamos que olvidamos introducir una prioridad explícita al terminal* `'='`*:*

```
$ cat -n Infixtree_conflict.yp
#
# Infixtree.yp
#

%{
use Data::Dumper;
%}
%left   '-' '+'
%left   '*' '/'
%left   NEG

%%
input:  #empty
        |   input line
;

line:     '\n'        { $_[1] }
        | exp '\n'     { print Dumper($_[1]); }
        | error '\n'   { $_[0]->YYErrok }
;

exp:        NUM
        |   VAR                 { $_[1] }
        |   VAR '=' exp         { bless [$_[1], $_[3]], 'ASSIGN' }
        |   exp '+' exp         { bless [$_[1], $_[3] ], 'PLUS'}
        |   exp '-' exp         { bless [$_[1], $_[3] ], 'MINUS'}
        |   exp '*' exp         { bless [$_[1], $_[3]], 'TIMES' }
        |   exp '/' exp         { bless [$_[1], $_[3]], 'DIVIDE' }
   ....
```

en este caso al compilar encontraremos conflictos:

```
$ yapp -v -m Infixtree Infixtree_conflict.yp
4 shift/reduce conflicts
```

En tal caso lo que debemos hacer es editar el fichero `.output`*. El comienzo del fichero es como sigue:*

```
$ cat -n Infixtree_conflict.output
1  Warnings:
2  ---------
3  4 shift/reduce conflicts
4
5  Conflicts:
6  ----------
7  Conflict in state 11 between rule 13 and token '-' resolved as reduce.
8  Conflict in state 11 between rule 13 and token '*' resolved as reduce.
  ...
```

Tal y como indica la expresión ...`resolved as` *..., las líneas como la 7, la 8 y siguientes se refieren a conflictos resueltos. Mas abajo encontraremos información sobre la causa de nuestros conflictos no resueltos:*

```
    ...
26  Conflict in state 23 between rule 11 and token '/' resolved as reduce.
27  State 25 contains 4 shift/reduce conflicts
```

Lo que nos informa que los conflictos ocurren en el estado 25 ante 4 terminales distintos. Nos vamos a la parte del fichero en la que aparece la información relativa al estado 25. Para ello, como el fichero es grande, buscamos por la cadena adecuada. En `vi` *buscaríamos por* `/^State 25`*. Las líneas correspondientes contienen:*

```
291 State 25:
292
293   exp -> VAR '=' exp .  (Rule 8)
294   exp -> exp . '+' exp  (Rule 9)
295   exp -> exp . '-' exp  (Rule 10)
296   exp -> exp . '*' exp  (Rule 11)
297   exp -> exp . '/' exp  (Rule 12)
298
299   '*' shift, and go to state 16
300   '+' shift, and go to state 13
301   '-' shift, and go to state 12
302   '/' shift, and go to state 15
303
304   '*' [reduce using rule 8 (exp)]
305   '+' [reduce using rule 8 (exp)]
306   '-' [reduce using rule 8 (exp)]
307   '/' [reduce using rule 8 (exp)]
308   $default  reduce using rule 8 (exp)
```

El comentario en la línea 308 (`$default` *...) indica que por defecto, ante cualquier otro terminal que no sea uno de los explícitamente listados, la acción a tomar por el analizador será reducir por la regla 8.*

Una revisión a la numeración de la gramática, al comienzo del fichero `.output` *nos permite ver cuál es la regla 8:*

```
29 Rules:
30 ------
```

```
0:  $start -> input $end
1:  input -> /* empty */
2:  input -> input line
3:  line -> '\n'
4:  line -> exp '\n'
5:  line -> error '\n'
6:  exp -> NUM
7:  exp -> VAR
8:  exp -> VAR '=' exp
9:  exp -> exp '+' exp
10: exp -> exp '-' exp
11: exp -> exp '*' exp
12: exp -> exp '/' exp
13: exp -> '-' exp
14: exp -> '(' exp ')'
```

Efectivamente, es la regla de asignación `exp -> VAR '=' exp`*. El conflicto aparece por que los terminales* `* + - /` *están en el conjunto FOLLOW(*`exp`*) y también cabe esperarlos respectivamente en las reglas 9, 10, 11 y 12 ya que el estado 25 contiene:*

```
exp -> exp . '+' exp  (Rule 9)
exp -> exp . '-' exp  (Rule 10)
exp -> exp . '*' exp  (Rule 11)
exp -> exp . '/' exp  (Rule 12)
```

Estamos ante un caso en el que se aplica el consejo número 8. Los items de la forma $B \to \beta_{\uparrow}\gamma$, son los de la forma `exp -> exp . '+' exp`*, etc. El item de la forma $A \to \alpha_{\uparrow}$ es en este caso* `exp -> VAR '=' exp`*.*

En efecto, en una expresión como `a = 4 + 3` *se produce una ambiguedad. ¿Debe interpretarse como* `(a = 4) + 3`*? ¿O bien como* `a = (4 + 3)`*?. La primera interpretación corresponde a reducir por la regla 8. La segunda a desplazar al estado 13. En este ejemplo, el conflicto se resuelve haciendo que tenga prioridad el desplazamiento, dando menor prioridad al terminal* `=` *que a los terminales* `* + - /`*.*

Ejercicio 14.17.1. *¿Que ocurre en el ejemplo anterior si dejamos que* `yapp` *aplique las reglas por defecto?*

14.18. Práctica: Un C simplificado

Escriba un analizador sintáctico usando `Parse::Yapp` para el siguiente lenguaje. La descripción utiliza una notación tipo BNF: las llaves indican 0 o mas repeticiones y los corchetes opcionalidad.

```
program             →  definitions { definitions }
definitions         →  datadefinition | functiondefinition
datadefinition      →  basictype declarator { ’,’ declarator } ’;’
declarator          →  ID { ’[’ constantexp ’]’ }
functiondefinition  →  [ basictype ] functionheader functionbody
basictype           →  INT | CHAR
functionheader      →  ID ’(’ [ parameters ] ’)’
parameters          →  basictype declarator { ’,’ basictype declarator }
functionbody        →  ’{’ { datadefinition } { statement } ’}’
statement           →  [ exp ] ’;’
                       | ’{’ { datadefinition } { statement } ’}’
                       | IF ’(’ exp ’)’ statement [ ELSE statement ]
                       | WHILE ’(’ exp ’)’ statement
                       | RETURN [ exp ] ’;’
constantexp         →  exp
exp                 →  lvalue ’=’ exp | lvalue ’+=’ exp
                       | exp ’&&’ exp | exp ’||’ exp |
                       | exp ’==’ exp | exp ’!=’ exp |
                       | exp ’<’ exp | exp ’>’ exp | exp ’<=’ exp | exp ’>=’ exp |
                       | exp ’+’ exp | exp ’-’ exp |
                       | exp ’*’ exp | exp ’/’ exp |
                       | unary
unary               →  ’++’ lvalue | ’−−’ lvalue | primary
primary             →  ’(’ exp ’)’ | ID ’(’ [ argumentlist ] ’)’ | lvalue | NUM | CHARACTER
lvalue              →  ID { ’[’ exp ’]’ }
argumentlist        →  exp { ’,’ exp }
```

Su analizador, además de seguir los consejos explícitados en la sección 14.17, deberá cumplir las siguientes especificaciones:

1. **Método de Trabajo**

   Parta de la definición BNF y proceda a introducir las reglas poco a poco:

```
%token declarator basictype functionheader functionbody
%%
program: definitionslist
       ;

definitionslist: definitions definitionslist
               | definitions
               ;

definitions: datadefinition
           | functiondefinition
           ;
datadefinition: basictype declaratorlist ';'
              ;

declaratorlist: declarator ',' declaratorlist
              | declarator
              ;
functiondefinition: basictype functionheader functionbody
                  | functionheader functionbody
                  ;
```

```

%%
```

a) Comienze trabajando en el cuerpo de la gramática.

b) Olvídese del analizador léxico por ahora. Su objetivo es tener una gramática limpia de conflictos y que reconozca el lenguaje dado.

c) Sustituya las repeticiones BNF por listas. Si una variable describe una lista de `cosas` llámela `cosaslist`.

d) Si tiene un elemento opcional en la BNF, por ejemplo, en la regla:
functiondefinition → [basictype] functionheader functionbody
sustitúyala por dos reglas una en la que aparece el elemento y otra en la que no.

```
functiondefinition: basictype functionheader functionbody
                  | functionheader functionbody
                  ;
```

e) Cada par de reglas que introduzca vuelva a recompilar con `yapp` la gramática para ver si se han producido conflictos. Cuando estoy editando la gramática suelo escribir a menudo la orden
`:!yapp %`
para recompilar:

```

     declaratorlist: declarator declaratorlist
                   | declarator
                   ;
     functiondefinition: basictype functionheader functionbody
                       | functionheader functionbody
                       ;

     %%
~
~
~
~
~
~
~
~
~
~
:!yapp %
```

Esto llama a `yapp` con el fichero bajo edición. Si hay errores los detectaré enseguida.

f) Insisto, procure detectar la aparición de un conflicto lo antes posible. Es terrible tener que limpiar una gramática llena de conflictos.

g) Ponga nombres significativos a las variables y terminales. Por favor, no los llame `d1`, `d2`, etc.

h) Cuando esté en el proceso de construcción de la gramática y aún le queden por rellenar variables sintácticas, declárelas como terminales mediante `%token` como en el código que aparece encima. De esta manera evitará las quejas de `yapp`.

2. **Resolución de Conflictos**

   Las operaciones de asignación tienen la prioridad mas baja, seguidas de las lógicas, los test de igualdad y después de los de comparación, a continuación las aditivas, multiplicativas y por último los `unary` y `primary`. Exprese la asociatividad natural y la prioridad especificada usando los mecanismos que `yapp` provee al efecto.

   La gramática 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:

   $$\text{if } E_1 \text{ then (if } E_2 \text{ then } S_1) \text{ else } S_2$$

   Utilice los mecanismos de priorización proporcionados por `yapp` para resolver el conflicto shift-reduce generado. ¿Es correcta en este caso particular la conducta a la que da lugar la acción `yapp` por defecto?

3. **Analizador Léxico**

   Además del tipo de terminal y su valor el analizador léxico deberá devolver el número de línea. El analizador léxico deberá aceptar comentarios C. En la gramática, el terminal `CHARACTER` se refiere a caracteres entre comillas simples (por ejemplo `'a'`).

   Se aconseja que las palabras reservadas del lenguaje no se traten con expresiones regulares específicas sino que se capturen en el patrón de identificador `[a-z_]\w+`. Se mantiene para ello un hash con las palabras reservadas que es inicializado al comienzo del programa. Cuando el analizador léxico encuentra un identificador mira en primer lugar en dicho hash para ver si es una palabra reservada y, si lo es, devuelve el terminal correspondiente. En caso contrario se trata de un identificador.

4. **Recuperación de Errores**

   Extienda la práctica con reglas para la recuperación de errores. Para las listas, siga los consejos dados en la sección 14.2. En aquellos casos en los que la introducción de las reglas de recuperación produzca ambiguedad, resuelva los conflictos.

