Principios de Programación Imperativa, Funcional y Orientada a Objetos *Una Introducción en Perl/Una Introducción a Perl*

Casiano R. León [1]

4 de octubre de 2009

---

[1] DEIOC Universidad de La Laguna

Índice general

5. Módulos 198

Índice de figuras

Índice de cuadros

A Juana

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

Agradecimientos/Acknowledgments

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

Special thanks to Larry Wall for giving us Perl.

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

Erratas

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

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/~lhp/perlexamples/perlexamples.ps`

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

`http://nereida.deioc.ull.es/~lhp/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 en cantidad y en muchos casos en calidad. 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]. Un libro extraordinario sobre el depurador de Perl es el de Foley y Lester [4]. Para saber en profundidad sobre expresiones regulares esta el libro de Friedl [5]. El libro de Stein [6] cubre temas de programación en redes. Miles de recetas clasificadas según los problemas pueden encontrarse en el libro de Christiansen [7]. El texto de Jennes y Cozens [8] es de gran ayuda cuando se considera la posibiliad de combinar C y Perl.

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 [9].

Los capítulos de análisis léxico y sintáctico están inspirados en el clásico libro del dragón [10].

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.

Capítulo 1

Introducción

1.1. Primeros Pasos

Donde Encontrar Perl y Como Instalarlo

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. Lea el tutorial en http://www.perl.com/download.csp para mas detalles.

Algunos enlaces sobre Perl en windows:

1. Discusión en PerlMonks sobre el tema: *Preferred Windows Perl?*
2. Strawberry Perl: http://win32.perl.org
3. Active State Perl: http://www.activestate.com/store/
4. Cygwin: http://www.cygwin.com/

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
```

Donde esta Perl en Nuestra Máquina

Comprueba en que directorio se guarda el ejecutable:

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

El comando `whereis` puede ser usado para encontrar los binarios, fuentes y páginas del manual de Perl:

```
lhp@nereida:~$ whereis perl
perl: /usr/bin/perl /etc/perl /usr/lib/perl /usr/local/lib/perl
      /usr/share/perl /usr/share/man/man1/perl.1.gz
```

Ejecución de un Programa 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!
```

Ejecución con el Depurador

También podemos usar el depurador para ejecutar el programa paso a paso. El depurador de Perl se activa usando la opción `-d` del intérprete:

```
$ perl -d hola.pl

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::(hola.pl:2):      print "hola!\n";
  DB<1> h
List/search source lines:               Control script execution:
  l [ln|sub]  List source code            T           Stack trace
  - or .      List previous/current line  s [expr]    Single step [in expr]
  v [line]    View around line            n [expr]    Next, steps over subs
  f filename  View source in file         <CR/Enter>  Repeat last n or s
  /pattern/ ?patt?   Search forw/backw    r           Return from subroutine
  M           Show module versions        c [ln|sub]  Continue until position
Debugger controls:                        L           List break/watch/actions
  o [...]     Set debugger options        t [expr]    Toggle trace [trace expr]
  <[<]|{[{]|>[>] [cmd] Do pre/post-prompt b [ln|event|sub] [cnd] Set breakpoint
  ! [N|pat]   Redo a previous command     B ln|*      Delete a/all breakpoints
  H [-num]    Display last num commands   a [ln] cmd  Do cmd before line
  = [a val]   Define/list an alias        A ln|*      Delete a/all actions
  h [db_cmd]  Get help on command         w expr      Add a watch expression
  h h         Complete help page          W expr|*    Delete a/all watch exprs
  |[|]db_cmd  Send output to pager        ![!] syscmd Run cmd in a subprocess
  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.

Consulte el libro de Foley y Lester [4] para saber mas sobre el depurador. También podemos consultar http://debugger.perl.org/. En http://refcards.com/docs/forda/perl-debugger/perl-debugger-refcard-a4.pdf se puede encontrar una página de referencia (PDF).

Instrucciones Para la Carga de Módulos en la ETSII

Cuando entra a una de las máquinas de los laboratorios de la ETSII encontrará en `/soft/perl5lib/` algunas distribuciones que he instalado.

Establezca las siguientes variables de entorno para poder acceder a las mismas:

```
[casiano@tonga]~/src/perl/tests/gtk2/> cat ~/bin/setperl5lib
# October 2008
# source ~/bin/setperl5lib
export PERL5LIB=/soft/perl5lib/share/perl/5.8.8/:/soft/perl5lib/lib/perl/5.8.8:/soft/perl5lib/lib/perl/5.8
export MANPATH=$MANPATH:/soft/perl5lib/man/
export PATH=$PATH:/soft/perl5lib/bin
```

Ejecución en la Línea de Comandos

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"'
16
```

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

En la jerga Perl a los programas ejecutados con la opción `-e` se les llama *one-liners*.

Carga de Módulos en One-liners

La opción `-M` seguida del nombre de un módulo (sin espacios) permite cargar un módulo para que las funciones proveídas por el mismo estén disponibles en el one-liner. Por ejemplo, el siguiente programa:

```
$ perl -MLWP::Simple -e "mirror('http://www.gutenberg.org/dirs/etext99/advsh12h.htm', 'ADVENS_OF_SHERLOCK.
```

usa la función `mirror` de LWP::Simple para crear un fichero `ADVENS_OF_SHERLOCK.html` a partir de la correspondiente página conteniendo la novela THE ADVENTURES OF SHERLOCK HOLMES en el Proyecto Gutenberg.

Uso de las Flechas en el Depurador para Navegar en la Historia

Cuando se usa el depurador, es muy cómodo que funcione el editor que permite usar las flechas para navegar en la historia de comandos. Si no le funcion pruebe a instalar el bundle `Bundle::CPAN`

usando el comando:

```
cpan Bundle::CPAN
```

o alternativamente este otro comando:

```
perl -MCPAN -e 'install Bundle::CPAN'
```

que carga el módulo CPAN y llama a la función `install` con argumento `Bundle::CPAN`.

Alternativamente, si se dispone de una distribución de Linux Ubuntu se puede instalar uno de los paquetes `libterm-readkey-perl` o `libterm-readline-gnu-perl`.

Uso del Depurador para Estudiar Perl

Una forma habitual de usar el depurador es utilizarlo para estudiar la sintáxis y semántica de algún constructo Perl. Para ello se combinan las opciones `-d` y `-e` en la llamada al intérprete. La opción `-e` se hace seguir de un programa trivial (p.ej. 0). A continuación se pasa a estudiar la conducta de las expresiones en las que estemos interesados. Veamos un ejemplo:

```
lhp@nereida:~/Lperl/doc$ perl -d -e 0
Loading DB routines from perl5db.pl version 1.28
Editor support available.
Enter h or `h h' for help, or `man perldebug' for more help.
main::(-e:1):   0
  DB<1> p sqrt(16)
  4
  DB<2> p "hola"x4
  holaholaholahola
  DB<3> q
```

Ejercicio 1.1.1. *Entre en el depurador y pruebe los comandos*

- `h h`
- `|h h`
- `perldoc print`

Ejercicio 1.1.2. *Entre en el depurador y pida ayuda sobre los comandos*

- `p`
- `x`

- `.`
- `l`
- `n`
- `s`

La Documentación en Perl

Para obtener información sobre un tópico o un módulo Perl utilize `perldoc`. Si tiene instalado el módulo `Tk::Pod` entonces puede utilizar el visualizador gráfico `tkpod`. Otra posibilidad es arrancar el programa `podwebserver` disponible en la distribución del módulo Pod::Webserver. Este programa arranca un servidor HTTP que sirve la documentación de los módulos instalados en la máquina:

```
$ podwebserver &
[2] 4069
$ I am process 4069 = perl Pod::Webserver v3.05
 Indexing all of @INC -- this might take a minute.
@INC = [ /soft/perl5lib/share/perl/5.8.8/ /soft/perl5lib/lib/perl/5.8.8 /soft/perl5lib/lib/perl/5.8 /soft/
 Done scanning @INC
You can now open your browser to http://localhost:8020/
```

La documentación puede ahora ser accedida usando nuestro navegador favorito:

```
$ firefox  http://localhost:8020/ &
```

La fuente de información sobre el lenguaje Perl mas fiable es la que se encuentra en la documentación `.pod` adjunta con su distribución. Veamos algunos de los parámetros que admite perldoc:

```
lhp@nereida:~$ perldoc -h
perldoc [options] PageName|ModuleName|ProgramName...
perldoc [options] -f BuiltinFunction
perldoc [options] -q FAQRegex

Options:
    -h   Display this help message
    -V   report version
    -r   Recursive search (slow)
    -i   Ignore case
    -t   Display pod using pod2text instead of pod2man and nroff
             (-t is the default on win32 unless -n is specified)
    -u   Display unformatted pod text
    -m   Display module's file in its entirety
    -n   Specify replacement for nroff
    -l   Display the module's file name
    -F   Arguments are file names, not modules
    -v   Verbosely describe what's going on
    -T   Send output to STDOUT without any pager
    -d output_filename_to_send_to
    -o output_format_name
    -M FormatterModuleNameToUse
    -w formatter_option:option_value
    -X   use index if present (looks for pod.idx at /usr/lib/perl/5.8)
    -q   Search the text of questions (not answers) in perlfaq[1-9]

PageName|ModuleName...
```

```
         is the name of a piece of documentation that you want to look at. You
         may either give a descriptive name of the page (as in the case of
         `perlfunc') the name of a module, either like `Term::Info' or like
         `Term/Info', or the name of a program, like `perldoc'.

BuiltinFunction
         is the name of a perl function.  Will extract documentation from
         `perlfunc'.

FAQRegex
         is a regex. Will search perlfaq[1-9] for and extract any
         questions that match.

Any switches in the PERLDOC environment variable will be used before the
command line arguments.  The optional pod index file contains a list of
filenames, one per line.
                                                        [Perldoc v3.14]
```

Hay una serie de documentos que pueden consultarse con `perldoc` o tkpod. Los mas básicos son:

- perlintro Perl introduction for beginners
- perltoc Perl documentation table of contents

Hay un conjunto extenso de tutoriales:

- Tutorials
    - perlreftut Perl references short introduction
    - perldsc Perl data structures intro
    - perllol Perl data structures: arrays of arrays
    - perlrequick Perl regular expressions quick start
    - perlretut Perl regular expressions tutorial
    - perlboot Perl OO tutorial for beginners
    - perltoot Perl OO tutorial, part 1
    - perltooc Perl OO tutorial, part 2
    - perlbot Perl OO tricks and examples
    - perlstyle Perl style guide
    - perlcheat Perl cheat sheet
    - perltrap Perl traps for the unwary
    - perldebtut Perl debugging tutorial
- perlfaq
    - perlfaq Perl frequently asked questions
    - perlfaq1 General Questions About Perl
    - perlfaq2 Obtaining and Learning about Perl
    - perlfaq3 Programming Tools
    - perlfaq4 Data Manipulation
    - perlfaq5 Files and Formats

- perlfaq6 Regexes
- perlfaq7 Perl Language Issues
- perlfaq8 System Interaction
- perlfaq9 Networking

Los manuales de referencia se dividen por tópicos:

- perlsyn Perl syntax
- perldata Perl data structures
- perlop Perl operators and precedence
- perlsub Perl subroutines
- perlfunc Perl built-in functions
- perlopentut Perl open() tutorial
- perlpacktut Perl pack() and unpack() tutorial
- perlpod Perl plain old documentation
- perlpodspec Perl plain old documentation format specification
- perlrun Perl execution and options
- perldiag Perl diagnostic messages
- perllexwarn Perl warnings and their control
- perldebug Perl debugging
- perlvar Perl predefined variables
- perlre Perl regular expressions, the rest of the story
- perlreref Perl regular expressions quick reference
- perlref Perl references, the rest of the story
- perlform Perl formats
- perlobj Perl objects
- perltie Perl objects hidden behind simple variables
- perldbmfilter Perl DBM filters
- perlipc Perl interprocess communication
- perlfork Perl fork() information
- perlnumber Perl number semantics
- perlthrtut Perl threads tutorial
- perlothrtut Old Perl threads tutorial
- perlport Perl portability guide
- perllocale Perl locale support

- perluniintro Perl Unicode introduction
- perlunicode Perl Unicode support
- perlebcdic Considerations for running Perl on EBCDIC platforms
- perlsec Perl security
- perlmod Perl modules: how they work
- perlmodlib Perl modules: how to write and use
- perlmodstyle Perl modules: how to write modules with style
- perlmodinstall Perl modules: how to install from CPAN
- perlnewmod Perl modules: preparing a new module for distribution
- perlutil utilities packaged with the Perl distribution
- perlcompile Perl compiler suite intro
- perlfilter Perl source filters

Por último - solo para gurus - estan los documentos que describen las partes internas de Perl:

- perlembed Perl ways to embed perl in your C or C++ application
- perldebguts Perl debugging guts and tips
- perlxstut Perl XS tutorial
- perlxs Perl XS application programming interface
- perlclib Internal replacements for standard C library functions
- perlguts Perl internal functions for those doing extensions
- perlcall Perl calling conventions from C
- perlapi Perl API listing (autogenerated)
- perlintern Perl internal functions (autogenerated)
- perliol C API for Perl's implementation of IO in Layers
- perlapio Perl internal IO abstraction interface
- perlhack Perl hackers guide

La opción `-l` de `perldoc` permite ver en que directorio se encuentra la documentación:

```
$ perldoc -l perlintro
/usr/share/perl/5.8/pod/perlintro.pod
```

Si se desea convertir desde `pod` a algún formato específico, por ejemplo a LaTeX basta con usar el conversor adecuado:

```
$ pod2latex -full ‘perldoc -l perlintro‘
$ ls -l perlintro.tex
-rw-r--r-- 1 pp2 pp2 24333 2008-02-21 13:22 perlintro.tex
$ pdflatex perlintro.tex
This is pdfeTeX, Version 3.141592-1.30.5-2.2 (Web2C 7.5.5)
entering extended mode
............................................................
$ ls -ltr *.pdf | tail -1
-rw-r--r-- 1 pp2 pp2 191367 2008-02-21 13:22 perlintro.pdf
```

En este caso hemos usado `pod2latex`. Para poder ejecutar la conversión de `.pod` a LaTeX asegúrese de tener instalado el módulo `Pod::LaTeX` (puede obtenerlo desde CPAN).

Generando Formatos con `perldoc`

La utilidad `perldoc` dispone de opciones para llamar a un conversor y producir automáticamente el formato deseado (si los correspondientes módulos están instalados):

- `-o output-formatname`

  especifica el formato de salida. `-oman`. Usar `-oformatname` hace que se carge un módulo para ese formato.

- `-M module-name`

  Especifica el módulo que queremos usar para formatear el `pod`. La clase debe proporcionar un método `parse_from_file`. Por ejemplo `perldoc -MPod::Perldoc::ToChecker`.

  Se pueden especificar varias clases separándolas por comas o punto y coma: `-MTk::SuperPod;Tk::Pod`.

- `-w option:value` o `-w option`

  Especifica opciones en la llamada al formateador utilizado. Por ejemplo `-w textsize:15` llamará al objeto formateador `$formatter->textsize(15)` antes de que se use para formatear. La forma `-w optionname` es una abreviación para `-w optionname:TRUE`.

Véase un ejemplo que produce formato LaTeX.

```
$ perldoc -oLaTeX -wAddPreamble:1 -f require > /tmp/require.tex
```

La opción `AddPreamble` indica al objeto formateador que se desea generar un documento completo y no una sección de un documento completo. Después de generar el fichero LaTeX podemos compilarlo con alguno de los compiladores de `latex` para producir un `dvi` o un `pdf`. Como LaTeXno está instalado en las máquinas de las salas copiamos el fichero a uno de los servidores y compilamos con `pdflatex`:

```
casiano@tonga:~$ scp millo:/tmp/require.tex .
require.tex                          100% 6534     6.4KB/s   00:00
casiano@tonga:~$ pdflatex require.tex
This is pdfeTeX, Version 3.141592-1.21a-2.2 (Web2C 7.5.4)
entering extended mode
....
Output written on require.pdf (4 pages, 62447 bytes).
Transcript written on require.log.
```

Una vez generado el `.pdf` podemos visualizarlo con `kpdf` e imprimirlo:

```
casiano@tonga:/tmp$
casiano@tonga:/tmp$ kpdf require.pdf &
```

Puede que encuentre útil usar la hoja de referencia rápida sobre Perl que se encuentra en http://johnbokma.com/perl/perl-quic

1.2. 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
```

Los comandos `x` y `p` muestran los contenidos de una variable. Perl es un lenguaje en el que el significado de las frases depende del contexto. El comando `x` evalúa la expresión en un contexto de lista. Es por eso que aparece el 0 junto al 5 en la respuesta. El resultado es una lista cuyo primer elemento, el elemento 0, es 5.

1.2.1. 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:

```
lhp@nereida:~/Lperl/src$ perl -wde 0
Loading DB routines from perl5db.pl version 1.28
Editor support available.