5. **Árbol de Análisis Abstracto**

   La semántica del lenguaje es similar a la del lenguaje C (por ejemplo, las expresiones lógicas se tratan como expresiones enteras). El analizador deberá producir un árbol sintáctico abstracto. Como se hizo para el lenguaje Tutu introducido en el capítulo 11, cada clase de nodo deberá corresponderse con una clase Perl. Por ejemplo, para una regla como

   ```
   exp '*' exp
   ```

   la acción asociada sería algo parecido a

   ```
   { bless [ $_[1], $_[3]], 'MULT' }
   ```

donde usamos un array anónimo. Mejor aún es usar un hash anónimo:

```
{ bless { LEFT => $_[1], RIGHT => $_[3]}, 'MULT' }
```

Defina formalmente el arbol especificando la gramática árbol correspondiente a su diseño (repase la sección 11.9.1). Introduzca en esta parte la tabla de símbolos. La tabla de símbolos es, como en el compilador de Tutu, una lista de referencias a hashes conteniendo las tablas de símbolos locales a cada bloque. En cada momento, la lista refleja el anidamiento de bloques actual. Es posible que, en la declaración de funciones, le interese crear un nuevo bloque en el que guardar los parámetros, de manera que las variables globales queden a nivel 0, los parámetros de una función a nivel 1 y las variables locales de la función a nivel 2 o superior.

14.19. La Gramática de `yapp` / `yacc`

En esta sección veremos con mas detalle, la sintaxis de `Parse::Yapp`, usando la propia notación `yapp` para describir el lenguaje. Un programa `yapp` 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.

```
yapp:   head body tail
head:   headsec '%%'
headsec: #empty
    |   decls
decls:  decls decl  | decl
body:   rulesec '%%'
rulesec:  rulesec rules | rules
rules:  IDENT ':' rhss ';'
tail:   /*empty*/
    |   TAILCODE
```

14.19.1. La Cabecera

En la cabecera se colocan las declaraciones de variables, terminales, etc.

```
decl:  '\n'
    |   TOKEN typedecl symlist '\n'
    |   ASSOC typedecl symlist '\n'
    |   START ident '\n'
    |   HEADCODE '\n'
    |   UNION CODE '\n'
    |   TYPE typedecl identlist '\n'
    |   EXPECT NUMBER '\n'

typedecl: # empty
    |   '<' IDENT '>'
```

El terminal `START` se corresponde con una declaración `%start` indicando cual es el símbolo de arranque de la gramática. Por defecto, el símbolo de arranque es el primero de la gramática.

El terminal `ASSOC` está por los terminales que indican precedencia y asociatividad. Esto se ve claro si se analiza el contenido del fichero `YappParse.yp` (480) en el que se puede encontrar el código del analizador léxico del módulo `Parse::Yapp`. El código dice:

```
...
if($lexlevel == 0) {# In head section
        $$input=~/\G%(left|right|nonassoc)/gc
```

```
        and return('ASSOC',[ uc($1), $lineno[0] ]);
            $$input=~/\G%(start)/gc
        and return('START',[ undef, $lineno[0] ]);
            $$input=~/\G%(expect)/gc
        and return('EXPECT',[ undef, $lineno[0] ]);
            $$input=~/\G%{/gc
    ...
```

La variable `$lexlevel` indica en que sección nos encontramos: cabecera, cuerpo o cola. El terminal `EXPECT` indica la presencia de una declaración `%expect` en el fuente, la cual cuando es seguida de un número indica el numero de conflictos shift-reduce que cabe esperar. Use `EXPECT` si quiere silenciar las advertencias de `yapp` sobre la presencia de conflictos cuya resolución automática considere correcta.

14.19.2. La Cabecera: Diferencias entre `yacc` y `yapp`

Las declaraciones de tipo correspondientes a `%union` y a las especificaciones de tipo entre símbolos menor mayor (`<tipo>`) en declaraciones `token` y `%type` no son usadas por `yapp`. Estas declaraciones son necesarias cuando el código de las acciones semánticas se escribe en `C` como es el caso de `yacc` y `bison`. Sigue un ejemplo de programa `yacc/bison` que usa declaraciones `%union` y de tipo para los atributos:

```
 1 %{
 2 #include <stdio.h>
 3
 4 #define CLASE(x) ((x == 1)?"global":"local")
 5 #define TIPO(x) ((x == 1)?"float":"integer")
 6 %}
 7
 8 %union {
 9   int n;   /* enumerado */
10   char *s; /* cadena */
11 }
12
13 %token <n> FLOAT INTEGER
14 %token <n> GLOBAL
15 %token <n> LOCAL
16 %token <s> NAME
17 %type <n> class type
18
19 %%
```

La declaración `%union` de la línea 8 indica que los atributos son de dos tipos: enteros y punteros a caracteres. El nombre del campo es posteriormente usado en las declaraciones de las líneas 13-17 para indicar el tipo del atributo asociado con la variable o con el terminal. Así, la declaración de la línea 13 indica que los terminales `FLOAT` e `INTEGER` son de tipo entero, mientras que la declaración de la línea 16 nos dice que el terminal `NAME` es de tipo cadena.

```
29 class
30   : GLOBAL { $$ = 1; }
31   | LOCAL  { $$ = 2; }
32   ;
33
34 type
35   : FLOAT   { $$ = 1; }
36   | INTEGER { $$ = 2; }
37   ;
```

La información proveída sobre los tipos permite a `yacc` introducir automáticamente en el código `C` producido los *typecasting* o ahormados para las asignaciones de las líneas 30-31 y 35-36. Obsérve que en `yacc` el atributo de la variable en la parte izquierda se denota por `$$`.

Otra diferencia entre `yacc` y `yapp` es que en `yacc` los atributos de la parte derecha no constituyen un vector, denotándose por `$1`, `$2`, `$3` ...

En ocasiones `yacc` no puede determinar el tipo de un atributo. En particular cuando se habla del atributo asociado con una acción intermedia, ya que esta no tiene variable sintáctica asociada explícitamente o bien cuando se habla de los atributos de símbolos que están a la izquierda de la reducción actual (véase la sección 14.13). Los atributos de símbolos a la izquierda de la producción actual se denotan en `yacc` por números no positivos `$0`, `$-1`, `$-2` ....

En estos casos el programador deberá especificar explícitamente el tipo del atributo usando la notación `$<tipo>#`. Donde `tipo` es uno de los campos de la `union` y `#` es el numeral del símbolo correspondiente:

```
39 namelist
40   : NAME   { printf("%s de clase %s, tipo %s\n",$1,CLASE($<n>-1),TIPO($<n>0)); }
41   | namelist ',' NAME
42       { printf("%s de clase %s, tipo %s\n",$3,CLASE($<n>-1),TIPO($<n>0)); }
43   ;
44 %%
```

14.19.3. El Cuerpo

El cuerpo de un programa `yapp` contiene la gramática y las acciones

```
rhss:   rhss '|' rule | rule
rule:   rhs prec epscode | rhs
rhs:    #empty
    |   rhselts
rhselts: rhselts rhselt | rhselt
rhselt: symbol | code
prec: PREC symbol
epscode: # vacio
    |   code
code:   CODE
```

Las acciones semánticas (variable sintáctica `code` y terminal `CODE`) se ejecutan siempre que ocurre una reducción por una regla y, en general, devuelven un valor semántico. El código de la acción se copia verbatim en el analizador. La estrategia usada por el analizador léxico es contar las llaves abrir y cerrar en el texto. Véase el correspondiente fragmento del analizador léxico:

```
    ....
    $lineno[0]=$lineno[1];
    ....
    $$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(2,"Unmatched { opened line $lineno[0]",-1);
        $code = substr($$input,$from,pos($$input)-$from-1);
        $lineno[1]+= $code=~tr/\n//;
        return('CODE',[ $code, $lineno[0] ]);
    };
```

Las llaves dentro de cadenas y comentarios no son significativas en la cuenta. El problema es que el reconocimiento de cadenas en Perl es mas difícil que en otros lenguajes: existe toda una variedad de formas de denotar una cadena. Por tanto, si el programador usuario de `yapp` necesita escribir una llave dentro de una cadena de doble comilla, deberá escaparla. Si la cadena es de simple comilla escaparla no es solución, pues aparecería el símbolo de escape en la cadena. En ese caso se deberá añadir un comentario con la correspondiente falsa llave. Siguen algunos ejemplos tomadados de la documentación de `Parse::Yapp`

```
"{ My string block }"
"\{ My other string block \}"
qq/ My unmatched brace \} /

# Casamos con el siguiente: {
q/ for my closing brace } / #

q/ My opening brace { /
# debe cerrarse: }
```

Ejercicio 14.19.1. *Genere programas de prueba* `yapp` *con cadenas que produzcan confusión en el analizador y observe el comportamiento. Pruébelas en las diferentes secciones en las que puede ocurrir código: en la cabecera, en el cuerpo y en la cola.*

14.19.4. La Cola: Diferencias entre `yacc` y `yapp`

La cola de un program `yapp` contiene las rutinas de soporte.

```
tail:   /*empty*/
    |   TAILCODE
```

el terminal `TAILCODE` al igual que los terminales `CODE` y `HEADCODE` indican que en ese punto se puede encontrar código Perl. La detección de `TAILCODE` y `HEADCODE` son mas sencillas que las de `CODE`.

La cola de un programa `yacc` es similar. Para el programa `yacc` cuya cabecera y cuerpo se mostraron en la sección 14.19.2 la cola es:

```
%%

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);
}
```

La declaración del manejador de fichero `yyin` en la línea 14 referencia el archivo de entrada para el analizador. La variable (comentada, línea 7) `yydebug` controla la información para la depuración de la gramática. Para que sea realmente efectiva, el programa deberá además compilarse definiendo la macro `YYDEBUG`. Sigue un ejemplo de `Makefile`:

```
inherited: y.tab.c lex.yy.c
   gcc -DYYDEBUG=1 -g -o inherited1 y.tab.c lex.yy.c
y.tab.c y.tab.h: inherited1.y
   yacc -d -v inherited1.y
lex.yy.c: inherited1.l y.tab.h
   flex -l inherited1.l
clean:
   - rm -f y.tab.c lex.yy.c *.o core inherited1
```

Al compilar tenemos:

```
pl@nereida:~/src/inherited$ make
yacc -d -v inherited1.y
flex -l inherited1.l
gcc -DYYDEBUG=1 -g -o inherited1 y.tab.c lex.yy.c
pl@nereida:~/src/inherited$ ls -ltr
total 232
-rw-r-----  1 pl users   242 Dec 10  2003 Makefile
-rw-r-----  1 pl users   404 Dec 10  2003 inherited1.l
-rw-r-----  1 pl users   878 Dec 10  2003 inherited1.y
-rw-rw----  1 pl users  1891 Jan 26 15:41 y.tab.h
-rw-rw----  1 pl users 30930 Jan 26 15:41 y.tab.c
-rw-rw----  1 pl users  2365 Jan 26 15:41 y.output
-rw-rw----  1 pl users 44909 Jan 26 15:41 lex.yy.c
-rwxrwx--x  1 pl users 56336 Jan 26 15:41 inherited1
```

14.19.5. El Análisis Léxico en `yacc`: `flex`

El analizador léxico para `yacc` desarrollado en las secciones anteriores ha sido escrito usando la variante `flex` del lenguaje `LEX`. Un programa `flex` tiene una estructura similar a la de un program `yacc` con tres partes: cabeza, cuerpo y cola separados por `%%`. Veamos como ejemplo de manejo de `flex`, los contenidos del fichero flex `inherited1.l` utilizado en las secciones anteriores:

```
%{
#include <string.h>
#include "y.tab.h"
%}
id [A-Za-z_][A-Za-z_0-9]*
white [ \t\n]+
%%
global    {  return GLOBAL; }
local     {  return LOCAL; }
float     {  return FLOAT; }
int       {  return INTEGER; }
{id}      {  yylval.s = strdup(yytext); return NAME; }
{white}   { ; }
,         { return yytext[0]; }
.         { fprintf(stderr,"Error. carácter inesperado.\n"); }
%%
int yywrap() { return 1; }
```

La cabeza contiene declaraciones `C` asi como definiciones regulares. El fichero `y.tab.h` que es incluído en la línea 3, fué generado por `yacc` y contiene, entre otras cosas, la información recolectada por `yacc` sobre los tipos de los atributos (declaración `%union`) y la enumeración de los terminales. Es, por tanto, necesario que la compilación con `yacc` preceda a la compilación con `flex`. La información en `y.tab.h` es usada por el analizador léxico para "sincronizarse" con el analizador sintáctico. Se definen en las líneas 5 y 6 las macros para el reconocimiento de identificadores (`id`) y blancos (`white`). Estas macros son llamadas en el cuerpo en las líneas 12 y 13. La estructura del cuerpo consiste en parejas formadas por una definición regular seguidas de una acción. La variable `yylval` contiene el atributo asociado con el terminal actual. Puesto que el token `NAME` fué declarado del tipo cadena (véase 14.19.2), se usa el correspondiente nombre de campo `yylval.s`. La cadena que acaba de casar queda guardada en la variable `yytext`, y su longitud queda en la variable entera global `yyleng`.

Una vez compilado con `flex` el fuente, obtenemos un fichero denominado `lex.yy.c`. Este fichero contiene la rutina `yylex()` que realiza el análisis léxico del lenguaje descrito.

La función `yylex()` analiza las entradas, buscando la secuencia mas larga que casa con alguna de las expresiones regulares y ejecuta la correspondiente acción. Si no se encuentra ningun emparejamiento se ejecuta la regla "por defecto", que es:

```
(.|\n)  { printf("%s",yytext); }
```

Si encuentran dos expresiones regulares con las que la cadena mas larga casa, elige la que figura primera en el programa `flex`.

Una vez que se ha ejecutado la correspondiente acción, `yylex()` continúa con el resto de la entrada, buscando por subsiguientes emparejamientos. Asi continúa hasta encontrar un final de fichero en cuyo caso termina, retornando un cero o bien hasta que una de las acciones explicitamente ejecuta una sentencia `return`.

Cuando el analizador léxico alcanza el final del fichero, el comportamiento en las subsiguientes llamadas a `yylex` resulta indefinido. En el momento en que `yylex` alcanza el final del fichero llama a la función `yywrap`, la cual retorna un valor de 0 o 1 según haya mas entrada o no. Si el valor es 0, la función `yylex` asume que la propia `yywrap` se ha encargado de abrir el nuevo fichero y asignarselo a `yyin`.

14.19.6. Práctica: Uso de `Yacc` y `Lex`

Use `yacc` y `flex` para completar los analizadores sintáctico y léxico descritos en las secciones 14.19.2, 14.19.4 y 14.19.5. La gramática en cuestión es similar a la descrita en la sección 14.13. Usando la variable `yydebug` y la macro `YYDEBUG` analize el comportamiento para la entrada `global float x,y`.

14.20. El Analizador Ascendente `Parse::Yapp`

El program `yapp` es un traductor y, por tanto, constituye un ejemplo de como escribir un traductor. El lenguaje fuente es el lenguaje `yacc` y el lenguaje objeto es `Perl`. Como es habitual en muchos lenguajes, el lenguaje objeto se ve "expandidoçon un conjunto de funciones de soporte. En el caso de `yapp` estas funciones de soporte, son en realidad métodos y están en el módulo `Parse::Yapp::Driver`. Cualquier módulo generado por `yapp` hereda de dicho módulo (véase por ejemplo, el módulo generado para nuestro ejemplo de la calculadora, en la sección 14.4).

Como se ve en la figura 14.3, los módulos generados por `yapp` heredan y usan la clase `Parse::Yapp::Driver` la cual contiene el analizador sintáctico LR genérico. Este módulo contiene los métodos de soporte visibles al usuario `YYParse`, `YYData`, `YYErrok`, `YYSemval`, etc.

La figura 14.3 muestra además el resto de los módulos que conforman el "compilador" `Parse::Yapp`. La herencia se ha representado mediante flechas contínuas. Las flechas punteadas indican una relación de uso entre los módulos. El guión `yapp` es un programa aparte que es usado para producir el correspondiente módulo desde el fichero conteniendo la gramática.

(Para ver el contenido de los módulos, descarge `yapp` desde CPAN:

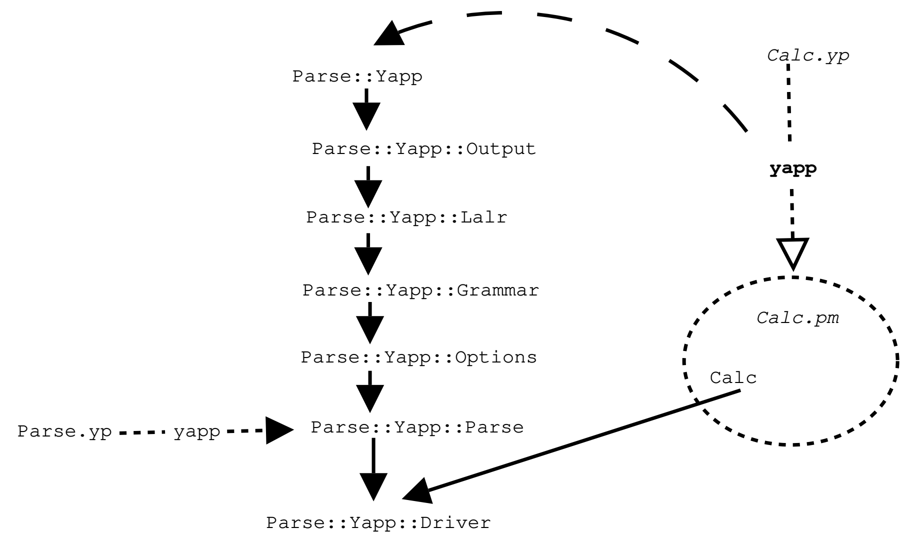

Figura 14.3: Esquema de herencia de `Parse::Yapp`. Las flechas contínuas indican herencia, las punteadas uso. La clase `Calc` es implementada en el módulo generado por `yapp`

`\http://search.cpan.org/~fdesar/Parse-Yapp-1.05/lib/Parse/Yapp.p`

o bien desde uno de nuestros servidores locales; en el mismo directorio en que se guarda la versión HTML de estos apuntes encontrará una copia de Parse-Yapp-1.05.tar.gz). La versión a la que se refiere este capítulo es la 1.05.

El módulo `Parse/Yapp/Yapp.pm` se limita a contener la documentación y descansa toda la tarea de análisis en los otros módulos. El módulo `Parse/Yapp/Output.pm` contiene los métodos `_CopyDriver` y `Output` los cuales se encargan de escribir el analizador: partiendo de un esqueleto genérico rellenan las partes específicas a partir de la información computada por los otros módulos.

El módulo `Parse/Yapp/Options.pm` analiza las opciones de entrada. El módulo `Parse/Yapp/Lalr.pm` calcula las tablas de análisis LALR. Por último el módulo `Parse/Yapp/Grammar` contiene varios métodos de soporte para el tratamiento de la gramática.

El modulo `Parse::Yapp::Driver` contiene el método `YYparse` encargado del análisis. En realidad, el método `YYparse` delega en el método privado `_Parse` la tarea de análisis. Esta es la estructura del analizador genérico usado por `yapp`. Léalo con cuidado y compare con la estructura explicada en la sección 14.5.

```
sub _Parse {
  my($self)=shift;

  my($rules,$states,$lex,$error)
     = @$self{ 'RULES', 'STATES', 'LEX', 'ERROR' };
  my($errstatus,$nberror,$token,$value,$stack,$check,$dotpos)
     = @$self{ 'ERRST', 'NBERR', 'TOKEN', 'VALUE', 'STACK', 'CHECK', 'DOTPOS' };

  $$errstatus=0;
  $$nberror=0;
  ($$token,$$value)=(undef,undef);
  @$stack=( [ 0, undef ] ); # push estado 0
  $$check='';
```

La componente 0 de `@$stack` es el estado, la componente 1 es el atributo.

```

  while(1) {
    my($actions,$act,$stateno);
```

```
17
18     $stateno=$$stack[-1][0];     # sacar el estado en el top de
19     $actions=$$states[$stateno]; # la pila
```

`$states` es una referencia a un vector. Cada entrada `$$states[$stateno]` es una referencia a un hash que contiene dos claves. La clave `ACTIONS` contiene las acciones para ese estado. La clave `GOTOS` contiene los saltos correspondientes a ese estado.

```
20
21     if  (exists($$actions{ACTIONS})) {
22       defined($$token) or do {
23         ($$token,$$value)=&$lex($self); # leer siguiente token
24       };
25
26       # guardar en $act la acción asociada con el estado y el token
27       $act = exists($$actions{ACTIONS}{$$token})?
28          $$actions{ACTIONS}{$$token} :
29          exists($$actions{DEFAULT})? $$actions{DEFAULT} : undef;
30     }
31     else { $act=$$actions{DEFAULT}; }
```

La entrada `DEFAULT` de una acción contiene la acción a ejecutar por defecto.

```
32
33     defined($act) and do {
34       $act > 0 and do { # $act >0 indica shift
35          $$errstatus and do { --$$errstatus; };
```

La línea 35 esta relacionada con la recuperación de errores. Cuando `yapp` ha podido desplazar varios terminales sin que se produzca error considerará que se ha recuperado con éxito del último error.

```
36          # Transitar: guardar (estado, valor)
37          push(@$stack,[ $act, $$value ]);
38          $$token ne ''   #Don't eat the eof
39          and $$token=$$value=undef;
40          next; # siguiente iteración
41       };
```

A menos que se trate del final de fichero, se reinicializa la pareja `($$token, $$value)` y se repite el bucle de análisis. Si `$act` es negativo se trata de una reducción y la entrada `$$rules[-$act]` es una referencia a un vector con tres elementos: la variable sintáctica, la longitud de la parte derecha y el código asociado:

```
43       # $act < 0, indica reduce
44       my($lhs,$len,$code,@sempar,$semval);
45
46       #obtenemos una referencia a la variable,
47       #longitud de la parte derecha, referencia
48       #a la acción
49       ($lhs,$len,$code)=@{$$rules[-$act]};
50       $act or $self->YYAccept();
```