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

main::(-e:1):   0
  DB<1> p 0377
255
  DB<2> p 0xff
255
  DB<3> p 0b1111_1111
255
```

Si lo que queremos es hacer la conversión inversa podemos usar `sprintf`:

```
lhp@nereida:~/Lbook$ perl -wde 0
main::(-e:1):   0
  DB<1> $num = 42;
  DB<2>   printf("%o", $num);
52
  DB<3> $oct = sprintf("%o", $num);
  DB<4> p $oct
52
```

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

Si se quiere precisión infinita existe un buen número de librerías disponibles. Para cálculo con enteros se puede, ejemplo, usar Math::Pari:

```
pp2@nereida:~/src/testing$ perl -wde 0
main::(-e:1):   0
  DB<1> use Math::Pari
  DB<2> $a = PARI 2
  DB<3> print $a**10000
19950631168807583848837421626835850838234968318861924548520089498529438830221946
63191996168403619459789933112942320912427155649134941378111759378593209632395785
57300467937945267652465512660598955205500869181933115425086084606181046855090748
66089624888090489894838009253941633257850621568309473902556912388065225096643874
44104675987162698545322286853816169431577562964076283688076073222853509164147618
39563814589694638994108409605362678210646214273333940365255656495306031426802349
69400335934316651459297773279665775606172582031407994198179607378245683762280037
30288548725190083446458145465055792960141483392161573458813925709537976911927780
08269577356744441230620187578363255027283237892707103738028663930314281332414016
24195671690574061419654342324638801248856147305207431992259611796250130992860241
70834080760593232016126849228849625584131284406153673895148711425631511108974551
42033138202029316409575964647560104058458415660720449628670165150619206310041864
22275908670900574606417856951911456055068251250406007519842261898059237118054444
78807290639524254833922198270740447316237676084661303377870603980341319713349365
46227005631699374555082417809728109832913144035718775247685098572769379264332215
99399876886660808368837838027643282775172273657572744784112294389733810861607423
25329197481312019760417828196569747589816453125843413595986278413012818540628347
66490886905210475808826158239619857701224070443305830758690393196046034049731565
83208672105913300903752823415539745394397715257455290510212310947321610753474825
74077527398634829849834075693795564663862187456949927901657210370136443313581721
```

```
43117913982229838458473344402709641828510050729277483645505786345011008529878123
89473928699540834346158807043959118985815145779177143619698728131459483783202081
47498217185801138907122825090582681743622057747592141765371568772561490458290499
24610286300815355833081301019876758562343435389554091756234008448875261626435686
48833519463720377293240094456246923254350400678027273837755376406726898636241037
49141096671855705075909810024678988017827192595338128242195402830275940844895501
46766683896979968862416363133763939033734558014076367418777110553842257394991101
86468219696581651485130494222369947714763069155468217682876200362777257723781365
33161119681128079266948188720129864366076855163986053460229787155751794738524636
94469230878942659482170080511203223654962881690357391213683383935917564187338505
10970271613915439590991598154654417336311656936031122249937969999226781732358023
11186264457529913575817500819983923628461524988108896023224436217377161808635701
54684840586223297928538756234865564405369626220189635710288123615675125433383032
70029097668650568557157505516727518899194129711337690149916181315171544007728650
57318955745092033018530484711381831540732405331903846208403642176370391155063978
90007428536721962809034779745333204683687958685802379522186291200807428195513179
48157624448298518461509704888027274721574688131594750409732115080498190455803416
826949787141316063210686391511681774304792596709376
```

1.2.2. Cadenas

Las cadenas son secuencias de caracteres y pueden contener cualquier 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"
```

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 1.2.2. *¿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";
print 'hola\\'';
```

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

Cuadro 1.1: Secuencias de Escape en Cadenas de Doble Comilla

Interpolación en Cadenas de Comillas Dobles

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. Vea el siguiente ejemplo:

```
pp2@nereida:~/doc/2005_2006$ perl -wde 0
main::(-e:1):   0
  DB<1> $que = "chuleta"; $n = 3
  DB<2> print "Antonio comió $n $ques.\n"
Use of uninitialized value in concatenation (.) or string at (eval 6)\
   [/usr/share/perl/5.8/perl5db.pl:628] line 2.
 at (eval 6)[/usr/share/perl/5.8/perl5db.pl:628] line 2
        eval '($@, $!, $^E, $,, $/, $\\, $^W) = @saved;package main; $^D = $^D | $DB::db_stop;
print "Antonio comió $n $ques.\\n";

;' called at /usr/share/perl/5.8/perl5db.pl line 628
        DB::eval called at /usr/share/perl/5.8/perl5db.pl line 3410
```

```
        DB::DB called at -e line 1
Antonio comió 3 .
```

El resultado es un mensaje de advertencia del depurador: La variable `$ques` no esta incializada. La salida es:

`Antonio comió 3 .`

Una variable no inicializada contiene el valor especial `undef`. Cuando el valor `undef` se evalúa en un contexto de cadena se evalúa como una cadena vacía.

Mediante el uso de llaves logramos la respuesta correcta:

```
  DB<3> print "Antonio comió $n ${que}s.\n"
Antonio comió 3 chuletas.
```

Otra solución es usar el operador de concatenación (denotado mediante un punto `"."`):

```
  DB<4> print "Antonio comió $n $que"."s.\n"
Antonio comió 3 chuletas.
```

Convirtiendo en `Uppercase` y `Lowercase`

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`.

```
lhp@nereida:~/Lperl/doc$ perl -de 0
main::(-e:1):   0
DB<1> p "Hola\cL\cMJuan\cL\cM"
Hola
Juan

DB<2> p "esta es una \Ugran\E palabra"
esta es una GRAN palabra
```

Los operadores `q` y `qq`

En Perl las diversas formas de entrecomillados son operadores (de la misma manera que la concatenación, la suma o el producto).

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!
```

Para sabe mas sobre este tipo de operadores consulte la sección *Quote and Quote-like Operators* de `perlop` (escriba `perldoc perlop` en la consola).

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 conteniendo 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";
```

Si se quiere obtener uno o mas elementos de una cadena se usa `substr`

```
  DB<1> $x = 'hola mundo'
  DB<2> p substr($x,5)
mundo
  DB<3> p substr($x,5,3)
mun
  DB<4> p substr($x,-2) # Segunda desde el final
do
  DB<5> p substr($x,2,2,'mi')
la
  DB<6> x $x
0  'homi mundo'
```

La función `index` retorna la posición de una subcadena dentro de una cadena. La función `rindex` es similar pero busca desde el final:

```
  DB<1> $x = 'hola mundo'
  DB<2> p index $x, 'mundo'
5
  DB<3> p index $x, 'o'
1
  DB<4> p index $x, 'o', 3
9
  DB<5> p rindex $x, 'o'          # busca desde el final
9
  DB<6> p rindex $x, 'o', 4
1
```

La función `reverse` invierte los elementos de una cadena:

```
  DB<1> $x = 'hola mundo'
  DB<2> $y = reverse $x
  DB<3> p $y
odnum aloh
```

La función `length` retorna la longitud de una cadena.

```
  DB<1> $x = 'hola mundo'
  DB<2> p length $x
10
```

La función `chop` elimina el último elemento de una cadena. La función `chomp` sólo elimina el último elemento si es un retorno de carro (mas concretamente si es el separador de lectura):

```
  DB<3> chop $x
  DB<4> p $x
hola mund
  DB<5> chomp $x
  DB<6> p $x
hola mund
  DB<7> $x .= "\n"
  DB<8> p $x
hola mund

  DB<9> chomp $x
  DB<10> p $x
hola mund
```

1.2.3. 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 C usada 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 ni hexadecimales. Para convertirlos 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* (como `&`, `|`, etc.) 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:

```
lhp@nereida:~/Lperl/doc$ perl -de 0
Loading DB routines from perl5db.pl version 1.28
Editor support available.
Enter h or `h h' for help, or `man perldebug' for more help.
main::(-e:1):   0
  DB<1> $a = 0xF0F
```

```
  DB<2> $b = 0xFF0
  DB<3> p $a&$b
3840
  DB<4> p 0xF00
3840
  DB<5> $a = "AB"
  DB<6> $b = "AD"
  DB<7> p "B" & "D"
@
  DB<8> p $a & $b
A@
```

1.2.4. Variables Mágicas

La Variable `$!`

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()`:

```
lhp@nereida:~/etc$ perl -wde 0
main::(-e:1):   0
  DB<1> open $FILE, "fichero"
  DB<2> print "$!"
No existe el fichero o el directorio
  DB<3> print $!+0
2
  DB<4> print $!
No existe el fichero o el directorio
```

La función `open` intenta abrir para lectura el fichero con nombre `fichero` para dejar una descripción del mismo en la variable `$FILE`. En el supuesto de que tal fichero no exista, se producirán los correspondientes mensajes de error almacenados en `$!`.

Creando Variables Sensibles al Contexto

Es muy sencillo crear variables con una conducta sensible al contexto (numérico o de cadena) usando la función `dualvar` de `Scalar::Util`. Observe la siguiente sesión con el depurador:

```
pl@nereida:~/doc/casiano/PLBOOK/PLBOOK$ perl -wde 0
main::(-e:1):   0
  DB<1> use Scalar::Util qw{dualvar} # Importamos 'dualvar' del módulo Scalar::Util
  DB<2> use Lingua::EN::Words2Nums   # Importamos el módulo Lingua::EN::Words2Nums
  DB<3> $x = "one thousand two hundred thirty one" # ¿Que tal anda nuestro inglés?
  DB<4> $x = dualvar words2nums($x), $x # Creamos una variable dual
  DB<5> print "$x plus one is ".($x+1)  # Observe la salida ...
one thousand two hundred thirty one plus one is 1232
  DB<6> use Roman                     # Otro ejemplo ...
  DB<7> $y = 457
  DB<8> $y = dualvar $y, roman($y)
  DB<9> p 0+$y." en números romanos es '$y'"
457 en números romanos es 'cdlvii'
```

Mas potente aún es `Contextual::Return` de Damian Conway, el cual permite crear variables contextuales multitipo con mas de dos tipos:

```
lhp@nereida:~/Lperl/src/testing$ cat -n context1.pl
 1  #!/usr/local/bin/perl -w
 2  use strict;
 3  use Contextual::Return;
 4
 5  my $x = BOOL { 0 } NUM { 3.14 } STR { "pi" };
 6
 7  unless ($x) { warn "¡El famoso número $x (".(0+$x).") pasa a ser falso!\n" } # executed!
lhp@nereida:~/Lperl/src/testing$ context1.pl
¡El famoso número pi (3.14) pasa a ser falso!
```

Si se usa la etiqueta `ACTIVE` el código de definición será ejecutado cada vez que la variable multiestado es evaluada. Por ejemplo:

```
lhp@nereida:~/Lperl/src/testing$ cat -n context2.pl
1  #!/usr/local/bin/perl -w
2  use strict;
3  use Contextual::Return;
4
5  my $now = ACTIVE NUM { time } STR { localtime };
6
7  print "Now: $now (".($now+0).")\n";
8  sleep(1);
9  print "Now: $now (".($now+0).")\n";
```

La ejecución del programa produce una salida como esta:

```
lhp@nereida:~/Lperl/src/testing$ context2.pl
Now: Sat Mar 15 11:45:58 2008 (1205581558)
Now: Sat Mar 15 11:45:59 2008 (1205581559)
```

La Variable $_

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 $_;
```

Otro ejemplo: 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 por defecto* $_:

```
lhp@nereida:~/etc$ perl -wde 0
Loading DB routines from perl5db.pl version 1.28
main::(-e:1):   0
  DB<1> $_ = "Hello World!\n"
  DB<2> p length
13
```

Variables Predefinidas en Perl

Existe un gran número de variables predefinidas en Perl que gobiernan diferentes aspectos del comportamiento del intérprete. Las iremos introduciendo a lo largo del curso. Por ahora, para saber más sobre las variables mágicas lea la documentación en `perlvar`:

```
$ perldoc perlvar
```

Como se señala en dicha documentación el módulo `English` permite usar un seudónimo para cada variable mágica. Por ejemplo, el alias de `$_` es `$ARG`:

```
lhp@nereida:~/etc$ perl -de 0
DB<1> use English
DB<2> $_ = 4
DB<3> p $ARG
4
```

1.3. Variables privadas

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

```
1 my $a = 4;
2 {
3   my $a = <STDIN>;
4   my $b = <STDIN>;
5   $a = ($a < $b)? $a : $b
6   print "$a\n";
7 }
8 print "$a\n"; # 4
```

Estas variables tienen como ámbito el bloque que las rodea. Asi la declaración de la variable léxica `$a` en la línea 3 oculta la variable léxica declarada en la línea 1. Las modificaciones de `$a` en el bloque de las líneas 2-7 no afecta a la variable `$a` declarada en la línea 1.

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

1.3.1. 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.

Cuando se alcanza el final del fichero, la lectura devuelve el valor especial `undef`.

En general un fichero se abre mediante la función `open`:

```
lhp@nereida:~/Lperl/src$ perl -wde 0
main::(-e:1):   0
DB<1> open $F, "> /tmp/file.txt" # apertura en modo escritura
DB<2> print $F "hello world\n"
```

```
DB<3> close($F)
DB<4> open $F, "/tmp/file.txt" # apertura en modo lectura
DB<5> $x = <$F>
DB<6> print $x
hello world
```

1.4. 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
```

1.4.1. 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 1.4.1. *¿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"` *(véase la impresión en la línea 3) 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* `&&`*?*

Para saber en que forma esta parentizando Perl un programa puede usarse el módulo `Deparse` con el siguiente formato:

```
lhp@nereida:~/Lperl/src$ cat -n parenthesis.pl
     1  $a = 4;
     2  $b = "hola";
     3  $c = print $a and $b
lhp@nereida:~/Lperl/src$ perl -MO=Deparse,-p parenthesis.pl
($a = 4);
($b = 'hola');
(($c = print($a)) and $b);
parenthesis.pl syntax OK
```

Así se ve claro que `$c` contiene el valor retornado por la llamada a `print($a)`.

1.4.2. 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 1.2: 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
```

1.5. Algunas Sentencias de Control

Los condicionales y los bucles son como en C. La mayor diferencia es que requieren que la sentencia bajo control sea siempre un bloque, esto es, este escrita entre llaves. Puede encontrar una breve introducción a las sentencias de control en perlintro.

if

```
        if ( condition ) {
            ...
        } elsif ( other condition ) {
            ...
        } else {
            ...
        }
```

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";
}
```

Forma Sufija

Si no se quieren escribir las llaves se puede usar la notación sufija de la sentecia:

```
        # la forma tradicional
        if ($zippy) {
            print "Yow!";
        }

        # al estilo Perl
        print "Yow!" if $zippy;
```

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

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

while

```
        while ( condition ) {
            ...
        }
```

También se puede usar en sufijo;

```
        print "LA LA LA\n" while 1;          # loops forever
```

for y foreach

Se puede usar como en C:

```
        for ($i = 0; $i <= $max; $i++) {
            ...
        }
```

O en estilo Perl:

```
lhp@europa:~/projects/perl/src/perltesting$ cat ./for.pl
#!/usr/bin/perl -w
use warnings;
use strict;

foreach my $i (1..5) {
  print "$i ";
}

print "\n";

print "$_ " foreach (1..5);

print "\n";

for my $i (1..5) {
  print "$i ";
}

print "\n";
```

Al ejecutarlo produce:

```
lhp@europa:~/projects/perl/src/perltesting$ ./for.pl
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

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.

next

El comando `next` comienza la siguiente iteración del bucle:

```
LINE: while (<STDIN>) {
    next LINE if /^#/;      # discard comments
    ...
}
```

last

El comando `last` produce la salida del bucle:

```
LINE: while (<STDIN>) {
    last LINE if /^$/;      # exit when done with header
    ...
}
```

Switch

Use el módulo Switch (no es necesario en las versiones posteriores a la 5.10):

```
lhp@europa:~/projects/perl/src/perltesting$ cat switch.pl
#!/usr/bin/perl -w
use strict;
use Perl6::Say;
use Switch;

my $val = <>;
chomp($val);

switch ($val) {
  case 1         { say "number 1" }
  case "a"       { say "string a" }
  case [1..10,42] { say "number in list" }
  case /^\w+$/   { say "pattern" }
  else           { say "previous case not true" }
}
```

Veamos algunas ejecuciones:

```
lhp@europa:~/projects/perl/src/perltesting$ ./switch.pl
5
number in list
lhp@europa:~/projects/perl/src/perltesting$ ./switch.pl
12bc
pattern
lhp@europa:~/projects/perl/src/perltesting$ ./switch.pl
.2.
previous case not true
```

1.6. 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`
- Añada:

  ```
  use strict;
  ```

  al comienzo de su programa. 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 incluya el pragma `use strict`.