Si `$act` es cero indica una acción de aceptación. El método `YYAccept` se encuentra en `Driver.pm`. Simplemente contiene:

```
sub YYAccept {
  my($self)=shift;
```

```
                ${$$self{CHECK}}='ACCEPT';
                  undef;
              }
```

Esta entrada será comprobada al final de la iteración para comprobar la condición de aceptación (a través de la variable `$check`, la cuál es una referencia).

```
51       $$dotpos=$len; # dotpos es la longitud de la regla
52       unpack('A1',$lhs) eq '@'    #In line rule
53       and do {
54         $lhs =~ /^\@[0-9]+\-([0-9]+)$/
55           or  die "In line rule name '$lhs' ill formed: ".
56                   "report it as a BUG.\n";
57         $$dotpos = $1;
58       };
```

En la línea 52 obtenemos el primer carácter en el nombre de la variable. Las acciones intermedias en `yapp` producen una variable auxiliar que comienza por `@` y casa con el patrón especificado en la línea 54. Obsérvese que el número después del guión contiene la posición relativa en la regla de la acción intermedia.

```
60       @sempar = $$dotpos ?
61          map { $$_[1] } @$stack[ -$$dotpos .. -1 ] : ();
```

El array `@sempar` se inicia a la lista vacía si `$len` es nulo. En caso contrario contiene la lista de los atributos de los últimos `$$dotpos` elementos referenciados en la pila. Si la regla es intermedia estamos haciendo referencia a los atributos de los símbolos a su izquierda.

```
62       $semval = $code ? &$code( $self, @sempar ) :
63                         @sempar ? $sempar[0] : undef;
```

Es en este punto que ocurre la ejecución de la acción. La subrutina referenciada por `$code` es llamada con primer argumento la referencia al objeto analizador `$self` y como argumentos los atributos que se han computado previamente en `@sempar`. Si no existe tal código se devuelve el atributo del primer elemento, si es que existe un tal primer elemento.

El valor retornado por la subrutina/acción asociada es guardado en `$semval`.

```
65       splice(@$stack,-$len,$len);
```

La función `splice` toma en general cuatro argumentos: el array a modificar, el índice en el cual es modificado, el número de elementos a suprimir y la lista de elementos extra a insertar. Aquí, la llamada a `splice` cambia los elementos de `@$stack` a partir del índice `-$len`. El número de elementos a suprimir es `$len`. A continuación se comprueba si hay que terminar, bien porque se ha llegado al estado de aceptación (`$$check eq 'ACCEPT'`) o porque ha habido un error fatal:

```
$$check eq 'ACCEPT' and do { return($semval); };
$$check eq 'ABORT' and  do { return(undef); };
```

Si las cosas van bien, se empuja en la cima de la pila el estado resultante de transitar desde el estado en la cima con la variable sintáctica en el lado izquierdo:

```
$$check eq 'ERROR' or  do {
    push(@$stack, [ $$states[$$stack[-1][0]]{GOTOS}{$lhs}, $semval ]);
    $$check='';
    next;
};
```

La expresión `$$states[$$stack[-1][0]]` es una referencia a un hash cuya clave `GOTOS` contiene una referencia a un hash conteniendo la tabla de transiciones del éstado en la cima de la pila (`$stack[-1][0]`). La entrada de clave `$lhs` contiene el estado al que se transita al ver la variable sintáctica de la izquierda de la regla de producción. El atributo asociado es el devuelto por la acción: `$semval`.

```
      $$check='';

    }; # fin de defined($act)

    # Manejo de errores: código suprimido
    ...

  }
}#_Parse
```

... y el bucle `while(1)` de la línea 15 continúa. Compare este código con el seudo-código introducido en la sección 14.5.

14.21. Práctica: `YappParse.yp`

Repase los fuentes de Yapp. Puede obtenerlos en

`http://search.cpan.org/~fdesar/Parse-Yapp-1.05/lib/Parse/Yapp.pm`

o bien en nuestros servidores locales, por ejemplo en el mismo directorio en que se guarda la versión HTML de estos apuntes encontrará una copia de Parse-Yapp-1.05.tar.gz. La versión a la que se refiere este capítulo es la 1.05.

Estudie el fuente del fichero `YappParse.yp`. Este fichero contiene la gramática `yapp` del lenguaje `yacc`. Además de las dos rutinas de soporte típicas, la de tratamiento de errores `_Error` y la de análisis léxico `_Lexer`, el fichero contiene una subrutina para el manejo de las reglas `_AddRules` y otra rutina `Parse` la cuál actúa como *wrapper* o filtro sobre el analizador `YYParse`.

Durante el análisis sintáctico de un programa `yapp` se construye una estructura de datos para la posterior manipulación y tratamiento de la gramática. Por ejemplo, para la gramática:

```
%%
S:  # empty
    |   'a' S 'b'  { print "S -> a S b\n" }
;
%%
```

La siguiente estructura de datos es construida:

```
$VAR1 = {
  'SYMS' => { 'S' => 2, '\'b\'' => 3, '\'a\'' => 3 },
  'NULL' => { 'S' => 1 },
  'EXPECT' => 0,
  'RULES' => [
    [ '$start', [ 'S', 'Special Symbol' ], undef, undef ],
    [ 'S', [], undef, undef ],
    [
      'S',
      [ '\'a\'', 'S', '\'b\'' ], undef, [ ' print "S -> a S b\\n" ', 3 ]
    ]
  ],
```

```
  'HEAD' => undef,
  'TAIL' => [ ' sub _Error { ...  } sub _Lexer { ...  } ...  ', 5 ],
  'PREC' => {},
  'START' => 'S',
  'TERM' => { '\'b\'' => undef, '\'a\'' => undef },
  'NTERM' => { 'S' => [ '1', '2' ] }
};
```

Explique el significado de los diferentes componentes de la estructura de datos resultante. Las componentes del hash que aparece arriba se corresponden con diversas variables usadas por `YYParse` durante el análisis. La correspondencia se establece dentro del método `Parse` cuando se hace la asignación:

```
   @$parsed{ 'HEAD', 'TAIL', 'RULES', 'NTERM', 'TERM',
             'NULL', 'PREC', 'SYMS',  'START', 'EXPECT' }
    =       (  $head,  $tail,  $rules,  $nterm,  $term,
              $nullable, $precterm, $syms, $start, $expect);
```

esta asignación es la que crea el hash. Las variables con identificadores en minúsculas son usadas en el analizador. Son visibles en todo el fichero ya que, aunque declaradas léxicas, su declaración se encuentra en la cabecera del analizador:

```
%{
require 5.004;

use Carp;

my($input,$lexlevel,@lineno,$nberr,$prec,$labelno);
my($syms,$head,$tail,$token,$term,$nterm,$rules,$precterm,$start,$nullable);
my($expect);

%}
```

Responda a las siguientes preguntas:

- ¿Que se guarda en `SYMS`?
- En este ejemplo los valores del hash `TERM` esta indefinidos. En general contienen un array anónimo con información sobre el terminal. ¿Que información exactamente?
- ¿Que contiene el hash `PREC`? ¿tiene algo que ver con la directiva `%prec`?
- ¿Cuál es el significado del 5 en la tercera componente de `TAIL`? ¿Qué contiene el array anónimo de clave `HEAD`?
- ¿Cómo es la estructura de la lista representando una regla? ¿Qué significan los diferentes elementos de la lista?
- ¿Cuál es el significado del hash con clave `NULL`? ¿En que forma es usado durante el análisis para controlar posibles errores?
- ¿Cómo se denotan las variables sintácticas asociadas con acciones intermedias? ¿Sabría señalar el lugar en `YappParse.yp` en que se trata con esta situación?
- ¿Cuál es el significado de la clave `EXPECT`?
- ¿Qué referencian los índices en el array anónimo asociado con clave `S` en `NTERM`?
- ¿Que limitaciones observa en `YappParse.yp`?

14.22. Práctica: El Análisis de las Acciones

Modifique el código de `YappParse.yp` para que el análisis léxico de las secciones de código (`HEADCODE`, `CODE` y `TAILCODE`) se haga a través de las correspondientes rutinas proveida como parámetros para el análisis por el usuario. La idea es ofrecer un primer paso que facilite la generación de analizadores en diferentes lenguajes `Perl`, `C`, etc.

Estudie el módulo `Text::Balanced`. Basándose en las funciones
`extract_codeblock` y `extract_quotelike`
del módulo `Text::Balanced`, resuelva el problema del reconocimiento de código Perl dentro del analizador léxico de `Parse::Yapp`, evitando forzar al usuario en la escritura de "llaves fantasma". Compare el rendimiento de esta solución con la que provee `Yapp`. Para analizar el *rendimiento* use el módulo `Benchmark`.

¿Cuáles son sus conclusiones? ¿Qué es mejor?

14.23. Práctica: Autoacciones

Extienda `Parse::Yapp` con una directiva `%autoaction CODE` la cuál cambia la acción por defecto. Cuando una regla de producción no tenga una acción asociada, en vez de ejecutarse la acción `yapp` por defecto se ejecutará el código especificado en `CODE`. La directiva podrá aparecer en la parte de cabecera o en el cuerpo del programa `yapp` en una sóla línea aparte. Si aparece en el cuerpo no debe hacerlo en medio de una regla.

Sigue un ejemplo de uso:

```
%{
use Data::Dumper;
my %tree_name = ('=' => 'eq', '+' => 'plus', '-' => 'minus',
                 '*' => 'times', '/' => 'divide');
%}
%right  '='
%left   '-' '+'
%left   '*' '/'
%left   NEG
%autoaction { [$tree_name{$_[2]}, $_[1], $_[3]] }

%%
input:               { undef }
        | input line { undef }
;

line:     '\n'         { undef }
        | exp '\n'     { print Dumper($_[1]); }
        | error '\n'   { $_[0]->YYErrok }
;

exp:        NUM                { $_[1] }
        |   VAR                { $_[1] }
        |   VAR '=' exp | exp '+' exp | exp '-' exp | exp '*' exp | exp '/' exp
        |   '-' exp %prec NEG   { ['neg', $_[2]] }
        |   '(' exp ')'        { $_[2] }
;

%%
```

y un ejemplo de ejecución:

```
$ ./useautoaction1.pl
2+3*4
^D
$VAR1 = [
          'plus',
          '2',
          [ 'times', '3', '4' ]
        ];
```

Analice la adecuación de los mensajes de error emitidos por el compilador de Perl cuando el código en la auto-acción contiene errores. ¿Son apropiados los números de línea?

Tenga en cuenta los siguientes consejos:

- Cuando compile con `yapp` su módulo use una orden como: `yapp -m Parse::Yapp::Parse Parse.yp`. Este es un caso en que el nombre del fichero de salida (`Parse.pm`) y el nómbre del package `Parse::Yapp::Parse` no coinciden. Este es un caso en que el nombre del fichero de salida (`Parse.pm`) y el nómbre del package `Parse::Yapp::Parse` no coinciden.
- Ahora tiene dos versiones de `Parse::Yapp` en su ordenador. El compilador de Perl va a intentar cargar la instalada. Para ello en su versión del script `yapp` puede incluir una línea que le indique al compilador que debe buscar primero en el lugar en el que se encuentra nuestra librería:

  ```
  BEGIN { unshift @INC, '/home/lhp/Lperl/src/yapp/Parse-Yapp-Auto/lib/' }
  ```

- ¿Qué estrategia a seguir? Una posibilidad es "hacerle creer" al resto de los módulos en `Yapp` que el usuario ha escrito el código de la autoacción en aquellas reglas en las que no existe código explícito asociado. Es posible realizar esta práctica modificando sólo el fichero `YappParse.yp`. El código original `Yapp` usa `undef` para indicar, en el campo adecuado, que una acción no fué definida. La idea es sustituir ese `undef` por el código asociado con la autoacción:

  ```
  my($code)= $autoaction? $autoaction:undef;
  ```

14.24. Práctica: Nuevos Métodos

Continuemos extendiendo `Yapp`.

- Introduzca en el módulo `Driver.pm` de `Yapp` un método `YYLhs` que devuelva el identificador de la variable sintáctica en el lado izquierdo de la regla de producción por la que se está reduciendo.
- Para tener disponible el lado izquierdo deberá modificar la conducta del analizador LALR (subrutina `_Parse`) para que guarde como un atributo el identificador de dicho noterminal.
- ¿Que identificador se devuelve asociado con las acciones intermedias?

Sigue un ejemplo de como programar haciendo uso de esta y la anterior extensión:

```
%right  '='
%left   '-' '+'
%left   '*' '/'
%left   NEG
%autoaction { my $n = $#_;  bless [@_[1..$n]], $_[0]->YYLhs }
%%
input:
        |   input line
;
line:     '\n'       { }
```

```
        | exp '\n'   { [ $_[1] ] }
        | error '\n' { }
;
exp:        NUM         |   VAR         |   VAR '=' exp
        |   exp '+' exp |   exp '-' exp |   exp '*' exp |   exp '/' exp
        |   '-' exp %prec NEG
        |   '(' exp ')' { [ $_[2] ] }
;
%%
...
```

Veamos la ejecución correspondiente al ejemplo anterior:

```
$ ./uselhs2.pl
2+3*4
$VAR1 = bless(
[
  bless( [], 'input' ),
  [
    bless( [
      bless( [ '2' ], 'exp' ),
      '+',
      bless( [
        bless( [ '3' ], 'exp' ), '*', bless( [ '4' ], 'exp' ) ], 'exp' )
    ], 'exp' )
  ]
], 'input' );
```

14.25. Práctica: Generación Automática de Árboles

Partiendo de la práctica anterior, introduzca una directiva `%autotree` que de lugar a la construcción del árbol de análisis concreto. La acción de construcción del árbol:

```
{ my $n = $#_;  bless [@_[1..$n]], $_[0]->YYLhs }
```

se ejecutará para cualquier regla que no tenga una acción explícita asociada.

14.26. Recuperacion de Errores: Visión Detallada

La subrutina `_Parse` contiene el algoritmo de análisis LR genérico. En esta sección nos concentraremos en la forma en la que se ha implantado en `yapp` la recuperación de errores.

```
sub _Parse {
  my($self)=shift;
  ...
  $$errstatus=0; $$nberror=0;
```

La variable `$$errstatus` nos indica la situación con respecto a la recuperación de errores. La variable `$$nberror` contiene el número total de errores.

```
  ($$token,$$value)=(undef,undef);
  @$stack=( [ 0, undef ] ); $$check='';
  while(1) {
    my($actions,$act,$stateno);
```

```
9     $stateno=$$stack[-1][0];
10    $actions=$$states[$stateno];
11
12    if  (exists($$actions{ACTIONS})) {
13      defined($$token) or  do { ($$token,$$value)=&$lex($self); };
14      ...
15    }
16    else { $act=$$actions{DEFAULT}; }
```

Si `$act` no esta definida es que ha ocurrido un error. En tal caso no se entra a estudiar si la acción es de desplazamiento o reducción.

```
17    defined($act) and do {
18      $act > 0 and do { #shift
19          $$errstatus and  do { --$$errstatus; };
20          ...
21          next;
22      };
23      #reduce
24      ....
25      $$check eq 'ERROR' or  do {
26        push(@$stack, [ $$states[$$stack[-1][0]]{GOTOS}{$lhs}, $semval ]);
27        $$check='';
28        next;
29      };
30      $$check='';
31    };
```

Si `$$errstatus` es cero es que estamos ante un nuevo error:

```
32    #Error
33    $$errstatus or   do {
34      $$errstatus = 1;
35      &$error($self);
36      $$errstatus # if 0, then YYErrok has been called
37      or  next;   # so continue parsing
38      ++$$nberror;
39    };
```

Como el error es "nuevo" se llama a la subrutina de tratamiento de errores `&$error`. Obsérvese que no se volverá a llamar a la rutina de manejo de errores hasta que `$$errstatus` vuelva a alcanzar el valor cero. Puesto que `&$error` ha sido escrita por el usuario, es posible que este haya llamado al método `YYErrok`. Si ese es el caso, es que el programador prefiere que el análisis continúe como si la recuperación de errores se hubiera completado.

Ahora se pone `$$errstatus` a 3:

```
47    $$errstatus=3;
```

Cada vez que se logre un desplazamiento con éxito `$$errstatus` será decrementado (línea 19).

A continuación se retiran estados de la pila hasta que se encuentre alguno que pueda transitar ante el terminale especial `error`:

```
48    while(@$stack
49        and (not exists($$states[$$stack[-1][0]]{ACTIONS})
50        or  not exists($$states[$$stack[-1][0]]{ACTIONS}{error})
51        or  $$states[$$stack[-1][0]]{ACTIONS}{error} <= 0)) {
```

```
52       pop(@$stack);
53     }
54     @$stack or do {
55       return(undef);
56     };
```

Si la pila quedó vacía se devuelve `undef`. En caso contrario es que el programador escribió alguna regla para la recuperación de errores. En ese caso, se transita al estado correspondiente:

```
57     #shift the error token
58     push(@$stack, [ $$states[$$stack[-1][0]]{ACTIONS}{error}, undef ]);
59   }
60   #never reached
61   croak("Error in driver logic. Please, report it as a BUG");
62 }#_Parse
```

Un poco antes tenemos el siguiente código:

```
41     $$errstatus == 3  #The next token is not valid: discard it
42     and  do {
43       $$token eq ''  # End of input: no hope
44       and  do { return(undef); };
45       $$token=$$value=undef;
46     };
```

Si hemos alcanzado el final de la entrada en una situación de error se abandona devolviendo `undef`.

Ejercicio 14.26.1. *Explique la razón para el comentario de la línea 41. Si* `$$errstatus` *es 3, el último terminal no ha producido un desplazamiento correcto. ¿Porqué?*

A continuación aparecen los códigos de los métodos implicados en la recuperación de errores:

```
sub YYErrok {
  my($self)=shift;

  ${$$self{ERRST}}=0;
    undef;
}
```

El método `YYErrok` cambia el valor referenciado por `$errstatus`. De esta forma se le da al programador `yapp` la oportunidad de anunciar que es muy probable que la fase de recuperación de errores se haya completado.

Los dos siguientes métodos devuelven el número de errores hasta el momento (`YYNberr`) y si nos encontramos o no en fase de recuperación de errores (`YYRecovering`):

```
sub YYNberr {
  my($self)=shift;

  ${$$self{NBERR}};
}

sub YYRecovering {
  my($self)=shift;

  ${$$self{ERRST}} != 0;
}
```

14.27. El Generador de Analizadores byacc

Existe una version del `yacc` de Berkeley que permite producir código para Perl:

```
> byacc -V
byacc: Berkeley yacc version 1.8.2 (C or perl)
```

Se trata por tanto de un generador de analizadores LALR. Es bastante compatible con `AT&T yacc`. Puedes encontrar una versión en formato `tar.gz` en nuestro servidor http://nereida.deioc.ull.es/~pl/pyacc-pack.tgz o también desde http://www.perl.com/CPAN/src/misc/.

El formato de llamada es:

```
byacc  [ -CPcdlrtv ] [ -b file_prefix ] [ -p symbol_prefix ] filename
```

Las opciones `C` o `c` permiten generar código C. Usando `-P` se genera código Perl. Las opciones `d` y `v` funcionan como es usual en `yacc`. Con `t` se incorpora código para la depuración de la gramática. Si se especifica `l` el código del usuario no es insertado. La opción `r` permite generar ficheros separados para el código y las tablas. No la use con Perl.

Fichero conteniendo la gramática:

```
%{
%}

%token INT EOL
%token LEFT_PAR RIGHT_PAR
%left PLUS MINUS
%left MULT DIV

%%
start: |
start input
;

input: expr EOL { print $1 . "\n"; }
| EOL
;

expr: INT { $p->mydebug("INT -> Expr!"); $$ = $1; }
| expr PLUS expr { $p->mydebug("PLUS -> Expr!"); $$ = $1 + $3; }
| expr MINUS expr { $p->mydebug("MINUS -> Expr!"); $$ = $1 - $3; }
| expr MULT expr { $p->mydebug("MULT -> Expr!"); $$ = $1 * $3; }
| expr DIV expr { $p->mydebug("DIV -> Expr!"); $$ = $1 / $3; }
| LEFT_PAR expr RIGHT_PAR { $p->mydebug("PARENS -> Expr!"); $$ = $2; }
;
%%

sub yyerror {
    my ($msg, $s) = @_;
    my ($package, $filename, $line) = caller;

    die "$msg at <DATA> \n$package\n$filename\n$line\n";
}

sub mydebug {
    my $p = shift;
    my $msg = shift;
```

```
    if ($p->{'yydebug'})
    {
        print "$msg\n";
    }
}
```

La compilación con `byacc` del fichero `calc.y` conteniendo la descripción de la gramática produce el módulo Perl conteniendo el analizador.

```
> ls -l
total 12
-rw-r-----    1 pl       casiano        47 Dec 29  2002 Makefile
-rw-r-----    1 pl       casiano       823 Dec 29  2002 calc.y
-rwxr-x--x    1 pl       casiano       627 Nov 10 15:37 tokenizer.pl
> cat Makefile
MyParser.pm: calc.y
        byacc -d -P MyParser $<

> make
byacc -d -P MyParser calc.y
> ls -ltr
total 28
-rw-r-----    1 pl       casiano       823 Dec 29  2002 calc.y
-rw-r-----    1 pl       casiano        47 Dec 29  2002 Makefile
-rwxr-x--x    1 pl       casiano       627 Nov 10 15:37 tokenizer.pl
-rw-rw----    1 pl       users          95 Nov 16 12:49 y.tab.ph
-rw-rw----    1 pl       users        9790 Nov 16 12:49 MyParser.pm
```

Observe que la opción `-P` es la que permite producir código Perl. Anteriormente se usaba la opción `-p`. Esto se hizo para mantener la compatibilidad con otras versiones de `yacc` en las que la opción `-p` se usa para cambiar el prefijo por defecto (`yy`). Ese es el significado actual de la opción `-p` en `perl-byacc`.