- Puede producir diagnósticos mas detallados poniendo:

  ```
  use diagnostics;
  ```

  al comienzo de su programa. Esto ya activa el modo `-w`. Vea un ejemplo. Dado el programa:

  ```
  lhp@europa:~/projects/perl/src/perltesting$ cat diag.pl
  use warnings;
  use strict;
  use Perl6::Say;

  my ($x, $y) = (1,0);
  say $x/$y;
  ```

  Su ejecución produce el mensaje de error:

  ```
  lhp@europa:~/projects/perl/src/perltesting$ perl diag.pl
  Illegal division by zero at diag.pl line 6.
  ```

  Mientras que la ejecución con diagnostics produce una descripción mas profusa:

  ```
  lhp@europa:~/projects/perl/src/perltesting$ perl -Mdiagnostics diag.pl
  Illegal division by zero at diag.pl line 6 (#1)
      (F) You tried to divide a number by 0.  Either something was wrong in
      your logic, or you need to put a conditional in to guard against
      meaningless input.

  Uncaught exception from user code:
          Illegal division by zero at diag.pl line 6.
   at diag.pl line 6
  ```

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

  Consulte el libro de Foley y Lester [4] para saber mas sobre el depurador. También podemos consultar http://debugger.perl.org/. En http://refcards.com/docs/forda/perl-debugger/perl-debugger-refcard-a4.pdf se puede encontrar una página de referencia (PDF).

- Puede instalar y usar `perlcritic`. Le informará sobre las malas prácticas que aprecia en su programa. Vea un ejemplo:

```
nereida:/tmp/Perl-Critic-1.080# cat hello.pl
print "Hello world!\n";

nereida:/tmp/Perl-Critic-1.080# perlcritic --brutal hello.pl
Return value of flagged function ignored - print at line 1, column 1.  See pages 208,278 of PBP.  (Sev
RCS keywords $Id$ not found at line 1, column 1.  See page 441 of PBP.  (Severity: 2)
RCS keywords $Revision$, $HeadURL$, $Date$ not found at line 1, column 1.  See page 441 of PBP.  (Seve
RCS keywords $Revision$, $Source$, $Date$ not found at line 1, column 1.  See page 441 of PBP.  (Sever
Module does not end with "1;" at line 1, column 1.  Must end with a recognizable true value.  (Severi
Code not contained in explicit package at line 1, column 1.  Violates encapsulation.  (Severity: 4)
No "VERSION" variable found at line 1, column 1.  See page 404 of PBP.  (Severity: 2)
Code before strictures are enabled at line 1, column 1.  See page 429 of PBP.  (Severity: 5)
Code before warnings are enabled at line 1, column 1.  See page 431 of PBP.  (Severity: 4)
```

La referencia PBP se refiere al libro Perl Best Practices [11] de D. Conway. En la dirección http://refcards.com/docs/vromansj/perl-best-practices/refguide.pdf Puede encontrar una tarjeta de referencia resumen con las 256 PBP del libro.

1.7. Una Brevísima Introducción a las Expresiones Regulares

Las Expresiones Regulares (abreviadas *regexp*) se pueden encontrar en muchos editores como `vi`, `emacs`, en programas como `grep`, `grep` y en lenguajes de programación como `awk` y `sed`. Se usan en busquedas y sustituciones avanzadas. Las Expresiones Regulares habitualmente se delimitan entre barras de división (`/regexp/`) pero, si se preceden de la `m` (de *matching*) pueden usarse otros delimitadores (por ejemplo `m{regexp}`, `m#regexp#`, etc.).

Resumen de las regexp mas Usadas

Las expresiones regulares son un lenguaje para la descripción de lenguajes. Una expresión regular define un lenguaje. Por ejemplo, la expresión regular `/[a-z]+/` define el lenguaje formado por las cadenas que consisten en repeticiones de una o mas letras minúsculas.

La tabla 1.3 muestra algunos de los metasímbolos usados en las expresiones regulares. En http://refcards.com/docs/trusketti/p puede encontrar un PDF con una hoja de referencia que resume el uso de las expresiones regulares en Perl.

Para saber más sobre expresiones regulares, estudie el capítulo 3. 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`.

El Operador de Binding

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

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

Si se omiten el operador `=~` y la variable entonces se usa la variable por defecto `$_`:

```
pp2@nereida:~/doc/2005_2006$ perl -wde 0
main::(-e:1):   0
  DB<1> $_ = '-3.2'
  DB<2> print "yes" if /\d+\.?\d*/ # igual que $_ =~ /\d+\.?\d*/
yes
  DB<3> print "yes" if /^\d+\.?\d*$/ # No casa. $_ empieza por '-'
  DB<4> print "yes" if /^-?\d+\.?\d*$/
yes
```

`*`	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 1.3: Algunos metasímbolos usados en las expresiones regulares

Clases de Caracteres

A menudo resulta necesario comprobar si la variable contiene una cifra, una vocal, o caracteres de control particulares. Una `clase de caracteres` se define 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(~,@,;,:^,_)
```

Se puede definir una clase de caracteres valiéndose de la complementaria mediante el uso del circumflejo `"^"`:

```
[^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 1.4: Abreviaturas para clases expreg

Paréntesis con memoria

Después de una operación de matching lo que casó con el primer paréntesis se conserva en la variable `$1`, lo que casó con el segundo en `$2`, etc.

Véase el siguiente ejemplo en el que se imprime lo que casó con los paréntesis décimo, undécimo, etc.:

```
lusasoft@LusaSoft:~$ perl -wde 0
main::(-e:1):   0
  DB<1> $x = "hello world!\n"
  DB<2> $x =~ /(h)(e)(l)(l)(o)( )(w)(o)(r)(l)(d)(!)(\n)/; print "$10 $11 $12 $13"
l d !
```

1.8. Un Programa Simple

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

```
lhp@nereida:~/Lperl/src$ cat -n example1.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3
 4  while (defined($_ = <STDIN>)) {
 5    if (/__END__/) { last; }
 6    elsif (/LHP/) { print; }
 7  }
 8
 9  =head1 NAME example1.pl
10
11  =head1 SYNOPSIS
12
13    example1.pl < file
14
15  =head1 DESCRIPTION
16
17  Reads file C<file> and prints all the lines containing
18  the string C<LHP> up to a line containing
19  the string C<__END__>. For example given the input file:
20
21  lhp@nereida:~/Lperl/src$ cat -n example1.input
22       1  Esta linea no se imprime
23       2  Esta linea si se imprime LHP
24       3  Esta no
25       4  Esta LHP si
26       5  __END__
27       6  y se acabo LHP
28       7  esta linea tampoco sale
29
30  the program produces the following output:
31
32    lhp@nereida:~/Lperl/src$ example1.pl < example1.input
33    Esta linea si se imprime LHP
34    Esta LHP si
```

La Documentación

Las líneas de la 9 a la 34 documentan el uso del programa mediante un lenguaje de marcas denominado `pod`. Podemos ver la documentación con `perldoc`:

```
lhp@nereida:~/Lperl/src$ perldoc example1.pl
```

```
EXAMPLE1(1)          User Contributed Perl Documentation         EXAMPLE1(1)



NAME example1.pl
SYNOPSIS
         example1.pl < file

DESCRIPTION
       Reads file "file" and prints all the lines containing the string "LHP"
       up to a line containing the string
       "__END__". For example given the input file:

       lhp@nereida:~/Lperl/src$ cat -n example1.input
            1  Esta linea no se imprime
            2  Esta linea si se imprime LHP
            3  Esta no
            4  Esta LHP si
            5  __END__
            6  y se acabo LHP
            7  esta linea tampoco sale

       the program produces the following output:

         lhp@nereida:~/Lperl/src$ example1.pl < example1.input
         Esta linea si se imprime LHP
         Esta LHP si



perl v5.8.8                       2007-02-23                     EXAMPLE1(1)
```

Para saber mas sobre el sistema de documentación Perl lea la sección 5.18.

Lectura En la línea 4 (`while (defined($_ = <STDIN>))`) almacenamos en la variable mágica `$_` 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.

Expresiones Regulares

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

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__/`.

Definición 1.8.1. Casar *significa que la cadena a la que se le hace el* binding*, 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__`.

El constructo `elsif`

El constructo *elsif* nos permite abreviar un `else` seguido de un `if`. Asi escribimos:

```
if (/__END__/) { last }  elsif (/LHP/) { print }
```

en vez de:

```
if (/__END__/) { last; }
else {
  if (/LHP/) { print; }
}
```

Un Programa sin Variables

Este es un ejemplo típico de programa Perl "moderadamente críptico": no aparece explícitamente ninguna variable salvo por la referencia a `$_` en la línea 4.

Ejecución

Sigue un ejemplo de ejecución:

```
lhp@nereida:~/Lperl/src$ cat -n example1.input
 1  Esta linea no se imprime
 2  Esta linea si se imprime LHP
 3  Esta no
 4  Esta LHP si
 5  __END__
 6  y se acabo LHP
 7  esta linea tampoco sale
 8
lhp@nereida:~/Lperl/src$ cat example1.input | example1.pl
Esta linea si se imprime LHP
Esta LHP si
```

Antes de seguir, responda a las siguientes preguntas:

Ejercicio 1.8.1. *Consideremos el siguiente programa:*

```
lhp@nereida:~/Lperl/src$ cat -n muerte_prematura4.pl
     1  #!/usr/local/bin/perl -w
     2  use strict;
     3  print while <STDIN>;
```

Véase un ejemplo de ejecución:

```
lhp@nereida:~/Lperl/src$ cat -E mp4.in2
0$
$
0$
lhp@nereida:~/Lperl/src$ cat mp4.in2 | muerte_prematura4.pl
0

0
```

Explique la salida. Conteste a las siguientes preguntas:

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

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

1.9. Breve Introducción al Manejo de Excepciones

Si se desea controlar las interrupciones que puedan ocurrir durante la entrada podemos establecer un código que maneje la señal `INT`, la cual es producida cuando se teclea `CTRL-C`. El manejador debe ser una subrutina o una referencia a una subrutina. He aqui un ejemplo:

```
lhp@europa:~/projects/perl/src/perltesting$ cat -n io.pl
#!/usr/bin/perl -w
use strict;

$SIG{INT} = sub {
  print "¡Que forma tan violenta de terminar!\n";
  exit(1);
};

print "Tu nombre: ";
my $x = <>;
chomp($x);
print "Hola $x\n";
```

Aun cuando es adelantar mucho, la expresión `$SIG{INT}` denota la entrada `INT` del hash/diccionario `%SIG`. 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
```

Los elementos individuales de un `hash` se acceden prefijando el nombre del hash de un `$` seguido de la clave entre llaves:

```
%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:

La palabra `sub` seguida de un código retorna un escalar que es una referencia a una subrutina cuya implementación viene dada por dicho código. Vea el siguiente ejemplo con el depurador:

```
casiano@tonga:/tmp$ perl -wde 0
main::(-e:1):   0
  DB<1> $x = sub { print "Hola mundo\n" }
  DB<2> $x->()
Hola mundo
```

Si ejecutamos el programa de ejemplo anterior obtenemos una salida como:

```
lhp@europa:~/projects/perl/src/perltesting$ ./io.pl
Tu nombre: Juan
Hola Juan
```

Si presionamos `CTRL-C` se produce la salida:

```
lhp@europa:~/projects/perl/src/perltesting$ ./io.pl
Tu nombre: ¡Que forma tan violenta de terminar!
```

1.10. Autentificación Automática

En la siguiente sección comenzaremos a usar el entorno de control de versiones Subversion usando como protocolo ssh. Para evitar la solicitud de claves cada vez que se comunica con el repositorio establezca autentificación SSH automática. Para ver como hacerlo puede consultar las instrucciones en:

http://search.cpan.org/ casiano/GRID-Machine/lib/GRID/Machine.pod#INSTALLATION

Tambien puede ayudarle leer el capítulo: Conexiones con ssh de los apuntes de Programación en Paralelo II.

Consulte también las páginas del manual Unix de

1. ssh,
2. ssh-key-gen,
3. ssh_config,
4. scp,
5. ssh-agent,
6. ssh-add,
7. sshd

1.11. Uso de Subversion

Use Subversion: Creación de un Repositorio Parece que en `banot` esta instalado subversion. Para crear un repositorio emita el comando `svnadmin create`:

```
-bash-3.1$ uname -a
Linux banot.etsii.ull.es 2.6.24.2 #3 SMP Fri Feb 15 10:39:28 WET 2008 i686 i686 i386 GNU/Linux
-bash-3.1$ svnadmin create /home/loginname/repository/
-bash-3.1$ ls -l repository/
total 28
drwxr-xr-x 2 loginname apache 4096 feb 28 11:58 conf
drwxr-xr-x 2 loginname apache 4096 feb 28 11:58 dav
drwxr-sr-x 5 loginname apache 4096 feb 28 12:09 db
-r--r--r-- 1 loginname apache    2 feb 28 11:58 format
drwxr-xr-x 2 loginname apache 4096 feb 28 11:58 hooks
drwxr-xr-x 2 loginname apache 4096 feb 28 11:58 locks
-rw-r--r-- 1 loginname apache  229 feb 28 11:58 README.txt
```

Una alternativa a considerar es ubicar el repositorio en un dispositivo de almacenamiento portable (pendriver)

Añadiendo Proyectos

Ahora esta en condiciones de añadir proyectos al repositorio creado usando `svn import`:

```
[loginname@tonga]~/src/perl/> uname -a
Linux tonga 2.6.24.2 #1 SMP Thu Feb 14 15:37:31 WET 2008 i686 i686 i386 GNU/Linux
[loginname@tonga]~/src/perl/> pwd
/home/loginname/src/perl
[loginname@tonga]~/src/perl/> ls -ld /home/loginname/src/perl/Grammar-0.02
drwxr-xr-x 5 loginname Profesor 4096 feb 28  2008 /home/loginname/src/perl/Grammar-0.02
[loginname@tonga]~/src/perl/> svn import -m 'Grammar Extended Module' \
                                        Grammar-0.02/ \
                                        svn+ssh://banot/home/loginname/repository/Grammar
Añadiendo      Grammar-0.02/t
Añadiendo      Grammar-0.02/t/Grammar.t
Añadiendo      Grammar-0.02/lib
Añadiendo      Grammar-0.02/lib/Grammar.pm
Añadiendo      Grammar-0.02/MANIFEST
Añadiendo      Grammar-0.02/META.yml
Añadiendo      Grammar-0.02/Makefile.PL
Añadiendo      Grammar-0.02/scripts
Añadiendo      Grammar-0.02/scripts/grammar.pl
Añadiendo      Grammar-0.02/scripts/Precedencia.yp
Añadiendo      Grammar-0.02/scripts/Calc.yp
Añadiendo      Grammar-0.02/scripts/aSb.yp
Añadiendo      Grammar-0.02/scripts/g1.yp
Añadiendo      Grammar-0.02/Changes
Añadiendo      Grammar-0.02/README

Commit de la revisión 2.
```

En general, los pasos para crear un nuevo proyecto son:

```
* mkdir /tmp/nombreProyecto
* mkdir /tmp/nombreProyecto/branches
* mkdir /tmp/nombreProyecto/tags
* mkdir /tmp/nombreProyecto/trunk
* svn mkdir file:///var/svn/nombreRepositorio/nombreProyecto -m 'Crear el proyecto nombreProyecto'
* svn import /tmp/nombreProyecto \
          file:///var/svn/nombreRepositorio/nombreProyecto \
          -m "Primera versión del proyecto nombreProyecto"
```

Obtener una Copia de Trabajo

La copia en `Grammar-0.02` ha sido usada para la creación del proyecto, pero no pertenece aún al proyecto. Es necesario descargar la copia del proyecto que existe en el repositorio. Para ello usamos `svn checkout`:

```
[loginname@tonga]~/src/perl/> rm -fR Grammar-0.02
[loginname@tonga]~/src/perl/> svn checkout svn+ssh://banot/home/loginname/repository/Grammar Grammar
A    Grammar/t
A    Grammar/t/Grammar.t
A    Grammar/MANIFEST
A    Grammar/META.yml
A    Grammar/lib
A    Grammar/lib/Grammar.pm
A    Grammar/Makefile.PL
```

```
A    Grammar/scripts
A    Grammar/scripts/grammar.pl
A    Grammar/scripts/Calc.yp
A    Grammar/scripts/Precedencia.yp
A    Grammar/scripts/aSb.yp
A    Grammar/scripts/g1.yp
A    Grammar/Changes
A    Grammar/README
Revisión obtenida: 2
```

Ahora disponemos de una copia de trabajo del proyecto en nuestra máquina local:

```
[loginname@tonga]~/src/perl/> tree Grammar
Grammar
|-- Changes
|-- MANIFEST
|-- META.yml
|-- Makefile.PL
|-- README
|-- lib
|   `-- Grammar.pm
|-- scripts
|   |-- Calc.yp
|   |-- Precedencia.yp
|   |-- aSb.yp
|   |-- g1.yp
|   `-- grammar.pl
`-- t
    `-- Grammar.t

3 directories, 12 files
[loginname@tonga]~/src/perl/>
[loginname@tonga]~/src/perl/> cd Grammar
[loginname@tonga]~/src/perl/Grammar/> ls -la
total 44
drwxr-xr-x 6 loginname Profesor 4096 feb 28  2008 .
drwxr-xr-x 5 loginname Profesor 4096 feb 28  2008 ..
-rw-r--r-- 1 loginname Profesor  150 feb 28  2008 Changes
drwxr-xr-x 3 loginname Profesor 4096 feb 28  2008 lib
-rw-r--r-- 1 loginname Profesor  614 feb 28  2008 Makefile.PL
-rw-r--r-- 1 loginname Profesor  229 feb 28  2008 MANIFEST
-rw-r--r-- 1 loginname Profesor  335 feb 28  2008 META.yml
-rw-r--r-- 1 loginname Profesor 1196 feb 28  2008 README
drwxr-xr-x 3 loginname Profesor 4096 feb 28  2008 scripts
drwxr-xr-x 6 loginname Profesor 4096 feb 28  2008 .svn
drwxr-xr-x 3 loginname Profesor 4096 feb 28  2008 t
```

Observe la presencia de los subdirectorios de control `.svn`.