El fichero `y.tab.ph` generado contiene las definiciones de los *tokens*:

```
 cat y.tab.ph
$INT=257;
$EOL=258;
$LEFT_PAR=259;
$RIGHT_PAR=260;
$PLUS=261;
$MINUS=262;
$MULT=263;
$DIV=264;
```

El programa `tokenizer.pl` contiene la llamada al analizador y la definición del analizador léxico:

```
> cat tokenizer.pl
#!/usr/local/bin/perl5.8.0

require 5.004;
use strict;
use Parse::YYLex;
use MyParser;

print STDERR "Version $Parse::ALex::VERSION\n";
```

```
my (@tokens) = ((LEFT_PAR => '\(',
                 RIGHT_PAR => '\)',
                 MINUS => '-',
                 PLUS => '\+',
                 MULT => '\*',
                 DIV => '/',
                 INT => '[1-9][0-9]*',
                 EOL => '\n',
                 ERROR => '.*'),
                sub { die "!can\'t analyze: \"$_[1]\"\n!"; });

my $lexer = Parse::YYLex->new(@tokens);

sub yyerror
{
    die "There was an error:" . join("\n", @_). "\n";
}

my $debug = 0;
my $parser = new MyParser($lexer->getyylex(), \&MyParser::yyerror , $debug);
$lexer->from(\*STDIN);
$parser->yyparse(\*STDIN);
```

El módulo `Parse::YYLex` contiene una versión de `Parse::Lex` que ha sido adaptada para funcionar con `byacc`. Todas las versiones de `yacc` esperan que el analizador léxico devuelva un *token* numérico, mientras que `Parse::Lex` devuelbe un objeto de la clase *token*. Veamos un ejemplo de ejecución:

```
> tokenizer.pl
Version 2.15
yydebug: state 0, reducing by rule 1 (start :)
yydebug: after reduction, shifting from state 0 to state 1
3*(5-9)
yydebug: state 1, reading 257 (INT)
yydebug: state 1, shifting to state 2
yydebug: state 2, reducing by rule 5 (expr : INT)
INT -> Expr!
yydebug: after reduction, shifting from state 1 to state 6
yydebug: state 6, reading 263 (MULT)
yydebug: state 6, shifting to state 11
yydebug: state 11, reading 259 (LEFT_PAR)
yydebug: state 11, shifting to state 4
yydebug: state 4, reading 257 (INT)
yydebug: state 4, shifting to state 2
yydebug: state 2, reducing by rule 5 (expr : INT)
INT -> Expr!
yydebug: after reduction, shifting from state 4 to state 7
yydebug: state 7, reading 262 (MINUS)
yydebug: state 7, shifting to state 10
yydebug: state 10, reading 257 (INT)
yydebug: state 10, shifting to state 2
yydebug: state 2, reducing by rule 5 (expr : INT)
INT -> Expr!
yydebug: after reduction, shifting from state 10 to state 15
yydebug: state 15, reading 260 (RIGHT_PAR)
```

```
yydebug: state 15, reducing by rule 7 (expr : expr MINUS expr)
MINUS -> Expr!
yydebug: after reduction, shifting from state 4 to state 7
yydebug: state 7, shifting to state 13
yydebug: state 13, reducing by rule 10 (expr : LEFT_PAR expr RIGHT_PAR)
PARENS -> Expr!
yydebug: after reduction, shifting from state 11 to state 16
yydebug: state 16, reducing by rule 8 (expr : expr MULT expr)
MULT -> Expr!
yydebug: after reduction, shifting from state 1 to state 6
yydebug: state 6, reading 258 (EOL)
yydebug: state 6, shifting to state 8
yydebug: state 8, reducing by rule 3 (input : expr EOL)
-12
yydebug: after reduction, shifting from state 1 to state 5
yydebug: state 5, reducing by rule 2 (start : start input)
yydebug: after reduction, shifting from state 0 to state 1
yydebug: state 1, reading 0 (end-of-file)
```

Apéndice

Código de `Calc.yp`

```
#
# Calc.yp
#
# Parse::Yapp input grammar example.
#
# This file is PUBLIC DOMAIN
#
#
%right  '='
%left   '-' '+'
%left   '*' '/'
%left   NEG
%right  '^'

%%
input:  #empty
        |   input line  { push(@{$_[1]},$_[2]); $_[1] }
;

line:       '\n'                { $_[1] }
        |   exp '\n'            { print "$_[1]\n" }
        |   error '\n' { $_[0]->YYErrok }
;

exp:        NUM
        |   VAR                 { $_[0]->YYData->{VARS}{$_[1]} }
        |   VAR '=' exp         { $_[0]->YYData->{VARS}{$_[1]}=$_[3] }
        |   exp '+' exp         { $_[1] + $_[3] }
        |   exp '-' exp         { $_[1] - $_[3] }
        |   exp '*' exp         { $_[1] * $_[3] }
        |   exp '/' exp         {
                                      $_[3]
                                  and return($_[1] / $_[3]);
                                  $_[0]->YYData->{ERRMSG}
                                    =   "Illegal division by zero.\n";
                                  $_[0]->YYError;
                                  undef
                                }
        |   '-' exp %prec NEG   { -$_[2] }
        |   exp '^' exp         { $_[1] ** $_[3] }
        |   '(' exp ')'         { $_[2] }
;

```

```
%%

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/^[ \t]//;

    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 => 0x10 );
}

```

Código de `Calc.pm`

Este es el resultado de la traducción mediante `yapp` del programa de la calculadora descrito en la página 467. Para ver un resumen de las tablas vaya a la página 477.

```
####################################################################
#
#    This file was generated using Parse::Yapp version 1.05.
#
#        Don’t edit this file, use source file instead.
#
#             ANY CHANGE MADE HERE WILL BE LOST !
#
####################################################################
package Calc;
use vars qw ( @ISA );
use strict;

@ISA= qw ( Parse::Yapp::Driver );
use Parse::Yapp::Driver;

sub new {
        my($class)=shift;
        ref($class)
    and $class=ref($class);

    my($self)=$class->SUPER::new( yyversion => ’1.05’,
                                  yystates => [
 {#State 0
 DEFAULT => -1, GOTOS => { ’input’ => 1 }
 },
 {#State 1
 ACTIONS => { ’NUM’ => 6, ’’ => 4, "-" => 2, "(" => 7, ’VAR’ => 8, "\n" => 5, ’error’ => 9 },
 GOTOS => { ’exp’ => 3, ’line’ => 10
 }
 },
 {#State 2
 ACTIONS => { ’NUM’ => 6, "-" => 2, "(" => 7, ’VAR’ => 8 },
 GOTOS => { ’exp’ => 11 }
 },
 {#State 3
 ACTIONS => { "-" => 12, "^" => 16, "*" => 17, "\n" => 14, "+" => 13, "/" => 15 }
 },
 {#State 4
 DEFAULT => 0
 },
```

```
{#State 5
DEFAULT => -3
},
{#State 6
DEFAULT => -6
},
{#State 7
ACTIONS => { 'NUM' => 6, "-" => 2, "(" => 7, 'VAR' => 8 },
GOTOS => { 'exp' => 18 }
},
{#State 8
ACTIONS => { "=" => 19 },
DEFAULT => -7
},
{#State 9
ACTIONS => { "\n" => 20 }
},
{#State 10
DEFAULT => -2
},
{#State 11
ACTIONS => {
"^" => 16
},
DEFAULT => -13
},
{#State 12
ACTIONS => { 'NUM' => 6, "-" => 2, "(" => 7, 'VAR' => 8
},
GOTOS => { 'exp' => 21 }
},
{#State 13
ACTIONS => { 'NUM' => 6, "-" => 2, "(" => 7, 'VAR' => 8 },
GOTOS => { 'exp' => 22 }
},
{#State 14
DEFAULT => -4
},
{#State 15
ACTIONS => { 'NUM' => 6, "-" => 2, "(" => 7, 'VAR' => 8 },
GOTOS => { 'exp' => 23 }
},
{#State 16
ACTIONS => { 'NUM' => 6, "-" => 2, "(" => 7, 'VAR' => 8 },
GOTOS => { 'exp' => 24 }
},
{#State 17
ACTIONS => { 'NUM' => 6, "-" => 2, "(" => 7, 'VAR' => 8 },
GOTOS => { 'exp' => 25 }
},
{#State 18
ACTIONS => { "-" => 12, "^" => 16, "*" => 17, "+" => 13, "/" => 15, ")" => 26 }
},
```

```
{#State 19
ACTIONS => { 'NUM' => 6, "-" => 2, "(" => 7, 'VAR' => 8 },
GOTOS => { 'exp' => 27 }
},
{#State 20
 DEFAULT => -5
 },
 {#State 21
 ACTIONS => { "^" => 16, "*" => 17, "/" => 15 },
 DEFAULT => -10
 },
 {#State 22
 ACTIONS => { "^" => 16, "*" => 17, "/" => 15 },
 DEFAULT => -9
 },
 {#State 23
 ACTIONS => { "^" => 16 },
 DEFAULT => -12
 },
 {#State 24
 ACTIONS => {
 "^" => 16
 },
 DEFAULT => -14
 },
 {#State 25
 ACTIONS => {
 "^" => 16
 },
 DEFAULT => -11
 },
 {#State 26
 DEFAULT => -15
 },
 {#State 27
 ACTIONS => { "-" => 12, "^" => 16, "*" => 17, "+" => 13, "/" => 15 },
 DEFAULT => -8
 }
],
                                  yyrules  => [
 [#Rule 0
  '$start', 2, undef ],
 [#Rule 1
  'input', 0, undef ],
 [#Rule 2
  'input', 2,
sub
#line 17 "Calc.yp"
{ push(@{$_[1]},$_[2]); $_[1] }
 ],
 [#Rule 3
  'line', 1,
sub
```

```
#line 20 "Calc.yp"
{ $_[1] }
 ],
 [#Rule 4
  'line', 2,
sub
#line 21 "Calc.yp"
{ print "$_[1]\n" }
 ],
 [#Rule 5
  'line', 2,
sub
#line 22 "Calc.yp"
{ $_[0]->YYErrok }
 ],
 [#Rule 6
  'exp', 1, undef
 ],
 [#Rule 7
  'exp', 1,
sub
#line 26 "Calc.yp"
{ $_[0]->YYData->{VARS}{$_[1]} }
 ],
 [#Rule 8
  'exp', 3,
sub
#line 27 "Calc.yp"
{ $_[0]->YYData->{VARS}{$_[1]}=$_[3] }
 ],
 [#Rule 9
  'exp', 3,
sub
#line 28 "Calc.yp"
{ $_[1] + $_[3] }
 ],
 [#Rule 10
  'exp', 3,
sub
#line 29 "Calc.yp"
{ $_[1] - $_[3] }
 ],
 [#Rule 11
  'exp', 3,
sub
#line 30 "Calc.yp"
{ $_[1] * $_[3] }
 ],
 [#Rule 12
  'exp', 3,
sub
#line 31 "Calc.yp"
{
```

```
                                      $_[3]
                                  and return($_[1] / $_[3]);
                                  $_[0]->YYData->{ERRMSG}
                                    =   "Illegal division by zero.\n";
                                  $_[0]->YYError;
                                  undef
                                }
 ],
 [#Rule 13
  'exp', 2,
sub
#line 39 "Calc.yp"
{ -$_[2] }
 ],
 [#Rule 14
  'exp', 3,
sub
#line 40 "Calc.yp"
{ $_[1] ** $_[3] }
 ],
 [#Rule 15
  'exp', 3,
sub
#line 41 "Calc.yp"
{ $_[2] }
 ]
],
                                  @_);
    bless($self,$class);
}

#line 44 "Calc.yp"


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/^[ \t]//;

```

```
    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 => 0x10 );
}

1;
```

Código de aSb.output

Este apéndice muestra el fichero `aSb.output` resultante de traducir mediante `yapp` con la opción `-v` el programa `aSb.yp`:

```
%%
S:                { print "S -> epsilon\n" }
    |   'a' S 'b'  { print "S -> a S b\n" }
;
%%
...
```

Este es el ejemplo que se usó en las secciones 14.2 y 14.3.2. Sigue el fichero `aSb.output`:

```
Rules:
------
0: $start -> S $end
1: S -> /* empty */
2: S -> 'a' S 'b'

States:
-------
State 0:

 $start -> . S $end (Rule 0)
 'a'shift, and go to state 2
 $default reduce using rule 1 (S)
 S go to state 1

State 1:

 $start -> S . $end (Rule 0)
 $end shift, and go to state 3

State 2:

 S -> 'a' . S 'b'(Rule 2)
 'a'shift, and go to state 2
 $default reduce using rule 1 (S)
 S go to state 4

State 3:

 $start -> S $end . (Rule 0)
 $default accept

State 4:
```

```

 S -> 'a' S . 'b'(Rule 2)
 'b'shift, and go to state 5

State 5:

 S -> 'a' S 'b' . (Rule 2)
 $default reduce using rule 2 (S)


Summary:
--------
Number of rules         : 3
Number of terminals     : 3
Number of non-terminals : 2
Number of states        : 6
```

Código de Calc.output

Este apéndice muestra un resumen del fichero `Calc.output` resultante de traducir mediante `yapp` con la opción `-v` el programa `Calc.yp`.

```
Conflicts:
----------
Conflict in state 11 between rule 13 and token ’-’ resolved as reduce.
Conflict in state 11 between rule 13 and token ’^’ resolved as shift.
Conflict in state 11 between rule 13 and token ’*’ resolved as reduce.
Conflict in state 11 between rule 13 and token ’+’ resolved as reduce.
Conflict in state 11 between rule 13 and token ’/’ resolved as reduce.
Conflict in state 21 between rule 10 and token ’-’ resolved as reduce.
Conflict in state 21 between rule 10 and token ’^’ resolved as shift.
Conflict in state 21 between rule 10 and token ’*’ resolved as shift.
Conflict in state 21 between rule 10 and token ’+’ resolved as reduce.
Conflict in state 21 between rule 10 and token ’/’ resolved as shift.
Conflict in state 22 between rule 9 and token ’-’ resolved as reduce.
Conflict in state 22 between rule 9 and token ’^’ resolved as shift.
Conflict in state 22 between rule 9 and token ’*’ resolved as shift.
Conflict in state 22 between rule 9 and token ’+’ resolved as reduce.
Conflict in state 22 between rule 9 and token ’/’ resolved as shift.
Conflict in state 23 between rule 12 and token ’-’ resolved as reduce.
Conflict in state 23 between rule 12 and token ’^’ resolved as shift.
Conflict in state 23 between rule 12 and token ’*’ resolved as reduce.
Conflict in state 23 between rule 12 and token ’+’ resolved as reduce.
Conflict in state 23 between rule 12 and token ’/’ resolved as reduce.
Conflict in state 24 between rule 14 and token ’-’ resolved as reduce.
Conflict in state 24 between rule 14 and token ’^’ resolved as shift.
Conflict in state 24 between rule 14 and token ’*’ resolved as reduce.
Conflict in state 24 between rule 14 and token ’+’ resolved as reduce.
Conflict in state 24 between rule 14 and token ’/’ resolved as reduce.
Conflict in state 25 between rule 11 and token ’-’ resolved as reduce.
Conflict in state 25 between rule 11 and token ’^’ resolved as shift.
Conflict in state 25 between rule 11 and token ’*’ resolved as reduce.
Conflict in state 25 between rule 11 and token ’+’ resolved as reduce.
Conflict in state 25 between rule 11 and token ’/’ resolved as reduce.
Conflict in state 27 between rule 8 and token ’-’ resolved as shift.
Conflict in state 27 between rule 8 and token ’^’ resolved as shift.
Conflict in state 27 between rule 8 and token ’*’ resolved as shift.
Conflict in state 27 between rule 8 and token ’+’ resolved as shift.
Conflict in state 27 between rule 8 and token ’/’ resolved as shift.

Rules:
------
0: $start -> input $end
```

```
1: input -> /* empty */
2: input -> input line
3: line -> '\n'
4: line -> exp '\n'
5: line -> error '\n'
6: exp -> NUM
7: exp -> VAR
8: exp -> VAR '=' exp
9: exp -> exp '+' exp
10: exp -> exp '-' exp
11: exp -> exp '*' exp
12: exp -> exp '/' exp
13: exp -> '-' exp
14: exp -> exp '^' exp
15: exp -> '(' exp ')'

States:
-------
State 0:

 $start -> . input $end (Rule 0)
 $default reduce using rule 1 (input)
 input go to state 1

State 1:

 $start -> input . $end (Rule 0)
 input -> input . line (Rule 2)

 $end shift, and go to state 4
 '(' shift, and go to state 7
 '-' shift, and go to state 2
 '\n' shift, and go to state 5
 NUM shift, and go to state 6
 VAR shift, and go to state 8
 error shift, and go to state 9

 exp go to state 3
 line go to state 10

.... # muchos mas estados

State 27:

 exp -> VAR '=' exp . (Rule 8)
 exp -> exp . '+' exp (Rule 9)
 exp -> exp . '-' exp (Rule 10)
 exp -> exp . '*' exp (Rule 11)
 exp -> exp . '/' exp (Rule 12)
 exp -> exp . '^' exp (Rule 14)

 '*' shift, and go to state 17
 '+' shift, and go to state 13
```

```
 '-' shift, and go to state 12
 '/' shift, and go to state 15
 '^' shift, and go to state 16

 $default reduce using rule 8 (exp)


Summary:
--------
Number of rules         : 16
Number of terminals     : 14
Number of non-terminals : 4
Number of states        : 28
```

Capítulo 15

Código de Parse.yp

```
%{# (c) Copyright Francois Desarmenien 1998-2001, all rights reserved.
# (see COPYRIGHT in Parse::Yapp.pm pod section for use and distribution rights)
#
# Parse/Yapp/Parser.yp: Parse::Yapp::Parser.pm source file
#
# Use: yapp -m 'Parse::Yapp::Parse' -o Parse/Yapp/Parse.pm YappParse.yp
#
# to generate the Parser module.
#
%}

%{
require 5.004;

use Carp;

my($input,$lexlevel,@lineno,$nberr,$prec,$labelno);
my($syms,$head,$tail,$token,$term,$nterm,$rules,$precterm,$start,$nullable);
my($expect);

%}

%%

# Main rule
yapp:   head body tail ;

#Common rules:

symbol: LITERAL {
                        exists($$syms{$_[1][0]})
                    or  do {
                        $$syms{$_[1][0]} = $_[1][1];
                        $$term{$_[1][0]} = undef;
                    };
                    $_[1]
                }
    |   ident   #default action
;

```

```
ident:  IDENT   {
                        exists($$syms{$_[1][0]})
                    or  do {
                        $$syms{$_[1][0]} = $_[1][1];
                        $$term{$_[1][0]} = undef;
                    };
                    $_[1]
                }
;


# Head section:
head:   headsec '%%'
;

headsec:    #empty  #default action
    |       decls   #default action
;

decls:  decls decl  #default action
    |   decl        #default action
;

decl:  '\n'                 #default action
    |   TOKEN typedecl symlist '\n'
            {
                for (@{$_[3]}) {
                    my($symbol,$lineno)=@$_;

                        exists($$token{$symbol})
                    and do {
                        _SyntaxError(0,
                                "Token $symbol redefined: ".
                                "Previously defined line $$syms{$symbol}",
                                $lineno);
                        next;
                    };
                    $$token{$symbol}=$lineno;
                    $$term{$symbol} = [ ];
                }
                undef
            }
    |   ASSOC typedecl symlist '\n'
            {
                for (@{$_[3]}) {
                    my($symbol,$lineno)=@$_;

                        defined($$term{$symbol}[0])
                    and do {
                        _SyntaxError(1,
                            "Precedence for symbol $symbol redefined: ".
                            "Previously defined line $$syms{$symbol}",
                            $lineno);
```

```
                        next;
                    };
                    $$token{$symbol}=$lineno;
                    $$term{$symbol} = [ $_[1][0], $prec ];
                }
                ++$prec;
                 undef
            }
    |   START ident '\n'                { $start=$_[2][0]; undef }
    |   HEADCODE '\n'                   { push(@$head,$_[1]); undef }
    |   UNION CODE '\n'                 { undef } #ignore
    |   TYPE typedecl identlist '\n'
            {
                for ( @{$_[3]} ) {
                    my($symbol,$lineno)=@$_;

                        exists($$nterm{$symbol})
                    and do {
                        _SyntaxError(0,
                                "Non-terminal $symbol redefined: ".
                                "Previously defined line $$syms{$symbol}",
                                $lineno);
                        next;
                    };
                    delete($$term{$symbol});   #not a terminal
                    $$nterm{$symbol}=undef;    #is a non-terminal
                }
            }
    |   EXPECT NUMBER '\n'             { $expect=$_[2][0]; undef }
    |   error '\n'                     { $_[0]->YYErrok }
;

typedecl:   #empty
    |       '<' IDENT '>'
;

symlist:    symlist symbol  { push(@{$_[1]},$_[2]); $_[1] }
    |       symbol          { [ $_[1] ] }
;

identlist:  identlist ident { push(@{$_[1]},$_[2]); $_[1] }
    |       ident           { [ $_[1] ] }
;

# Rule section
body:   rulesec '%%'
            {
                    $start
                or  $start=$$rules[1][0];

                    ref($$nterm{$start})
                or  _SyntaxError(2,"Start symbol $start not found ".
                                   "in rules section",$_[2][1]);
```

```

                $$rules[0]=[ '$start', [ $start, chr(0) ], undef, undef ];
            }
    |   '%%'    { _SyntaxError(2,"No rules in input grammar",$_[1][1]); }
;

rulesec:  rulesec rules #default action
    |     rules         #default action
;

rules:  IDENT ':' rhss ';'  { _AddRules($_[1],$_[3]); undef }
  | error ';'           { $_[0]->YYErrok }
;

rhss: rhss '|' rule   { push(@{$_[1]},$_[3]); $_[1] }
  | rule            { [ $_[1] ] }
;

rule:   rhs prec epscode    { push(@{$_[1]}, $_[2], $_[3]); $_[1] }
    |   rhs                 {
                                my($code)=undef;

                                    defined($_[1])
                                and $_[1][-1][0] eq 'CODE'
                                and $code = ${pop(@{$_[1]})}[1];

                                push(@{$_[1]}, undef, $code);

                                $_[1]
                            }
;

rhs:    #empty      #default action (will return undef)
    |   rhselts     #default action
;

rhselts:    rhselts rhselt  { push(@{$_[1]},$_[2]); $_[1] }
    |   rhselt      { [ $_[1] ] }
;

rhselt:     symbol    { [ 'SYMB', $_[1] ] }
    | code    { [ 'CODE', $_[1] ] }
    ;

prec: PREC symbol
        {
                        defined($$term{$_[2][0]})
                    or  do {
                        _SyntaxError(1,"No precedence for symbol $_[2][0]",
                                        $_[2][1]);
                        return undef;
                    };