Actualización del Proyecto

Ahora podemos modificar el proyecto y hacer públicos los cambios mediante `svn commit`:

```
loginname@tonga]~/src/perl/Grammar/> svn rm META.yml
D         META.yml
[loginname@tonga]~/src/perl/Grammar/> ls -la
```

```
total 40
drwxr-xr-x 6 loginname Profesor 4096 feb 28  2008 .
drwxr-xr-x 5 loginname Profesor 4096 feb 28 12:34 ..
-rw-r--r-- 1 loginname Profesor  150 feb 28 12:34 Changes
drwxr-xr-x 3 loginname Profesor 4096 feb 28 12:34 lib
-rw-r--r-- 1 loginname Profesor  614 feb 28 12:34 Makefile.PL
-rw-r--r-- 1 loginname Profesor  229 feb 28 12:34 MANIFEST
-rw-r--r-- 1 loginname Profesor 1196 feb 28 12:34 README
drwxr-xr-x 3 loginname Profesor 4096 feb 28 12:34 scripts
drwxr-xr-x 6 loginname Profesor 4096 feb 28  2008 .svn
drwxr-xr-x 3 loginname Profesor 4096 feb 28 12:34 t
[loginname@tonga]~/src/perl/Grammar/> echo "Modifico README" >> README
[loginname@tonga]~/src/perl/Grammar/> svn commit -m 'Just testing ...'
Eliminando     META.yml
Enviando       README
Transmitiendo contenido de archivos .
Commit de la revisión 3.
```

Observe que ya no es necesario especificar el lugar en el que se encuentra el repositorio: esa información esta guardada en los subdirectorios de administración de subversion `.svn`

El servicio de subversion parece funcionar desde fuera de la red del centro. Véase la conexión desde una maquina exterior:

```
pp2@nereida:/tmp$  svn checkout svn+ssh://loginname@banot.etsii.ull.es/home/loginname/repository/Grammar G
loginname@banot.etsii.ull.es's password:
loginname@banot.etsii.ull.es's password:
A    Grammar/t
A    Grammar/t/Grammar.t
A    Grammar/MANIFEST
A    Grammar/lib
A    Grammar/lib/Grammar.pm
A    Grammar/Makefile.PL
A    Grammar/scripts
A    Grammar/scripts/grammar.pl
A    Grammar/scripts/Calc.yp
A    Grammar/scripts/Precedencia.yp
A    Grammar/scripts/aSb.yp
A    Grammar/scripts/g1.yp
A    Grammar/Changes
A    Grammar/README
Revisión obtenida: 3
```

Comandos Básicos

- Añadir y eliminar directorios o ficheros individuales al proyecto

  ```
  svn add directorio_o_fichero
  svn remove directorio_o_fichero
  ```

- Guardar los cambios

  ```
  svn commit -m "Nueva version"
  ```

- Actualizar el proyecto

  ```
  svn update
  ```

Referencias
Consulte

1. El libro *Version Control with Subversion* [12] disponible en http://svnbook.red-bean.com/en/1.4/svn-book.html .
2. Vea la página de la ETSII http://cc.etsii.ull.es/svn .
3. Vea los capítulos disponibles del libro Subversion in Action de la editorial Manning
4. Practical Subversion de APress
5. La hoja de referencia en http://www.digilife.be/quickreferences/QRC/Subversion%20Quick%20Reference%20Card.pdf

En KDE puede instalar el cliente gráfico KDEsvn. Puede encontrar algunas capturas de pantalla del mismo en http://kdesvn.alwins-world.de/screenshots.

1.12. Práctica: Area de un Círculo

Escriba un programa que solicite de la entrada un radio de una circunferencia e imprima el área del correspondiente círculo ($S = \pi \times r^2$). El proceso debe repetirse hasta que el radio contiene una cadena no numérica. Compruebe además que el radio no es negativo.

1. Escriba la documentación de uso del programa (véase `perldoc` perlpod).
2. Encuentre los errores por medio del depurador (véase perldebtut).
3. Utilice subversion. Cree un repositorio. Añada al repositorio un proyecto `practicas_lhp` conteniendo la jerarquia

```
practicas_lhp
     |
     |-trunk
     |   |
     |   `-area_de_un_circulo
     |            |
     |            |- area_de_un_circulo.pl
     |            |- MANIFEST
     |            |- Makefile
     |            |- test.input
     |            `- test.expected
     |-branches
     |
     `-tags
```

   - Cree una copia de trabajo del tronco en un directorio `practicas_lhp`
   - Añada la propiedad `svn:executable` a `area_de_un_circulo.pl`
   - Añada la propiedad `svn:keywords` a `area_de_un_circulo.pl`
   - `MANIFEST` es la lista de ficheros que forman parte de la distribución
   - El programa Makefile debe tener al menos tres objetivos:
     - El objetivo `dist` creará un fichero tar.gz `area_de_un_circulo.tar.gz` conteniendo todos los ficheros de la distribución
     - El objetivo `test` ejecutará `area_de_un_circulo.pl` con entrada en `test.input` y comprobará que la salida es igual a la ubicada en `test.expected` (Véase el comando cmp)

  - El objetivo `man` deberá producir un fichero de manual a partir de la documentación (véase pod2man). Un ejemplo:

    ```
    $ pod2man area_de_un_circulo.pl > ./docs/man1/area_de_un_circulo.1
    ```

    Ahora podemos ver el manual con el comando:

    ```
    $ man -Mdocs area_de_un_circulo
    ```

4. Declare todas sus variables. Asegúrese de que usa strict.

   Declare las variable mediante la palabra reservada `my`:

   ```
   my $a =4;
   {
     my ($b, $c) = ("a", 9.2);
     $a = $b + 2*$c;
   }
   print $a;
   ```

   Una variable declarada con `my` tiene por ámbito el del bloque en que fué declarada o el fichero si esta fuera de bloques.

5. Use perlcritic

6. Constantes: Para definir la constante $\pi = 4 \times atan(1) = 4 \times \pi/4$ use el módulo `constant`:

   ```
   pl@nereida:~/src/perl/uploadlab/Education-Labs/lib/Education$ perl -wde 0
   main::(-e:1):   0
     DB<1> use constant PI    => 4 * atan2(1, 1)
     DB<2> print PI."\n"
   3.14159265358979
   ```

7. Para obtener información sobre `atan2` use `perldoc -f atan2`:

   ```
   lhp@nereida:~/Lperl/src$ perldoc -f atan2
          atan2 Y,X
                  Returns the arctangent of Y/X in the range -PI to PI.
                  For the tangent operation, you may use the "Math::Trig::tan" function,
                  or use the familiar relation:

                      sub tan { sin($_[0]) / cos($_[0])  }

                  Note that atan2(0, 0) is not well-defined.
   ```

8. Compruebe el comportamiento de su programa contra entradas

   a) Cero, muy cerca de cero

   b) Negativas

   c) Cadenas de caracteres

   d) Cadenas de caracteres que contienen números, p. ej. `"one 1.0 two 2.0"`

   e) Diferentes formatos de flotante: `1e2`, `1.0e-1`, etc.

9. Controle la aparición de interrupciones de teclado estableciendo un manejador de la señal `INT`

1.13. Arrays y Listas

1.13.1. Introducción

Una variable `array` puede almacenar una secuencia de valores escalares, los cuales se identifican unívocamente a través de un índice.

```
@days  = (1,2,3,4,5,6,7);
```

El operador `..`

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 # Los elementos son mostrados sin separación
45678
  DB<3> p "@b" # Interpolación: los elementos son separados
  4 5 6 7 8
  DB<4> x @b
0  4
1  5
2  6
3  7
4  8
  DB<5> @a = 2.1 .. 4.1
  DB<6> 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
```

Acceso a los Elementos de un Array

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

```
foreach my $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 *for*, se usará la *variable mágica por defecto* `$_`. Así el bucle anterior es equivalente a:

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

Troceado de `arrays`

Un `slice` o `trozo` de un `array` resulta de indexar el array en un subconjunto de índices:

```
  @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]
```

La función `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, cuando `reverse` es aplicado a una lista 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
  DB<9> @a = qw{one two three}
  DB<10> $x = reverse @a
  DB<11> p $x
eerhtowteno
```

Si se aplica a una cadena en un contexto escalar produce la recíproca de la cadena:

```
  DB<29> q
lhp@nereida:~/public_html/cgi-bin$ perl -wde 0
main::(-e:1):   0
  DB<1> $a = "dabale arroz a la zorra el abad"
  DB<2> p scalar(reverse($a))
daba le arroz al a zorra elabad
```

Sin embargo, en un contexto de lista la cadena queda sin modificar:

```
  DB<3> p reverse($a)
dabale arroz a la zorra el abad
```

Tenga cuidado si esta trabajando en un entorno que usa UTF-8. En ese caso `reverse` puede producir un resultado erróneo:

```
lhp@nereida:~/Lperl/src$ perl -we ’print reverse("dábale arroz a la zorra el abad")."\n"’
daba le arroz al a zorra elab##d
```

Para solucionarlo es necesario usar el módulo `utf8` y poner el fichero de salida en modo `:utf8`:

```
lhp@nereida:~/Lperl/src$ cat -n reverse.pl
 1  #!/usr/local/bin/perl -w
 2  use strict;
 3  use warnings;
 4  use utf8;
 5  binmode STDOUT, ':utf8';
 6  my $x = shift || "dábale arroz a la zorra el abad";
 7  print reverse($x)."\n"; # Contexto escalar forzado por el "."
```

Cuando se ejecuta se obtiene:

```
lhp@nereida:~/Lperl/src$ reverse.pl
daba le arroz al a zorra elabád
```

Dinamicidad de los Arrays

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`.

Ultimo Indice de un Array

La variable `$#days` almacena el índice del último elemento del `array`.

Indices Negativos

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.
DB<5> p $#a
9
```

Número de elementos de un Array

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`.

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

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
```

Diferencias entre Arrays y Listas

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;
```

Asignación a Listas y Asignación a Arrays

Se puede asignar listas de valores a listas de variables:

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

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

En una *Asignación a una lista*, si hay mas variables en la lista que elementos en la parte derecha , las variables extra quedan indefinidas (`undef`). Si por el contrario hay menos variables que elementos en la parte derecha, los elementos extra son ignorados.

Veamos un ejemplo:

```
lhp@nereida:~/Lperl/src$ perl -wde 0
main::(-e:1):   0
  DB<1> @a = 1..3
  DB<2> @b = 1..5
  DB<3> ($a,$b) = @a
  DB<4> x ($a,$b)
0  1
1  2
  DB<5> ($c,$b,$a) = (2,7)
  DB<6> x ($c,$b,$a)
0  2
1  7
2  undef
  DB<7> ($c,$b,$a) = (2,7,9,12,@b)
  DB<8> x ($c,$b,$a)
0  2
1  7
2  9
```

Sin embargo si la parte izquierda es un array, la variable se hace igual a la lista que esta en la parte derecha:

```
  DB<9> @b = reverse @a
  DB<10> x @b
0  3
1  2
2  1
```

El Operador `qw`

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);
```

Un Array No Inicializado es Vacío

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) == 0  es 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> $" = "," # separador de output de arrays
  DB<7> p "@a"
0,,2,3,4,,6,,8,9
  DB<8> @a[1..7] = ()
  DB<9> p "@a"
0,,,,,,,,8,9
```

Interpolación 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 1.13.1. *¿Cuál es la explicación para la salida de* `p $x` *en el último ejemplo?*

Ejercicio 1.13.2. *Dado la siguiente sesión con el depurador*

```
nereida:~/perl/src> perl -de 0
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?. ¿Se produce alguna dvertencia? ¿Que ocurre si se cambia la última línea por esta otra?

```
DB<3> no warnings; $" = ', '; print "@a"
```

Bucles sobre `arrays`: el Indice como alias 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:

```
lhp@nereida:~/Lperl/src$ cat -n foreach1.pl
     1  #!/usr/bin/perl -w
     2  use strict;
```

```

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

su ejecución produce la siguiente salida:

```
lhp@nereida:~/Lperl/src$ foreach1.pl
2 4 6 8 10 12 14 16 18 20
```

1.13.2. Operadores de Listas

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 3.

El siguiente código corresponde a la segunda forma de uso: 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 1.13.3. *Explique la interpretación que hace Perl de los siguientes fragmentos de código. Consulte 1.13.2 y el manual de Perl para entender el uso de la función* `grep`*:*

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

El operador `map`

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

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

El operador `-s` retorna el tamaño de un fichero. 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`. Observe la ausencia de coma.

Ejercicio 1.13.4.

1. *Explique que salida produce la transformación:*

   ```
   @a = grep { defined } map { /(\d+)/; $1 } glob('/tmp/*')
   ```

   El operador `glob` *produce una lista con los ficheros descritos por la expresión* shell *que se le pasa como argumento (vea* `perldoc -f glob`*). La expresión regular* `/(\d+)/` *casa con números enteros sin signo. Al estar paréntizada la cadena que ha casado queda automáticamente guardada* *en la variable especial* `$1`. El valor devuelto por un bloque es la última sentencia evaluada, *en este caso es el valor de* `$1`.

Cuando un casamiento con una expresión regular tiene lugar en un contexto de lista se devuelve una lista con las cadenas que han casado con los paréntesis:

```
  DB<0> $a = "hola 123 b 56 c"
  DB<1> @a = $a =~ m{(\d+)\s(\w+)}
  DB<2> x @a
0  123
1  'b'
  DB<3> @a = $a =~ m{(\d+)\s(\w+)}g
  DB<4> x @a
0  123
1  'b'
2  56
3  'c'
```

Quizá le ayude a entender la salida la siguiente sesión con el depurador:

```
  DB<0> x glob('/tmp/*.pl')
0  '/tmp/cgisearch.pl'
1  '/tmp/ficha.pl'
2  '/tmp/uploadpractica.config.pl'
3  '/tmp/uploadpractica.pl'
  DB<1> x map { m{/([^u/]+)$}; $1 } glob('/tmp/*.pl')
0  'cgisearch.pl'
1  'ficha.pl'
2  undef
3  undef
  DB<2> x grep { defined} map { m{/([^u/]+)$}; $1 } glob('/tmp/*.pl')
0  'cgisearch.pl'
1  'ficha.pl'
```

2. *La expresión*

```
@a = map { /(\d+)/ } glob('/tmp/*')
```

produce el mismo resultado que la anterior. ¿Porqué? ¿Que retorna cada una de las evaluaciones individuales del bloque? ¿En que contexto lista o escalar se evalúa `/(\d+)/`*?*

Observe este comando en el depurador:

```
 DB<1> x @a = map { /(\d+)/ } ("a123", "b", "c42")
0  123
1  42
```

3. *¿Que salida produce la siguiente variante de la expresión?*

```
@a = map { scalar(/(\d+)/) } glob('/tmp/*')
```

`push`, `pop`, `shift`, `unshift` 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` ó `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)
```

La Función `join`

La función `join` convierte un arreglo en un escalar La llamada a `join EXPR, LIST` concatena las cadenas en `LIST` separadas por `EXPR`:

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

La Función `split`

La función `split` es la inversa de `join`: convierte un escalar en un arreglo. `split /PATTERN/,EXPR,LIMIT` retorna el arreglo resultante de partir la cadena `EXPR`. Si se especifica `LIMIT` indica el número máximo de campos en los que se divide.