```

```
                    ++$$precterm{$_[2][0]};
                    $$term{$_[2][0]}[1];
        }
;

epscode:        { undef }
    | code    { $_[1] }
;

code:   CODE      { $_[1] }
;

# Tail section:

tail:       /*empty*/
        |   TAILCODE    { $tail=$_[1] }
;

%%
sub _Error {
    my($value)=$_[0]->YYCurval;

    my($what)= $token ? "input: '$$value[0]'" : "end of input";

    _SyntaxError(1,"Unexpected $what",$$value[1]);
}

sub _Lexer {

    #At EOF
        pos($$input) >= length($$input)
    and return('',[ undef, -1 ]);

    #In TAIL section
        $lexlevel > 1
    and do {
        my($pos)=pos($$input);

        $lineno[0]=$lineno[1];
        $lineno[1]=-1;
        pos($$input)=length($$input);
        return('TAILCODE',[ substr($$input,$pos), $lineno[0] ]);
    };

    #Skip blanks
            $lexlevel == 0
        ?   $$input=~m{\G((?:
                                [\t\ ]+    # Any white space char but \n
                            |   \#[^\n]*  # Perl like comments
                            |   /\*.*?\*/ # C like comments
                            )+)}xsgc
        :   $$input=~m{\G((?:
                                \s+       # any white space char
```

```
                            |   \#[^\n]*  # Perl like comments
                            |   /\*.*?\*/ # C like comments
                            )+)}xsgc
    and do {
        my($blanks)=$1;

        #Maybe At EOF
            pos($$input) >= length($$input)
        and return('',[ undef, -1 ]);

        $lineno[1]+= $blanks=~tr/\n//;
    };

    $lineno[0]=$lineno[1];

        $$input=~/\G([A-Za-z_][A-Za-z0-9_]*)/gc
    and return('IDENT',[ $1, $lineno[0] ]);

        $$input=~/\G('(?:[^'\\]|\\\\|\\'|\\)+?')/gc
    and do {
            $1 eq "'error'"
        and do {
            _SyntaxError(0,"Literal 'error' ".
                           "will be treated as error token",$lineno[0]);
            return('IDENT',[ 'error', $lineno[0] ]);
        };
        return('LITERAL',[ $1, $lineno[0] ]);
    };

        $$input=~/\G(%%)/gc
    and do {
        ++$lexlevel;
        return($1, [ $1, $lineno[0] ]);
    };

        $$input=~/\G{/gc
    and do {
        my($level,$from,$code);

        $from=pos($$input);

        $level=1;
        while($$input=~/([{}])/gc) {
                substr($$input,pos($$input)-1,1) eq '\\' #Quoted
            and next;
                $level += ($1 eq '{' ? 1 : -1)
            or last;
        }
            $level
        and  _SyntaxError(2,"Unmatched { opened line $lineno[0]",-1);
        $code = substr($$input,$from,pos($$input)-$from-1);
        $lineno[1]+= $code=~tr/\n//;
        return('CODE',[ $code, $lineno[0] ]);
```

```
    };

    if($lexlevel == 0) {# In head section
            $$input=~/\G%(left|right|nonassoc)/gc
        and return('ASSOC',[ uc($1), $lineno[0] ]);
            $$input=~/\G%(start)/gc
        and return('START',[ undef, $lineno[0] ]);
            $$input=~/\G%(expect)/gc
        and return('EXPECT',[ undef, $lineno[0] ]);
            $$input=~/\G%{/gc
        and do {
            my($code);

                $$input=~/\G(.*?)%}/sgc
            or  _SyntaxError(2,"Unmatched %{ opened line $lineno[0]",-1);

            $code=$1;
            $lineno[1]+= $code=~tr/\n//;
            return('HEADCODE',[ $code, $lineno[0] ]);
        };
            $$input=~/\G%(token)/gc
        and return('TOKEN',[ undef, $lineno[0] ]);
            $$input=~/\G%(type)/gc
        and return('TYPE',[ undef, $lineno[0] ]);
            $$input=~/\G%(union)/gc
        and return('UNION',[ undef, $lineno[0] ]);
            $$input=~/\G([0-9]+)/gc
        and return('NUMBER',[ $1, $lineno[0] ]);

    }
    else {# In rule section
            $$input=~/\G%(prec)/gc
        and return('PREC',[ undef, $lineno[0] ]);
    }

    #Always return something
        $$input=~/\G(.)/sg
    or  die "Parse::Yapp::Grammar::Parse: Match (.) failed: report as a BUG";

        $1 eq "\n"
    and ++$lineno[1];

    ( $1 ,[ $1, $lineno[0] ]);

}

sub _SyntaxError {
    my($level,$message,$lineno)=@_;

    $message= "*".
              [ 'Warning', 'Error', 'Fatal' ]->[$level].
              "* $message, at ".
              ($lineno < 0 ? "eof" : "line $lineno").
```

```
              ".\n";

        $level > 1
    and die $message;

    warn $message;

        $level > 0
    and ++$nberr;

        $nberr == 20
    and die "*Fatal* Too many errors detected.\n"
}

sub _AddRules {
    my($lhs,$lineno)=@{$_[0]};
    my($rhss)=$_[1];

        ref($$nterm{$lhs})
    and do {
        _SyntaxError(1,"Non-terminal $lhs redefined: ".
                       "Previously declared line $$syms{$lhs}",$lineno);
        return;
    };

        ref($$term{$lhs})
    and do {
        my($where) = exists($$token{$lhs}) ? $$token{$lhs} : $$syms{$lhs};
        _SyntaxError(1,"Non-terminal $lhs previously ".
                       "declared as token line $where",$lineno);
        return;
    };

        ref($$nterm{$lhs})      #declared through %type
    or  do {
            $$syms{$lhs}=$lineno;   #Say it's declared here
            delete($$term{$lhs});   #No more a terminal
    };
    $$nterm{$lhs}=[];       #It's a non-terminal now

    my($epsrules)=0;        #To issue a warning if more than one epsilon rule

    for my $rhs (@$rhss) {
        my($tmprule)=[ $lhs, [ ], splice(@$rhs,-2) ]; #Init rule

            @$rhs
        or  do {
            ++$$nullable{$lhs};
            ++$epsrules;
        };

        for (0..$#$rhs) {
            my($what,$value)=@{$$rhs[$_]};
```

```
                $what eq 'CODE'
            and do {
                my($name)='@'.++$labelno."-$_";
                push(@$rules,[ $name, [], undef, $value ]);
                push(@{$$tmprule[1]},$name);
                next;
            };
            push(@{$$tmprule[1]},$$value[0]);
        }
        push(@$rules,$tmprule);
        push(@{$$nterm{$lhs}},$#$rules);
    }

        $epsrules > 1
    and _SyntaxError(0,"More than one empty rule for symbol $lhs",$lineno);
}

sub Parse {
    my($self)=shift;

        @_ > 0
    or  croak("No input grammar\n");

    my($parsed)={};

    $input=\$_[0];

    $lexlevel=0;
    @lineno=(1,1);
    $nberr=0;
    $prec=0;
    $labelno=0;

    $head=();
    $tail="";

    $syms={};
    $token={};
    $term={};
    $nterm={};
    $rules=[ undef ];   #reserve slot 0 for start rule
    $precterm={};

    $start="";
    $nullable={};
    $expect=0;

    pos($$input)=0;


    $self->YYParse(yylex => \&_Lexer, yyerror => \&_Error);

```

```
        $nberr
    and _SyntaxError(2,"Errors detected: No output",-1);

    @$parsed{ 'HEAD', 'TAIL', 'RULES', 'NTERM', 'TERM',
              'NULL', 'PREC', 'SYMS',  'START', 'EXPECT' }
    =       (  $head,  $tail,  $rules,  $nterm,  $term,
               $nullable, $precterm, $syms, $start, $expect);

    undef($input);
    undef($lexlevel);
    undef(@lineno);
    undef($nberr);
    undef($prec);
    undef($labelno);

    undef($head);
    undef($tail);

    undef($syms);
    undef($token);
    undef($term);
    undef($nterm);
    undef($rules);
    undef($precterm);

    undef($start);
    undef($nullable);
    undef($expect);

    $parsed
}
```

Índice alfabético

Bibliografía

[1] Schwartz L. and Phoenix T. *Learning Perl*. O'Reilly, USA, 2001. ISBN 0-596-00132-0.

[2] Srinivasan Sriram. *Advanced Perl Programming*. O'Reilly, USA, 1997. ISBN 1-56592-220-4.

[3] Wall L., Christiansen T., and Schwartz L. *Programming Perl*. O'Reilly, USA, 1996.

[4] Jefrrey E.F. Friedl. *Mastering Regular Expressions*. O'Reilly, USA, 1997. ISBN 1-56592-257-3.

[5] Lincoln D. Stein. *Network Programming with Perl*. Addison Wesley, USA, 2001. ISBN 0-201-61571-1.

[6] Christiansen T. and Schwartz L. *Perl Cookbook*. O'Reilly, USA, 1998. ISBN 1-56592-243-3.

[7] Joseph Hall and Randall L. Schwartz. *Effective Perl Programming. Writing Better Programs with Perl*. Addison Wesley, USA, 2001. ISBN 0-201-41975-0.

[8] Sean M. Burke. *Perl and LWP*. O'Reilly, 2002.

[9] A. Descartes and T. Bunce. *Programming the Perl DBI*. O'Reilly, 2000.

[10] Brian Ingerson. *Pathollogically Polluting perl with C, Python and Other Rubbish using Inline.pm*. http://nereida.deioc.ull.es/ lhp/pdfps/ingy.inline.pdf, 2001.

[11] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

[12] Peter Scott. *Perl Medic: Maintaining Inherited Code*. Addison Wesley, USA, 2004. ISBN 0201795264.

[13] S. Martello and P. Toth. *Knapsack Problems*. John Wiley and Sons, Chichester, 1990.

[14] Conway D. *Object Oriented Perl*. Manning, Greenwich, USA, 2000.

[15] Paul Bausch. *Amazon Hacks*. O'Reilly, 2003.