```
lhp@nereida:~/projects/perl/src$ perl -wde 0
main::(-e:1):   0
  DB<1> $a = 'a:b:c:d:e'
  DB<2> @a = split /:/, $a
  DB<3> x @a
0  'a'
1  'b'
2  'c'
3  'd'
4  'e'
  DB<4> @a = split /:/, $a, 3
  DB<5> x @a
0  'a'
1  'b'
2  'c:d:e'
  DB<6> $a = 'a:b:c:d::'
  DB<7> @a = split /:/, $a
  DB<8> x @a # Los nulos finales se suprimen
0  'a'
1  'b'
2  'c'
3  'd'
  DB<9> @a = split /:/, $a, -1
  DB<10> x @a # No si LIMIT es negativo
0  'a'
1  'b'
2  'c'
3  'd'
4  ''
5  ''
```

Observe que en `split` el primer parámetro es una expresión regular mientras que en `join` es un carácter.

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

El Bloque de Comparación

La función `sort` admite como primer argumento un bloque que determina la función de comparación. Dicho bloque depende de dos variables ”especiales” `a` y `b`.

El valor retornado por un bloque es el valor asociado con la última sentencia del bloque. En este caso `sort` espera que el bloque devuelva -1, 0 ó 1. Dicho valor es utilizado como elemento de comparación. Por ejemplo:

```
lhp@nereida:~/Lperl/src$ perl -dwe 0
main::(-e:1):   0
  DB<1> @a = (4, 7, 9, 12, -1)
  DB<2> p 4 <=> 7
-1
  DB<3> p 9 <=> 2
1
  DB<4> p 13 <=> 12+1
0
  DB<5> @a = sort { $a <=> $b } @a;
  DB<6> p "@a"
-1 4 7 9 12
  DB<7> @a = sort { $b <=> $a } @a;
  DB<8> p "@a"
12 9 7 4 -1
```

Ejemplo de Ordenación

El siguiente ejemplo ordena los usuarios de un sistema unix atendiendo a su `uid`:

```
lhp@nereida:~/projects/perl/src$ cat -n sortperuid.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  my @user =  grep { $_ !~ /^#/ } ‘cat /etc/passwd‘;
 4  my (@name, @uid, $x);
 5
 6  for (my $i=0; $i < @user; $i++) {
 7    ($name[$i], $x, $uid[$i]) = split ’:’, $user[$i];
 8  }
 9
10  @name = @name[
11    sort {$uid[$a] <=> $uid[$b]} 0..$#name
12  ];
13
14  local $" = "\n";
15  print "@name\n";
```

En la línea 3 se obtiene en `@user` la lísta de líneas no comentadas en `/etc/passwd`. en las líneas 6-8 se inicializan los arrays `@name` y `@uid` a los nombres (login) y uid de los usuarios (campos primero y tercero de `/etc/passwd`). Las líneas 10-12 ordenan primero la lista `0..$#name` de acuerdo con el valor de `uid`. El nuevo conjunto ordenado de índices es utilizado para reindexar el array `name`.

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

Los Módulos `List::Util` y `List::MoreUtils`

El módulo List::Util provee un conjunto de funciones para el manejo de listas. Veamos algunos ejemplos de uso de `List::Util`:

```
  DB<1> use List::Util qw(first max maxstr min minstr reduce shuffle sum)
  DB<2> @a = map { int(rand 20) } 1..5
  DB<3> x @a
0  8
1  9
2  4
3  16
4  5
  DB<4> x min @a
0  4
  DB<5> x max @a
0  16
  DB<6>x first { $_ > 8 } @a
0  9
  DB<7> x sum @a
0  42
  DB<8> x shuffle @a
0  8
1  5
2  16
3  4
4  9
  DB<9> x reduce { $a." $b" } @a
0  '8 9 4 16 5'
```

Existe también un módulo List::MoreUtils que provee mas funciones para el manejo de listas. He aqui un fragmento de la sección *SYNOPSIS* de la documentación:

```
          use List::MoreUtils qw(any all none notall true false firstidx first_index
                                 lastidx last_index insert_after insert_after_string
                                 apply after after_incl before before_incl indexes
                                 firstval first_value lastval last_value each_array
                                 each_arrayref pairwise natatime mesh zip uniq minmax);
```

Veamos algunos ejemplos de uso de `List::MoreUtils`:

```
lhp@nereida:~/Lperl/src/XSUB/h2xsexample/Coord/script$ perl -de 0
main::(-e:1):   0
  DB<1> use List::MoreUtils qw(:all) # Importar todas las funciones
  DB<2> @a = (1..8,-2,-3)
  DB<4> print "@a" if any { $_ > 0 } @a
1 2 3 4 5 6 7 8 -2 -3
  DB<5> print "@a" if all { $_ > 0 } @a

  DB<6> print (false {$_ > 0} @a),"\n" # Número de elementos para los que es falsa la condición
2
  DB<7> print (firstidx {$_ < 0} @a),"\n"
```

```
8
  DB<8> @a = (1..3,2..5,3..6)
  DB<9> x uniq(@a) # Array con los elementos distintos
0  1
1  2
2  3
3  4
4  5
5  6
  DB<8> @a = 1..5; @b = 'a'..'e'; @c = 10..14
  DB<9> x mesh @a, @b, @c # Mezcla los 3 arrays
0  1
1  'a'
2  10
3  2
4  'b'
5  11
6  3
7  'c'
8  12
9  4
10  'd'
11  13
12  5
13  'e'
14  14
```

El Array Especial `@ARGV`

El array especial `@ARGV` contiene la lista de argumentos del programa.

El array `@ARGV` es usado como array por defecto cuando se realiza una operación de arrays dentro del programa principal:

```
nereida:~/etc> perl -wde 0 one two three
main::(-e:1):   0
  DB<1> p "@ARGV"
one two three
  DB<2> p $^X # La variable mágica $^X tiene el camino al intérprete Perl utilizado
/usr/bin/perl
  DB<3> $a = shift # No se especifica el arrray: es @ARGV
  DB<4> x $a
0  'one'
  DB<5> $b = pop
  DB<6> x $b
0  'three'
  DB<7> x $^O # La variable $^O contiene el nombre del sistema operativo.
0  'linux'
  DB<8> x $0  # La variable mágica $0 contiene el nombre del programa
0  '-e'
```

- La variable especial `$X̂` contiene el nombre del ejecutable Perl que esta siendo utilizado.
- La variable `$0` contiene el nombre del ejecutable. Si el programa fué ejecutado con la opción `-e command` su nombre es `-e`.
- La variable `$^O` contiene el nombre del sistema operativo.

Allanamiento de las listas

En contra de lo que quizá algunos podrían 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");
```

1.13.3. Ejercicios

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";
  ```

- 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?

Elemento o Trozo

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

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.

```
lhp@nereida:~/Lperl/src$ perl -wde 0
main::(-e:1):   0
  DB<1> @text = (1, 2, 3, 4, 5)
  DB<2> @text[@text] = 0
  DB<3> x @text
0  1
1  0
2  undef
3  undef
4  undef
5  undef
  DB<4> @text = (1, 2, 3, 4, 5)
  DB<5> $text[@text] = 0
  DB<6> x @text
0  1
1  2
2  3
3  4
4  5
5  0
```

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:

```
lhp@nereida:~/Lperl/src$ perl -wde 0
main::(-e:1):   0
  DB<1> !!cat prueba1.txt
12
23
3
4
2
1
  DB<2> open $f, 'prueba1.txt'
  DB<3> @x = <$f>
  DB<4> chomp(@x)
  DB<5> x @x[-3..-1]
0  4
1  2
2  1
  DB<6> $/ = '3'
  DB<7> open $f, 'prueba1.txt'
  DB<8> @x = <$f>
  DB<9> chomp(@x)
  DB<10> x @x
0  '12
```

```
2'
1  '
'
2  '
4
2
1
'
```

Leemos y almacenamos las líneas en el *array* `@x`. La orden `chomp` Elimina todos los separadores de registro de todas las líneas en la lista.

Ejercicio 1.13.5. *¿Porqué han desaparecido los* `3` *en la salida de la línea 10?*

grep

Explique la interpretación que hace Perl del siguiente fragmento de código. Consulte 1.13.2 y el manual de Perl para entender el uso de la función `grep`. Lea el capítulo 3 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/`.

1.13.4. Práctica: Fichero en Orden Inverso

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

La siguiente sesión con el debuger sugiere una solución:

```
pp2@nereida:/tmp$ perl -wde 0
main::(-e:1):   0
  DB<1> open $F, "prueba.txt" # Abrimos fichero
  DB<2> @a = <$F> # Leemos el fichero en un contexto de lista
  DB<3> x @a      # Que hay en @a?
0  1
1  2
2  3
3  4
4  5
5  6
6  7
```

1.13.5. Práctica: En Orden ASCIIbético

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

1.13.6. 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. La siguiente sesión con el depurador sugiere una posible solución:

```
  DB<1> !!cat prueba.txt # Tenemos un fichero de entrada
C
A
a
B
d
```

```
B
b
  DB<2> open $F, "prueba.txt" # Lo abrimos para lectura
  DB<3> @a = <$F>             # Y leemos
  DB<4> x @a                  # @a contiene las lineas
0  'C                         # incluyendo retornos de
'                             # carro
1  'A
'
2  'a
'
3  'B
'
4  'd
'
5  'B
'
6  'b
'
  DB<5> x chomp(@a)          # Número de registros eliminados. Haga perldoc -f chomp
0  7
  DB<6> x @a
0  'C '
1  'A'
2  'a'
3  'B'
4  'd'
5  'B'
6  'b'
  DB<7> x sort { uc($a) cmp uc($b) } @a # perldoc -f uc
0  'A'
1  'a'
2  'B'
3  'B'
4  'b'
5  'C '
6  'd'
```

1.13.7. Práctica: Indexación

Escriba un programa que lea una lista de números (entre 1 y 26) e imprima las letras mayúsculas correspondientes a 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
```

La siguiente sesión con el depurador muestra algunos de los problemas que pueden surgir asi como algunas de las posibles soluciones:

```
lhp@nereida:~/Lperl/src$ perl -wde 0
main::(-e:1):   0
  DB<1> !!cat prueba1.txt
12
23
3
```

```
4
2
1
  DB<2> open $F, "prueba1.txt"
  DB<3> @a = <$F>
  DB<4> @L = 'A'..'Z'
  DB<5> x @L[@a] # Mmm... no es exactamente lo que queremos
0  'M'
1  'X'
2  'D'
3  'E'
4  'C'
5  'B'
  DB<6> p scalar(@L)
26
  DB<7> @L[1..26] = 'A'..'Z'
  DB<8> x @L[@a]
0  'L'
1  'W'
2  'C'
3  'D'
4  'B'
5  'A'
```

1.13.8. 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, restas, productos y divisiones. Asumiremos también que cualesquiera dos terminales están separados por uno o mas 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 3.6.

  ```
  lhp@nereida:~/Lperl/src$ perl -wde 0
  main::(-e:1):   0
    DB<1> x split /\s+/, "2   3    7 +   * 4 -"
  0  2
  1  3
  2  7
  3  '+'
  4  '*'
  5  4
  6  '-'
  ```

- La necesidad de separar los terminales mediante blancos se ha hecho pensando en simplificar la labor de detectarlos, de manera que se pueda hacer usando `split`. ¿Se puede suprimir esta restricción sin complicar excesivamente la detección de terminales?

  La siguiente sesión con el depurador sugiere como hacerlo:

```
lhp@nereida:~/Lperl/src$ perl -wde 0
main::(-e:1):   0
  DB<1> $x = "2 13+2*3-"
  DB<2> @tokens = grep { defined } $x =~ m{\s*(\d+)|([-+*/])\s*}g
  DB<3> x @tokens
0  2
1  13
2  '+'
3  2
4  '*'
5  3
6  '-'
  DB<4> $x = "6 3/  2*3-"
  DB<5> @tokens = grep { defined } $x =~ m{\s*(\d+)|([-+*/])\s*}g
  DB<6> x @tokens
0  6
1  3
2  '/'
3  2
4  '*'
5  3
6  '-'
```

Observe que esta solución puede ser ”excesiva”:

```
  DB<7> $x = "6 3cc/  2*xx3-"

  DB<8> x grep { defined } $x =~ m{\s*(\d+)|([-+*/])\s*}g
0  6
1  3
2  '/'
3  2
4  '*'
5  3
6  '-'
```

- Utilice una lista auxiliar como pila para la evaluación. Los números se empujan, los operadores binarios extraen dos elementos y empujan el resultado. Compruebe que el operador binario es uno de los esperados (`[-+*/]`).
- 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 1.4.1). 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";
```

- Utilice la función `eval`. Esta función recibe una cadena y evalúa la expresión Perl que representa. Vea un ejemplo de uso con el depurador:

```
  DB<1> $x = '4 + 2'
  DB<2> p $x
4 + 2
  DB<3> $y = eval($x)
  DB<4> p $y
6
  DB<5> p eval('3+$y')
9
```

Para saber mas sobre la función `eval` consulte la documentación escribiendo `perldoc -f eval`.

- Recuerde que `*` y `+` son metasímbolos dentro de una expresión regular.
- La variable especial `$&` almacena la cadena que casó con la última expresión regular usada, `$1`, `$2`, etc. contienen las cadenas que casaron con los paréntesis que aparecen en la expresión regular.

1.13.9. Práctica: Ordenación Internacional

El módulo Sort::ArbBiLex permite generar subrutinas de ordenación a partir de una especificación de agrupaciones de caracteres equivalentes.

Escriba un programa que reciba como entrada un fichero de texto y dos números y ordene de acuerdo a los criterios de ordenación del español las líneas comprendidas entre esos dos números. Si no se proporcionan números ordenará todo el fichero. Si sólo se proporciona un número ordenará desde esa línea hasta el final del fichero.

```
$ spsort resumen.txt 10 30 # ordenar líneas entre la 10 y la 30
$ spsort resumen.txt 30    # de la 30 en adelante
$ spsort resumen.txt       # ordenar todo resumen.txt
```

Véase también

- International Sorting with Perl's sort

1.14. 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.

1.14.1. 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.

1.14.2. 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);
```

La diferencia entre el operador coma y el operador `=>` es que 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"
```

1.14.3. 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. En este ejemplo en la expresión `(%a, %c)` los hashes son automáticamente convertidos en listas:

```
%a = ( 'x', 5, 'y', 3);
%c = ( 'u', 5, 'v', 4);
@b = (%a, %c);
```

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
```

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

```
lhp@nereida:~/public_html/cgi-bin/tt$ perl -wde 0
main::(-e:1):   0
  DB<1> %a = (one, 1, two, 2, three)
Odd number of elements in hash assignment at (eval 5)
```

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
```

1.14.4. Troceado de un `hash`

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

```
hp@nereida:~/public_html/cgi-bin/tt$ perl -wde 0
main::(-e:1):   0
  DB<1> %a = (x => 5, y => 3, z => "woof", t => undef)
  DB<2> print @a{"y", "z"}
3woof
  DB<3> print @a{qw(y z) }
3woof
  DB<4> print @a{y, z}
Transliteration pattern not terminated at (eval 8)
```

Dos observaciones:

1. Puesto que la lista de claves no es un único identificador se hace conveniente el uso de comillas
2. El prefijo del trozo de un hash es `@`. El símbolo `%` se reserva para el hash completo. Un trozo de un hash es la lista de valores para las claves especificadas. No es un escalar (hay varios elementos) ni un `hash` (no ha claves)

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, a`

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?

1.14.5. Inversión de un Hash

Para construir el hash recíproco de un cierto hash `%h` podríamos escribir:

```
%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.

1.14.6. 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";
}
```

Si sólo quisieramos los valores escribiremos:

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

La función `each` será tratada en la sección 1.14.7.

1.14.7. La función `each`

La función `each` puede ser usada para iterar sobre un hash.

`each` en Contexto Escalar

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:

```
DB<1> %a = (juan=>5, pedro=>0, marta=>10)
DB<2> print "$a{$n} " while defined($n = each %a)
0 10 5
```

`each` en Contexto de Lista

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:

```
DB<4> print "$k => $v\n" while ($k, $v) = each %a
pedro => 0
marta => 10
juan => 5
```

Inicialización del Iterador

Cada hash es una estructura de datos que contiene un *iterador privado*. Esto permite el anidamiento de bucles `each`, siempre que esten iterando sobre hashes diferentes. El iterador es reinicializado automáticamente en las siguientes circunstancias:

1. Al usar las funciones `keys` o `values` sobre el hash. Por ejemplo:

   ```
   nereida:~/src/perl/YappWithDefaultAction/lib/Parse/Eyapp> perl -wde 0
   main::(-e:1):   0
     DB<1>  %a = (juan=>5, pedro=>0, marta=>10)
     DB<2> $n = each %a; print $n
   pedro
     DB<3> $n = each %a; print $n
   marta
     DB<4> @b = keys %a;  $n = each %a; print $n
   pedro
   ```

2. También es reinicializado si se almacena una nueva lista de valores en el *hash completo*
3. 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. Incluso si se inicializa un trozo:

```
lhp@nereida:~/public_html/cgi-bin/tt$ perl -wde 0
main::(-e:1):   0
  DB<1>  %a = (juan=>5, pedro=>0, marta=>10)
  DB<2> $n = each %a; print $n
pedro
  DB<3> @a{juan,marta} = (2,3)
  DB<4> $n = each %a; print $n
marta
  DB<5> x %a
0  'pedro'
1  0
2  'marta'
3  3
4  'juan'
5  2
```

1.14.8. Las funciones `delete` y `exists`

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
```

La Función `exists`

La función `exists` devuelve `TRUE` si el elemento especificado (de un hash o un array) ha sido inicializado, incluso si el valor correspondiente es `undef`.

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
```

Uso de `delete` sobre Trozos

También se puede pasar como argumento de `delete` un trozo del hash (para saber más sobre trozos de hashes vea la sección 1.14.4):

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

1.14.9. 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
```

1.14.10. Obtener el Conjunto de Elementos de una Lista

Ya vimos que la función `uniq` de `List::MoreUtils` devuelve el conjunto de elementos de una lista:

```
my @x = uniq 1, 1, 2, 2, 3, 5, 3, 4; # 1 2 3 5 4
my $x = uniq 1, 1, 2, 2, 3, 5, 3, 4; # 5
```

Una alternativa para obtener un array `@out` con los elementos no repetidos de un array `@in` es 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} = (); # Creamos un hash con claves en @in
@out = keys %saw;
```

1.14.11. Bloqueo de las Claves de un Hash

El módulo `Hash::Util` permite bloquear las claves de un hash. Observe la siguiente sesión con el depurador:

```
pp2@nereida:~/LGRID_Machine$ perl -wde 0
main::(-e:1):   0
  DB<1>  use Hash::Util qw(lock_keys unlock_keys lock_value  unlock_value lock_hash   unlock_hash hash_see
  DB<2> %h = (foo => 42, bar => 23)
  DB<3>  lock_keys(%h)
  DB<4> x %h
0  'bar'
1  23
2  'foo'
3  42
  DB<5> $h{chum} = 32
Attempt to access disallowed key 'chum' in a restricted hash at (eval 9)[/usr/share/perl/5.8/perl5db.pl:62
```

La llamada a `lock_keys` bloque las claves. Sin embargo es posible suprimir claves existentes:

```
  DB<6> delete($h{foo})
  DB<7> x %h
0  'bar'
1  23
  DB<8> $h{foo} = 5
```

En el momento del bloqueo es posible especificar que claves se bloquean:

```
  DB<9> %t = (a => 1, b => 2)
  DB<10> lock_keys(%t, keys(%t), qw{c d e})
  DB<11> @t{qw{c d e}} = 1..3
  DB<12> x %t
0  'e'
1  3
2  'c'
3  1
4  'a'
5  1
```

```
6  'b'
7  2
8  'd'
9  2
  DB<13> $t{x} = 4
Attempt to access disallowed key 'x' in a restricted hash at (eval 17)[/usr/share/perl/5.8/perl5db.pl:628]
```

Es posible liberar el bloqueo sobre una clave mediante `unlock_keys`:

```
  DB<14> unlock_keys(%t)
  DB<15> $t{x} = 4
  DB<16> x %t
0  'e'
1  3
2  'c'
3  1
4  'a'
5  1
6  'b'
7  2
8  'x'
9  4
10  'd'
11  2
```

Es posible bloquear los valores de un hash cuyas claves hayan sido bloqueadas:

```
DB<17> %d = qw{one 1 two 2 three 3}
DB<18> lock_keys(%d)
DB<19> lock_value(%d, 'two')
DB<20> $d{one} = 4
DB<21> $d{two} = 3
Modification of a read-only value attempted at (eval 35)[/usr/share/perl/5.8/perl5db.pl:628] line 2.
```

1.14.12. 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:

```
lhp@nereida:~/Lperl/src$ perl -wde 0 notas.txt
main::(-e:1):   0
  DB<1> @a = <>             # Leemos el fchero 'notas.txt' en @a
  DB<2> p @a[0..3]          # 4 primeras líneas
Ferrer Pérez, Eduardo & 9'6\\
García  García, Laura & 7'5 \\
García Medina, Anai & 6'5\\
García Rodríguez, Pedro & 7'2\\

  DB<3> %h = map { /(.*)\s+\&\s+(\d+'\d*).*/ } @a
  DB<4> p $h{'Ferrer Pérez, Eduardo'}
9'6
  DB<5> p $h{'García Medina, Anai'}
6'5
  DB<7> $h{$_} =~ s/'/./ for (keys %h)
  DB<8> p $h{'García Medina, Anai'}
6.5
  DB<9> @b = sort { $h{$b} <=> $h{$a} } keys %h
  DB<10> print "$_ => $h{$_}\n" for (@b[0..5])
Rodríguez González, Andrés => 10.0
Pedrín Ruiz, Ana => 9.6
Ferrer Pérez, Eduardo => 9.6
Pérez García, Adolfo => 9.5
Rodríguez Rodríguez, Jorge => 9.2
Marrero Díaz, Jose Antonio => 8.0
  DB<11> $max = max(map { length } @b)    # Salida formateada
  DB<12> printf("%-${max}s\t=>\t%4s\n",$_,$h{$_}) for @b[0..3]
Rodríguez González, Andrés      =>      10.0
Pedrín Ruiz, Ana                =>       9.6
Ferrer Pérez, Eduardo           =>       9.6
Pérez García, Adolfo            =>       9.5
```

El uso de la expresión regular en la línea 3 en un contexto de lista hace que `map` construya una lista de pares (nombre, nota) que es ”enrollada” en el hash `%h`. Los pares (nombre, nota) son retornados por la expresión regular al ser el resultado del matching con el primer y segundo paréntesis.

Puede que le convenga leer las secciones 3.1, 3.2 y 3.4 y en particular el uso de paréntesis y el operador de sustitución en las expresiones regulares.

Para la ordenación repase la sección 1.13.2 y en particular el ejemplo en el que los usuarios del sistema se ordenan según su `uid`.

1.15. Subrutinas

1.15.1. 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, opcionalmente precedido de un `&`. Como ocurre en C las subrutinas son objetos globales, visibles desde cualquier punto del programa. Sin embargo se permite ubicarlas en el interior de otra subrutina:

```
hp@nereida:~/Lperl/src$ cat -n nestedsubs.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  my $n;
 4
 5  sub marine {
 6
 7    sub submarine {
 8      $n +=1;
 9      print "sub submarine $n!\n";
10    }
11
12    $n +=1;
13    print "sub marine $n!\n";
14  }
15
16  marine;
17  submarine;
```

Observe como - a diferencia de lo que ocurre en Pascal - la subrutina `submarine` es accesible desde el programa principal:

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

1.15.2. Argumentos y valores de retorno

La Lista de Argumentos `@_`

Los argumentos pasados a una subrutina están disponibles dentro del bloque via el `array` especial `@_`.

Retorno de Valores

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.

Ejemplo

Veamos un ejemplo (el hash `%ENV` contiene las variables de entorno de la ejecución del programa):

```
lhp@nereida:~/Lperl/src$ cat -n return.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use List::Util qw(max);
 4
 5  sub dictionary_order {
 6    my @ordered = sort @_;
 7    return @ordered;
 8  }
 9
```

```
10  sub maxlength {
11    max map { length } @_;
12  }
13
14  # Formatted print of a hash
15  sub fprinth {
16    my %h = @_;
17
18    my @a = &dictionary_order(keys %h);
19    my $max = maxlength(@a);
20    printf("%-${max}s => %s\n",$_,$h{$_}) for @a;
21  }
22
23  # main
24  fprinth(%ENV);
```

El formato `%-${max}s` hace que el argumento se alinee a izquierda y se expanda con blancos hasta `max` espacios.

Observe como los argumentos en la llamada se han puesto entre paréntesis. Al ejecutar el programa anterior obtenemos una salida similar a esta:

```
lhp@nereida:~/Lperl/src$ return.pl
CVSROOT          => /var/cvs
HOME             => /home/lhp
LANG             => es_US.UTF-8
LANGUAGE         => es_ES:es:en_GB:en
LESSCLOSE        => /usr/bin/lesspipe %s %s
....
```

Uso del prefijo `&`

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);
```

Omisión de la Lista de Argumentos en la LLamada

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:

```
lhp@nereida:~/Lperl/src$ cat -n get_next.pl
 1  sub get_next { return <>; }
 2
 3  prompt();  # correcto
 4  $next = get_next(); #correcto
 5  print $next;
 6
```

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

Ejercicio 1.15.1. *¿En que contexto se evalúa la llamada al operador diamante en la línea 1 en el programa anterior?*

Detectando si un Argumento es de Escritura

La función `readonly` en Scalar::Util permite determinar si un argumento es una variable o un valor:

```
lhp@nereida:~/Lperl/src/testing$ cat -n readonly.pl
 1  use warnings;
 2  use strict;
 3  use Carp;
 4  use Scalar::Util qw(readonly);
 5
 6  sub ro {
 7    print "readonly($_[0])=".readonly($_[0])."\n";
 8    $_[0] = 'xxx';
 9  }
10
11  my $x = 4;
12
13  eval { ro($x) } or print $@;
14  eval { ro(-8) } or print $@;
15  eval { ro('lonely') } or print $@;
```

Cuando se ejecuta, el programa anterior produce una salida como:

```
lhp@nereida:~/Lperl/src/testing$ perl readonly.pl
readonly(4)=0
readonly(-8)=8388608
Modification of a read-only value attempted at readonly.pl line 8.
readonly(lonely)=8388608
Modification of a read-only value attempted at readonly.pl line 8.
```

1.15.3. Otros modos de llamar a una subrutina

LLamada sin Paréntesis

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 {
  my $a  = shift;   # shift asume el arreglo @_

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

LLamada con Ampersand y sin Lista de Argumentos

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;
  my $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 importante diferencia entre:

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

y

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

LLamada con Retorno al Padre

Una última variación utiliza el prefijo `&` pero invoca a la subrutina a través 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.

1.15.4. 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 (por ejemplo, escalar) no afecta en absoluto el valor de otro tipo de objeto (por ejemplo, lista) que tenga el mismo nombre. Asi pues 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 código que muestra los usuarios de un sistema Unix que casan con la expresión regular dada como argumento en la línea de comandos:

```
lhp@nereida:~/Lperl/src$ cat -n samenamespace.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  my $user = shift;
 4     $user = ’.*’ unless defined($user);
 5
 6  die "can’t find /etc/passwd\n" unless -r "/etc/passwd";
 7  my @user = grep {/$user/i } ‘cat /etc/passwd‘;
 8
 9  foreach $user (@user) {
10    my ( $name ) = split ’:’, $user, 2;
11    print "$name\n";
12  }
```

La llamada a `shift` de la línea 2 actúa sobre el array especial `@ARGV`. Este array contiene la lista de argumentos pasados en la línea de comandos.

El operador `-r` devuelve `TRUE` si el fichero con ese nombre es de lectura (línea 6).

1.15.5. La declaración `our`

Perl 5.6.0 introdujo la declaración `our` que permite declarar variables de paquete. Un paquete es un espacio de nombres.

Una declaración `package` cambia el ”espacio de nombres” hasta que encontremos una nueva declaración `package`, o hasta el final del bloque actual. El `package` inicial es el package `main`. Cuando sea necesario hacer explícito a que `package` pertenece una variable se prefija su nombre con el del `package`, separado por `::`. Una `$variable` declarada con `our` en el paquete `$nombre::de::paquete` es accesible desde cualquier fichero y desde cualquier paquete vía su nombre completo: `$nombre::de::paquete::variable`. Veamos un ejemplo:

```
package C110;
# estamos en el espacio de nombres C110

our $a = 5;           # variable del paquete C110
fun1              # función del paquete C110
{
  print  "$a\n";
}

package D110;
# ahora estamos en el espacio de nombres D110
# ...salimos del paquete C110

our $a = 7;           # esta $a es del paquete D110
print  $a;        # imprime 7

print  $C110::a;
# imprime 5
# note como podemos acceder el espacio de nombres C110...
# note el $ y los ::

C110::fun1;       # llama a fun1 de C110...imprime: 5
```

Nombre Completo

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.

Visibilidad Léxica de `our`

Una declaración `our $x` hace que `$x` sea visible en el ámbito léxico. Por ejemplo el siguiente programa es considerado sintácticamente incorrecto:

```
nereida:/home/lhp/projects/perl/src# cat -n our2.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3
     4  sub tutu {
     5    $x = 5;
     6    print "$x\n";
     7  }
     8
     9  our $x = 4;
```

```
    10  tutu();
    11  print "$x\n";
```

Al compilar obtenemos errores que indican que `$x` no ha sido declarada antes del uso:

```
nereida:/home/lhp/projects/perl/src# perl -c our2.pl
Global symbol "$x" requires explicit package name at our2.pl line 5.
Global symbol "$x" requires explicit package name at our2.pl line 6.
```

Para convertirlo en un programa correcto debemos declarar `$x` antes de su uso, por ejemplo en la línea 3. Vamos, sin embargo, a hacer algo mas extraño:

```
nereida:/home/lhp/projects/perl/src# cat -n our.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3
     4  sub tutu {
     5    our $x = 5;
     6    print "$x\n";
     7  }
     8
     9  our $x = 4;
    10  tutu();
    11  print "$x\n";
```

Sólo hay una variable `$x` en el paquete actual. La declaración de la línea 5 ”hace visible” `$x` en las líneas 5-7, pero es la misma variable que se declara en la línea 9. El programa produce la salida:

```
nereida:/home/lhp/projects/perl/src# our.pl
5
5
```

Los Accesos con Nombre Completo son Aceptados por el Compilador

Otra forma de reconvertir el programa en sintácticamente correcto es hacer uso del nombre completo de la variable:

```
lhp@nereida:~/projects/perl/src$ cat -n our3.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3
     4  sub tutu {
     5    $main::x = 5;
     6    print "$main::x\n";
     7  }
     8
     9  our $x = 4;
    10  tutu();
    11  print "$x\n";
```

El paquete en el que reside el programa principal se denomina `main::`

```
lhp@nereida:~/projects/perl/src$ perl -c our3.pl
our3.pl syntax OK
```

1.15.6. El uso de `local`

Las variables de paquete se pueden ”localizar” mediante la palabra reservada `local`. El valor de la variable es salvado y recuperado al finalizar el bloque en el que se las ”localiza” mediante `local`. Las variables localizadas mediante `local` son de paquete y por tanto siguen siendo accesibles desde cualquier subrutina. La localidad se refiere al ambito temporal de su valor, no a su visibilidad, que es global. Así, la palabra `local` debe entenderse en el sentido de valor local (`local value`).

Semántica de `local`

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

DECLARACIÓN DE `local`	SIGNIFICADO

DECLARACIÓN DE `local`

```
{
  local($SomeVar);
  $SomeVar = ’My Value’;
      ...
}
```

SIGNIFICADO

```
{
  my $TempCopy = $SomeVar;
  $SomeVar = undef;
  $SomeVar = ’My Value’;
      ...
  $SomeVar = $TempCopy;
}
```

Ejemplo

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

```
lhp@nereida:~/Lperl/src$ cat -n local.pl
#!/usr/bin/perl -w
use strict;

our $x;

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

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

. . . y su ejecución:

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

Localización de las Variables Mágicas

Las variables especiales como `$_` pertenecen al paquete `main::` y pueden ser calificadas con ”local” pero no pueden ser calificadas con `my`.

Localización Automática

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
```

1.15.7. Argumentos con Nombre

En Perl es posible escribir una subrutina que soporte llamada de *argumentos con nombre*. La idea es ”enrollar” los argumentos de la subrutina en un `hash`. Supongamos que queremos escribir una subrutina `listdir` 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`:

```
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.
}
```

1.15.8. `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
```

1.15.9. 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)
```

El Módulo `Contextual::Return`

Se puede obtener una información mas detallada sobre el contexto de la llamada usando el módulo `Contextual::Return`:

```
pl@nereida:~/src/perl/testing$ cat contextual.pl
#!/usr/local/bin/perl -w
use strict;
use Contextual::Return;

sub sensible {
  return STR { "one" }
         NUM { 1 }
  ;
}

print "Result = ".sensible()."\n";
print "Result = ".(0+sensible())."\n";
```

Al ejecutar este programa obtenemos:

```
pl@nereida:~/src/perl/testing$ contextual.pl
Result = one
Result = 1
```

`Contextual::Return` tambien permite crear variables contextuales multitipo con mas de dos tipos (a diferencia de `dualvar` que está restringida a los tipos cadena y número):

```
lhp@nereida:~/Lperl/src/testing$ cat -n context1.pl
 1  #!/usr/local/bin/perl -w
 2  use strict;
 3  use Contextual::Return;
 4
 5  my $x = BOOL { 0 } NUM { 3.14 } STR { "pi" };
 6
 7  unless ($x) { warn "¡El famoso número $x (".(0+$x).") pasa a ser falso!\n" } # executed!
lhp@nereida:~/Lperl/src/testing$ context1.pl
¡El famoso número pi (3.14) pasa a ser falso!
```

Si se usa la etiqueta `ACTIVE` el código de definición será ejecutado cada vez que la variable multiestado es evaluada. Por ejemplo:

```
lhp@nereida:~/Lperl/src/testing$ cat -n context2.pl
1  #!/usr/local/bin/perl -w
2  use strict;
3  use Contextual::Return;
4
5  my $now = ACTIVE NUM { time } STR { localtime };
6
7  print "Now: $now (".($now+0).")\n";
8  sleep(1);
9  print "Now: $now (".($now+0).")\n";
```

La ejecución del programa produce una salida como esta:

```
lhp@nereida:~/Lperl/src/testing$ context2.pl
Now: Sat Mar 15 11:45:58 2008 (1205581558)
Now: Sat Mar 15 11:45:59 2008 (1205581559)
```

1.15.10. ¿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 *paquete* o *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];
```

1.15.11. Práctica: Máximo

Rellene las zonas de puntos en el esqueleto de programa que figura a continuación. La función `max` deberá calcular el máximo o el mínimo de los elementos de una lista. El primer argumento de la función es una cadena conteniendo el operador de comparación a usar (por ejemplo `<` o `>` o `gt` o `lt`). Utilice la función `eval` la cuál recibe una cadena y evalúa la expresión Perl que representa.

```
lhp@nereida:˜/Lperl/src$ cat -n max.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3
     4  sub max {
     5    my $op = (shift || "<");
     6
     7    return undef unless ((@_) and $op =˜ m{([<>]=?)|(gt)|(lt)});
     8    my $m = shift;
     9    my $comparison = ’$m ’.$op.’ $x ’;
    10
    11    for my $x (@_) {
    12      $m = $x if ....;
    13    }
    14    $m;
    15  }
    16
    17  my $m1 = max("<", 5, 4, 9, 1, 0);
    18  my $m2 = max(">",-1, -5, 0, 13, 12, 7);
    19  my $m3 = max("lt",-1, "b", 0, 13, 12, 7, "a");
    20  my $m4 = max();
    21
    22  print "$m1, $m2, $m3\n";
    23
    24  print "$m4\n" if defined($m4);
```

Comente en su respuesta el significado de las líneas 5 y 7. Documente el código. Sigue un ejemplo de ejecución:

```
lhp@nereida:~/Lperl/src$ ./max.pl
9, -5, b
lhp@nereida:~/Lperl/src$
```

1.15.12. 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.

1.15.13. 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:

```
lhp@nereida:~/Lperl/src$ perl -wde 0
main::(-e:1):   0
  DB<1> $a = 4,5,6
  DB<2> x $a
0  4
  DB<3> $a = (4,5,6)
  DB<4> p $a
6
  DB<5> @a = (4,5,6)
  DB<6> $a = @a
  DB<7> p $a
3
  DB<8> $a = print (4,3),7
```

```
43
  DB<9> p $a
1
  DB<10> $x = ($a = print (4,3),7)
43
  DB<11> p $x
7
```

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

1.15.14. 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); $angle = atan2($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.

Son ejemplos de llamadas legales:

```
p2c(x=>1, r=>1)
p2c(x=>1, y => 0)
p2c(r =>1, angle => 0.2)
```

La función devolverá un hash con los valores de `x`, `y`, `r` y `angle`.

Se considerará un error llamar a la función con un número de argumentos distinto de dos.

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:

```
pp2@nereida:~/pp2bin$ perl -MMath::Trig -wde 0
main::(-e:1):   0
  DB<1> p atan2(0, -1)
3.14159265358979
  DB<2> $pi = atan2(0, -1)
  DB<3> p tan($pi/4)
1
```

La llamada al intérprete `perl` con la opción `-MMath::Trig` carga el módulo `Math::Trig` de la misma manera que lo hace el pragma `use`.

1.15.15. Práctica: Postfijo y Subrutina

Reescriba la práctica de la calculadora en posfijo realizada en la sección 1.13.8 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 3.2 y 3.12. Una alternativa es el operador `splice` (sección 1.13.2).

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

Capítulo 2

Entrada /Salida

2.1. El Operador Diamante y el Manejador de Ficheros `ARGV`

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 leerá desde la entrada estándar.

Se puede usar el guión corto como sinónimo de leer desde la entrada estándar. El siguiente programa muestra los ficheros proveidos en la línea de comandos. Enumera las líneas y justifica los números de línea a la derecha según el valor de `$width`:

```
lhp@nereida:~/Lperl/src$ cat -n diamond.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use POSIX qw(ceil);
 4  use List::Util qw(max);
 5  my $width = shift if (@ARGV) and $ARGV[0] =~ m{^\d+$};
 6  $width = 2 unless $width;
 7  my @lines = map { (split /\s+/, ‘wc $_‘)[1] } @ARGV;
 8  @lines = map { ceil(log($_)) } @lines;
 9  $width = max( $width, @lines);
10  my $n = 1;
11  my $OLDARGV = "";
12  while (<>) {
13    if ($OLDARGV ne $ARGV) {
14      printf("FILE $ARGV\n");
15      $OLDARGV = $ARGV;
16      $n = 1;
17    }
18    printf("%${width}d %s", $n++, $_);
19  }
```

La variable `$ARGV` contiene el nombre del fichero actual. Veamos varias ejecuciones del programa:

```
lhp@nereida:~/Lperl/src$ diamond.pl 12 test.txt
FILE test.txt
           1 Welcome ...

lhp@nereida:~/Lperl/src$ diamond.pl test.txt which.pl
FILE test.txt
```

```
   1 Welcome ...
FILE which.pl
   1 #!/usr/bin/perl -w
   2
   3 $pattern = shift || ’’;
   4 $pattern =~ s{::}{/}g;
   5 #$pattern =~ s{$}{.pm};
   6 @dirs = @INC;
   7 #push @dirs, qw(
   8 #    /etc/perl
   9 #    /usr/local/lib/perl/5.8.4
  10 #    /usr/local/share/perl/5.8.4
  11 #    /usr/lib/perl5
  12 #    /usr/share/perl5
  13 #    /usr/lib/perl/5.8
  14 #    /usr/share/perl/5.8
  15 #    /usr/local/lib/site_perl
  16 #);
  17
  18 for (@dirs) {
  19   my $file = $_."/".$pattern.’.pm’;
  20   print "$file\n" if (-e $file);
  21   $file = $_."/".$pattern.’.pod’;
  22   print "$file\n" if (-e $file);
  23   $file = $_."/pod/".$pattern.’.pod’;
  24   print "$file\n" if (-e $file);
  25 }
```

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.

```
lhp@nereida:~/Lperl/src$ cat -n diamond2.pl
     1  #!/usr/local/bin/perl -w
     2  use strict;
     3  my $n;
     4  @ARGV = qw#dummy.bk.1 dummy.bk.2#;
     5  while (<>) {
     6    print $n++." $_";
     7  }
```

Veamos un ejemplo de ejecución:

```
lhp@nereida:~/Lperl/src$ diamond.pl dummy.bk.1 dummy.bk.2
FILE dummy.bk.1
 1 ABCDEF
 2 ABCDEF
 3 ABCDEF
FILE dummy.bk.2
 1 ABCDEF
 2 ABCDEF
 3 ABCDEF
lhp@nereida:~/Lperl/src$ diamond2.pl 34 45 gethostbyname.pl
```

```
0 ABCDEF
1 ABCDEF
2 ABCDEF
3 ABCDEF
4 ABCDEF
5 ABCDEF
```

El manejador de ficheros especial usado por el operador diamante es conocido como `ARGV`. La expresión `<>` es un sinónimo de `<ARGV>`.

2.2. El manejador `ARGVOUT`

Si se usa la variable especial `$^ I` se asume que los elementos de `@ARGV` son ficheros. Los ficheros son copiados al sufijo indicado en `$^I`. La salida estándar pasa a estar en los ficheros de `@ARGV`. Este manejador de ficheros es conocido como `ARGVOUT`.

Vea el siguiente ejemplo:

```
lhp@nereida:~/Lperl/src$ cat -n argvout.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3
 4  $^I=".bk";
 5
 6  while (<>) {
 7    tr/a-z/A-Z/;
 8    print;          # a ARGVOUT,
 9  }
```

Al ejecutar el programa anterior tenemos:

```
lhp@nereida:~/Lperl/src$ diamond.pl dummy.bk.1 dummy.bk.2
FILE dummy.bk.1
 1 abcdef
 2 abcdef
 3 abcdef
FILE dummy.bk.2
 1 abcdef
 2 abcdef
 3 abcdef
lhp@nereida:~/Lperl/src$ argvout.pl dummy.bk.1 dummy.bk.2
lhp@nereida:~/Lperl/src$ diamond.pl dummy.bk.1 dummy.bk.2
FILE dummy.bk.1
 1 ABCDEF
 2 ABCDEF
 3 ABCDEF
FILE dummy.bk.2
 1 ABCDEF
 2 ABCDEF
 3 ABCDEF
```

2.3. Uso de Perl desde la Línea de Comandos: 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`.

Otro ejemplo: la sustitución que sigue ocurre en todos los ficheros `info.txt` en todos los subdirectorios de los subdirectorios que comiencen por `alu`:

```
perl -e 's/\|hyperpage//gi' -p -i.bak  alu*/*/info.txt
```

Las *opciones de línea* de comandos significan lo siguiente:

-e puede usarse para definir el script en la línea de comandos. Multiples -e te permiten escribir un multi-script. Cuando se usa -e, perl no busca por un fichero de script entre la lista de argumentos.

-p La opción `-p` hace que perl incluya un bucle alrededor de tu "script" al estilo sed:

```
while (<>) {
        ...             # your script goes here
} continue {
        print;
}
```

El siguiente fragmento de la documentación e `perlrun` detalla la semántica de `-p` y `-i`:

```
From the shell, saying

  $ perl -p -i.orig -e "s/foo/bar/; ... "

is the same as using the program:

  #!/usr/bin/perl -pi.orig
  s/foo/bar/;

which is equivalent to

  #!/usr/bin/perl
  $extension = '.orig';
  LINE: while (<>) {
      if ($ARGV ne $oldargv) {
          if ($extension !~ /\*/) {
              $backup = $ARGV . $extension;
          }
          else {
              ($backup = $extension) =~ s/\*/$ARGV/g;
          }
          rename($ARGV, $backup);
          open(ARGVOUT, ">$ARGV");
          select(ARGVOUT);
          $oldargv = $ARGV;
      }
      s/foo/bar/;
  }
  continue {
      print;  # this prints to original filename
  }
```

```
    select(STDOUT);

  except that the -i form doesn't need to compare $ARGV to $oldargv to know when
  the filename has changed.  It does, however, use ARGVOUT for the selected
  filehandle.  Note that STDOUT is restored as the default output filehandle
  after the loop.
```

-n Nótese que las líneas se imprimen automáticamente. Para suprimir la impresión usa la opción `-n`

-i[ext] La opción `-i` Expresa que los ficheros procesados serán modificados. Se renombra el fichero de entrada `file.in` a `file.in.ext`, abriendo el de salida con el mismo nombre del fichero de entrada `file.in`.

Se selecciona dicho fichero como de salida por defecto para las sentencias `print`. Si se proporciona una extensión se hace una copia de seguridad. Si no, no se hace copia de seguridad.

En general las opciones pueden ponerse en la primera línea del “script”, donde se indica el intérprete. Asi pues, decir
`perl -p -i.bak -e "s/foo/bar/;"`
es equivalente a usar el “script”:

```
#!/usr/bin/perl -pi.bak
s/foo/bar/;
```

Ejercicio 2.3.1. *Lea* `perldoc perlrun`

Ejercicio 2.3.2. *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;
```

Ejercicio 2.3.3. *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 constituyen los elementos del `array x`.

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

```
print sort <>;
```

2.4. El Manejador de Ficheros `DATA`

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__`.

2.5. Operaciones sobre Ficheros

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`.

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 (`>`).

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.

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.

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"; }
}
```

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
```

Ejercicio 2.5.1. *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* `\$_`.

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
```

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
```

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
***
```

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: $!";
```

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.

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

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

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

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 $_
}
```

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 (*Midnight 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 `$_`.

La Función `openhandle`

La función `openhandle` en Scalar::Util permite saber si una expresión es un manejador de fichero (o un atado mediante `tie` de un manejador).

```
     openhandle FH
```

Retorna `FH` si `FH` puede ser usado como manejador y esta abierto. En otro caso retorna `undef`:

```
 $fh = openhandle(*STDIN);           # \*STDIN
 $fh = openhandle(\*STDIN);          # \*STDIN
 $fh = openhandle(*NOTOPEN);         # undef
 $fh = openhandle("scalar");         # undef
```

2.6. 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`.

2.7. Ficheros Binarios

La función read La función `read FILEHANDLE,SCALAR,LENGTH,OFFSET` lee desde el fichero `FILEHANDLE` un número de `LENGTH` caracteres en la variable `SCALAR`. Devuelve el número de caracteres leídos, 0 si es el final de fichero y `undef` si hubo un error. Es posible indicar que el fichero usado es binario mediante la función `binmode` (línea 7 en el ejemplo que sigue):

```
lhp@nereida:~/Lperl/src$ cat -n binaryfiles.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  my $buffer = "";
 4  my $file = shift;
 5
 6  open(FILE, "< $file");
 7  binmode(FILE);
 8  my $s = -s $file;
 9  read(FILE, $buffer, $s);
10  close(FILE);
11
12  open(FILE, "> $file.sal");
13  syswrite(FILE, $buffer, $s);
14  close(FILE);
15
16  my $c;
17  foreach (split(//, $buffer)) {
18    printf("%02x ", ord($_));
19    print "\n" unless ++$c % 20;
20  }
21  print "\n";
```

La salida de la ejecución nos permite comprobar que la copia de un ejecutable `a.out` preserva el formato binario:

```
lhp@nereida:~/Lperl/src$ ./binaryfiles.pl a.out | head -3
7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 02 00 03 00
01 00 00 00 e0 82 04 08 34 00 00 00 f4 0d 00 00 00 00 00 00
34 00 20 00 07 00 28 00 22 00 1f 00 06 00 00 00 34 00 00 00
lhp@nereida:~/Lperl/src$ ls -ltr | tail -2
-rwxr-xr-x   1 lhp lhp   341 2006-06-21 18:00 binaryfiles.pl
-rw-r--r--   1 lhp lhp  6969 2006-06-21 18:02 a.out.sal
lhp@nereida:~/Lperl/src$ chmod a+x a.out.sal; ./a.out.sal
hello world!
lhp@nereida:~/Lperl/src$ ./a.out
hello world!
lhp@nereida:~/Lperl/src$
```

La función `syswrite`

La función `syswrite` (línea 13) tiene el siguiente formato:

```
syswrite FILEHANDLE,SCALAR,LENGTH,OFFSET
syswrite FILEHANDLE,SCALAR,LENGTH
syswrite FILEHANDLE,SCALAR
```

Intenta escribir `LENGTH` bytes de `SCALAR` en `FILEHANDLE` saltándose los buffers de E/S. Si no se especifica `LENGTH` se escribe todo `SCALAR`.

2.8. La función `localtime`

`localtime` en contexto de Lista
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'
```

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

```
lhp@nereida:~$ perl -wde 0
main::(-e:1):   0
  DB<1> p localtime
45307331084931
  DB<2> p scalar(localtime)
Thu Apr  3 07:30:54 2008
```

La función `gmtime`

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

Ejercicio 2.8.2. *Usando el depurador de Perl calcule que día del calendario fué hace 500 días.*

El Módulo `Time::HiRes`

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
```

2.9. Directorios

El Operador `chdir`

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

```
lhp@nereida:~/Lperl/src/dir$ cat -n chdir.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3
     4  my $directory = shift;
     5  my $expreg = (shift or ’.*’);
     6  $expreg = qr/$expreg/o;
     7  chdir $directory or die "no encuentro $directory: $!";
     8  my @files = ‘ls -a‘;
     9
    10  for (@files) {
    11    next unless /$expreg/;
    12    chop;
    13    printf "$_\n";
    14  }
```

Sigue un ejemplo de ejecución:

```
lhp@nereida:~/Lperl/src/dir$ chdir.pl /tmp/ ’\d+$’
IO-Tty-1.02
Parallel-Simple-Pipe-0.01
ssh-eSbkNq1402
v274545
```

Obtener el directorio actual

Use el módulo Cwd:

```
use Cwd;
my $dir = getcwd;

use Cwd ’abs_path’;
my $abs_path = abs_path($file);
```

Obtener el Camino Absoluto de un Fichero

Utilice la función `abs_path` en Cwd para conocer el camino absoluto de un fichero o directorio:

```
pp2@nereida:~/src/perl/testing$ perl -wde 0
  DB<1> use Cwd ’abs_path’
  DB<2> x abs_path(’add.pl’)
0  ’/home/pp2/src/perl/testing/add.pl’
  DB<3> x abs_path(’./add.pl’)
0  ’/home/pp2/src/perl/testing/add.pl’
  DB<4> x abs_path(’/home/pp2/src/perl/testing/add.pl’)
0  ’/home/pp2/src/perl/testing/add.pl’
  DB<5> x abs_path(’../testing/add.pl’)
0  ’/home/pp2/src/perl/testing/add.pl’
  DB<6> x abs_path(’../testing/’) # También con directorios
0  ’/home/pp2/src/perl/testing’
```

Los operadores `opendir`, `readdir` y `closedir`

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

```
lhp@nereida:~/Lperl/src/dir$ cat -n dirhandles.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3
     4  my $directory = shift;
     5  my $regexp = (shift or '.*');
     6  opendir DH, $directory or die "No encuentro $directory: $!";
     7
     8  foreach my $file (readdir DH) {
     9    printf "$file\n" if $file =~ m/$regexp/o;
    10  }
    11  closedir DH;
```

Sigue un ejemplo de ejecución:

```
lhp@nereida:~/Lperl/src/dir$ dirhandles.pl /tmp/ '\d+$'
ssh-eSbkNq1402
v274545
Parallel-Simple-Pipe-0.01
IO-Tty-1.02
```

- 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.

Lista Resumida de Operadores de Directorios

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

`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

Los módulos `File::Basename` y `File::Spec`

Los módulos `File::Basename` y `File::Spec` permiten el manejo de nombres de fichero de una manera portable. Veamos por ejemplo la función `catfile`:

```
lhp@nereida:~/Lperl/src/perl_networking/ch2$ perl -de 0
  DB<1> use File::Spec::Functions;
  DB<2> $x = catfile('a', 'b', 'c')
  DB<3> x $x
0  'a/b/c'
```

En un sistema windows `$x` contendría `'a\b\c'`. Para saber mas, lea 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`.

Sigue un ejemplo de uso de la función `fileparse`:

```
  DB<1> use File::Basename
  DB<2> @a = fileparse("/foo/bar/baz", qr/\.[^.]*/)
  DB<3> x @a
0  'baz'
1  '/foo/bar/'
2  ''
  DB<4> @a = fileparse("/foo/bar/baz.pm", qr/\.[^.]*/)
  DB<5> x @a
0  'baz'
1  '/foo/bar/'
2  '.pm'
```

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 = glob ".* *";
```

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

Lectura desde Globs

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";
```

Lectura de Directorios en Contexto de Lista

```
lhp@nereida:~/alu/0607$ perl -wde 0
main::(-e:1):   0
  DB<1> use List::Util qw(sum)
  DB<2> p sum(map { -s $_ } <alu*>)
163840
```

En este código, la lectura `<alu*>` se produce en un contexto de lista. Por ello el directorio completo es leído y produce una lista con todos los nombres de ficheros y directorios `alu*` existentes en el directorio actual. El subsecuente `map` produce los tamaños y la llamada a `sum` devuelve el tamaño total.

Contexto Booleano: Bucles con Lecturas de Directorios

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 debería tener una ”muerte” prematura. La codificación correcta sería:

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

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
```

Esta modificación de la evaluación ”normal” de una expresión en un contexto lógico/booleano no es (del todo) privilegio exclusivo de Perl. El programador puede conseguir un efecto similar usando el módulo `Contextual::Return` de Damian Conway, el cual permite crear variables contextuales multitipo con mas de dos tipos:

```
lhp@nereida:~/Lperl/src/testing$ cat -n context1.pl
 1  #!/usr/local/bin/perl -w
 2  use strict;
 3  use Contextual::Return;
 4
 5  my $x = BOOL { 0 } NUM { 3.14 } STR { "pi" };
 6
 7  unless ($x) { warn "¡El famoso número $x (".(0+$x).") pasa a ser falso!\n" } # executed!
lhp@nereida:~/Lperl/src/testing$ context1.pl
¡El famoso número pi (3.14) pasa a ser falso!
```

2.10. Operaciones con ficheros, links y directorios

La función `unlink`

Use el operador `unlink` para suprimir ficheros:

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

Devuelve el número de ficheros suprimidos.

La función `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: $!";
```

La función `mkdir`

Para crear un directorio use `mkdir`:

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

Crear y Suprimir Directorios Recursivamente: El Módulo `File::Path`

El módulo File::Path provee un medio conveniente para la creación y destrucción recursiva de directorios. El siguiente ejemplo usa la versión 2.04 del módulo:

```
pp2@nereida:~/src/perl/testing$ perl -MFile::Path -wde 0
Loading DB routines from perl5db.pl version 1.28
main::(-e:1):   0
  DB<1>  mkpath( ’/tmp/chum/cham/’, ’/tmp/chum/chim’, {verbose => 1} )
mkdir /tmp/chum
mkdir /tmp/chum/cham/
mkdir /tmp/chum/chim
  DB<2> !! ls -ld /tmp/chum /tmp/chum/cham/ /tmp/chum/chim
drwxr-xr-x 4 pp2 pp2 4096 2008-04-23 11:39 /tmp/chum
drwxr-xr-x 2 pp2 pp2 4096 2008-04-23 11:39 /tmp/chum/cham/
drwxr-xr-x 2 pp2 pp2 4096 2008-04-23 11:39 /tmp/chum/chim
  DB<3>  rmtree(’/tmp/chum/cham/’, ’/tmp/chum/chim’, { verbose => 1, error  => \$err_list })
```

```
rmdir /tmp/chum/cham
rmdir /tmp/chum/chim
  DB<4> x $err_list
0  ARRAY(0x84b6178)
     empty array
  DB<5> !! ls -ld /tmp/chum /tmp/chum/cham/ /tmp/chum/chim
ls: /tmp/chum/cham/: No existe el fichero o el directorio
ls: /tmp/chum/chim: No existe el fichero o el directorio
drwxr-xr-x 2 pp2 pp2 4096 2008-04-23 11:42 /tmp/chum
(Command exited 2)
  DB<6>  mkpath( '/tmp/chum/cham/', '/tmp/chum/chim', {verbose => 1} )
mkdir /tmp/chum/cham/
mkdir /tmp/chum/chim

  DB<7> !! touch /tmp/chum/cham/tutu.txt
  DB<8> !! touch /tmp/chum/chim/titi.txt
  DB<9> x rmtree('/tmp/chum/cham/', '/tmp/chum/chim', { verbose => 1, error  => \$err_list })
unlink /tmp/chum/cham/tutu.txt
rmdir /tmp/chum/cham
unlink /tmp/chum/chim/titi.txt
rmdir /tmp/chum/chim
0  4
```

La función `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.

La función `rmdir`

Utilice la función `rmdir` para suprimir directorios:

```
rmdir "tutu/";
```

La función `chmod`

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

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

La función `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:

```
lhp@nereida:/tmp$ perl -wde 0
  DB<1> @a = getpwnam('elvira')
  DB<2> x @a
0  'elvira'
1  'x'
2  1033
```

```
3  1033
4  ’’
5  ’’
6  ’Elvira,,,’
7  ’/home/elvira’
8  ’/bin/bash’
  DB<3> $a = getpwnam(’elvira’)
  DB<4> x $a
0  1033
  DB<5>  x getgrnam "users"
0  ’users’
1  ’x’
2  100
3  ’’
  DB<6>  x scalar(getgrnam "users")
0  100
```

2.11. Renombrar Ficheros

El comando `rename` es un estandard Unix que permite renombrar múltiples ficheros. Veamos algunos ejemplos de uso de rename:

```
rename ’s/\.bak$//’ *.bak
rename ’y/A-Z/a-z/’ *
rename ’s/\.flip$/.flop/’       # rename *.flip to *.flop
rename s/flip/flop/             # rename *flip* to *flop*
rename ’s/ˆs\.(.*)/$1.X/’
rename ’s/$/.orig/ */*.[ch]’
rename ’y/A-Z/a-z/’
rename ’y/A-Z/a-z/ if -B’       # lo mismo pero con binarios
rename ’s{(\d+)}{"-".(2*$1)}e’ node*.html
```

Procesado de los Argumentos

Veamos el procesado de la línea de comandos:

```
lhp@nereida:˜/Lperl/src/cookbook/ch16$ cat -n /usr/bin/rename
 1  #!/usr/bin/perl -w
..  ... # comments
32  use strict;
33
34  use Getopt::Long;
35  Getopt::Long::Configure(’bundling’);
36
37  my ($verbose, $no_act, $force, $op);
38
39  die "Usage: rename [-v] [-n] [-f] perlexpr [filenames]\n"
40      unless GetOptions(
41          ’v|verbose’ => \$verbose,
42          ’n|no-act’  => \$no_act,
43          ’f|force’   => \$force,
44      ) and $op = shift;
```

El uso de `GetOptions` significa que podemos llamar al programa de la forma:

```
$ rename -n 's/1628/chuchu/' node-1628.html
node-1628.html renamed as node-chuchu.html
$ ls -l *chuchu*
ls: *chuchu*: No existe el fichero o el directorio
```

La opción `-n` hace que la ejecución sea ”simulada”

```
$ rename -no 's/1628/chuchu/' node-1628.html
Unknown option: o
Usage: rename [-v] [-n] [-f] perlexpr [filenames]
```

La llamada `Getopt::Long::Configure('bundling')`hace que sea posible juntar varias opciones. Por ejemplo, si `a`, `v` y `x` son opciones válidas, entonces un agrupamiento como `-vax` activa las tres.

Como se ha usado bundling, `-no` es sinónimo de `-n -o`. Si se quieren usar prefijos o nombres completos se debe usar el formato largo con dos guiones:

```
$ rename --no-act 's/1628/chuchu/' node-1628.html
node-1628.html renamed as node-chuchu.html
```

La presencia de la opción `n` o de `--no-act` hace que la variable `$no_act` se inicia a verdadero.

La función `GetOptions` retorna falso si se produjo un error procesando la línea de comandos. Los argumentos no procesados, aquellos que no se corresponden a las opciones descritas en la llamada permancen en `@ARGV`.

Véase la documentación del módulo Getopt::Long para los detalles.

Si `$no_act` se adopta el modo `$verbose`:

```
46  $verbose++ if $no_act;
```

Si no se proveen ficheros se leen desde `STDIN`:

```
48  if (!@ARGV) {
49      print "reading filenames from STDIN\n" if $verbose;
50      @ARGV = <STDIN>;
51      chop(@ARGV);
52  }
```

Evaluación de la Expresión de Sustitución

El programa pasado como argumento es evaluado en la línea 56. Si contiene errores `$@` contendrá el mensaje de error (línea 57).

```
54  for (@ARGV) {
55      my $was = $_; # viejo nombre del fichero
56      eval $op;     # sustitución efectuada
57      die $@ if $@; # alto si la expresión es inválida
```

Si el nuevo nombre es igual al viejo no se hacen cambios:

```
58      next if $was eq $_; # ignore quietly
```

A menos que se especificara `--force` no se renombra el fichero si existe ya uno con ese nombre:

```
59      if (-e $_ and !$force) # existe fichero con el
60      {                      # nuevo nombre
61          warn  "$was not renamed: $_ already exists\n";
62      }
```

La evaluación en cortocircuito hace que si se especifico la opción `--no-act` no se ejecute `rename`:

```
63      elsif ($no_act or rename($was, $_)) # Se renombran mediante "rename"
64      {
65          print "$was renamed as $_\n" if $verbose;
66      }
67      else
68      {
69          warn  "Can’t rename $was $_: $!\n";
70      }
71  }
```

Si `$no_act` es verdadero nunca se evalúa `rename($was, $_)` y se pasa a ejecutar la línea 65.

El Código

Sigue el código completo:

```
pp2@europa:~$ sed -ne ’32,71p’ ‘which rename‘ | cat -n
 1  use strict;
 2
 3  use Getopt::Long;
 4  Getopt::Long::Configure(’bundling’);
 5
 6  my ($verbose, $no_act, $force, $op);
 7
 8  die "Usage: rename [-v] [-n] [-f] perlexpr [filenames]\n"
 9      unless GetOptions(
10          ’v|verbose’ => \$verbose,
11          ’n|no-act’  => \$no_act,
12          ’f|force’   => \$force,
13      ) and $op = shift;
14
15  $verbose++ if $no_act;
16
17  if (!@ARGV) {
18      print "reading filenames from STDIN\n" if $verbose;
19      @ARGV = <STDIN>;
20      chop(@ARGV);
21  }
22
23  for (@ARGV) {
24      my $was = $_;
25      eval $op;
26      die $@ if $@;
27      next if $was eq $_; # ignore quietly
28      if (-e $_ and !$force)
29      {
30          warn  "$was not renamed: $_ already exists\n";
31      }
32      elsif ($no_act or rename $was, $_)
33      {
34          print "$was renamed as $_\n" if $verbose;
35      }
36      else
37      {
38          warn  "Can’t rename $was $_: $!\n";
```

```
    }
}
```

2.12. Práctica: Descenso Recursivo en Subdirectorios

Usando el módulo File::Find escriba un programa que muestre los ficheros en el directorio actual mayores que un cierto tamaño dado y cuyo tiempo de modificación es mayor que un cierto número de días.

File::Find

El módulo File::Find provee la función `find`. La sintáxis es:

```
find(\&wanted,  @directories);
```

`find()` hace un recorrido primero profundo de los directorios especificados en `@directories`. Para cada fichero o directorio encontrado se llama a la subrutina `&wanted`. Además, para cada directorio que se encuentre se cambia `chdir()` al directorio y se continúa la búsqueda, invocando a la función `&wanted`.

La función `wanted` es lo que se denomina un callback (Callback_(computer_science) En ella el programador escribe lo que quiere hacer con el fichero o directorio: imprimirlo, renombrarlo, borrarlo, etc. La función será llamada sin argumentos y su valor de retorno será ignorado.

Cuando se la llama:

- `$File::Find::dir` es el directorio actual
- `$_` contiene el nombre del fichero actual
- `$File::Find::name` es el nombre completo del fichero

Estas variables son automáticamente localizadas por `find`.

Veamos un ejemplo de uso:

```
lhp@europa:~/Lperl/src/perltesting/structinline$ perl -wde 0
main::(-e:1):   0
DB<1> !!tree -sD '.'
.
|-- [       4096 Feb 13  8:18]  _Inline
|   |-- [       4096 Feb 13  8:18]  build
|   |-- [        390 Feb 13  8:18]  config
|   `-- [       4096 Feb 13  8:18]  lib
|       `-- [       4096 Feb 13  8:18]  auto
|           `-- [       4096 Feb 13  8:18]  example_pl_7eb7
|               |-- [          0 Feb 13  8:18]  example_pl_7eb7.bs
|               |-- [        584 Feb 13  8:18]  example_pl_7eb7.inl
|               `-- [      30969 Feb 13  8:18]  example_pl_7eb7.so
|-- [        778 Feb 13  8:18]  all.t
|-- [        758 Feb 13  8:18]  anon.t
`-- [        750 Feb 13  8:18]  example.pl

5 directories, 7 files
DB<2> use File::Find
DB<3> sub callback { return unless -f $_; push @large, $File::Find::name if -s $_ > 30000 }
DB<4> find(\&callback, '.')
DB<5> x @large
0  './_Inline/lib/auto/example_pl_7eb7/example_pl_7eb7.so'
1  './_Inline/lib/auto/example_pl_7eb7/.svn/text-base/example_pl_7eb7.so.svn-base'
```

El ejemplo encuentra un fichero adicional al mostrado por `tree` que se encuentra en un directorio escondido (`.svn`).

Referencias

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
```

Getopt::Long

Utilice Getopt::Long para leer las opciones en la línea de comandos. Deberá admitir las llamadas:

```
grandes -s size -m dias directorios
grandes -v -s  size -m dias directorios # modo verbose
grandes -h                              # ayuda
grandes -V                              # version, autor
```

En modo verbose el programa informa de cada directorio que está procesando.

Pod::Usage

Use Pod::Usage para escribir la ayuda.

Por ejemplo, la llamada:

```
pod2usage( -verbose => 1,);
```

hace que se muestren las secciones `SYNOPSIS` así como cualesquiera secciones (esto es, definidas con `=head1`) tituladas `OPTIONS`, `ARGUMENTS`, o `OPTIONS AND ARGUMENTS`.

Véase también

- Lea los artículos de Randall Schwartz (Merlin)
    - http://www.stonehenge.com/merlyn/UnixReview/col43.html Finding old things (Unix Review Column 43 Sep 2002),
    - http://www.stonehenge.com/merlyn/LinuxMag/col45.html Finding things (Linux Magazine Column 45 Feb 2003) y
    - http://www.stonehenge.com/merlyn/LinuxMag/col46.html Finding things by rule (Linux Magazine Column 46 Mar 2003).
- Consulte también el módulo File::Find::Rule Vea un ejemplo de uso de dicho módulo:

```
lhp@nereida:~/Lperl/src$ perl -wde 0
main::(-e:1):   0
  DB<1> use File::Find::Rule
  DB<2> @subdirs = File::Find::Rule->directory->name('perl*')->in('.')
  DB<3> p 0+@subdirs
7
  DB<4> @progs = File::Find::Rule->file->name('*.pl')->size('>10Ki')->in('.')
  DB<5> p 0+@progs
27
```

```
  DB<6> x $progs[0]
0  'kpad.pl'
  DB<7> !!ls -l kpad.pl
-rwxr-xr-x 1 lhp lhp 12881 2006-06-15 12:52 kpad.pl
```

Capítulo 3

Expresiones Regulares

3.1. 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;
}

```

3.2. 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.

3.3. 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`.

3.4. 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`).

3.5. 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`.

3.6. 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]`

3.7. 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
```

3.8. `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>
```

3.9. 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

3.10. 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
```

3.11. 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 3.20 para ver los contenidos del fichero `double.in`. En dicho fichero, el patrón `three.*three` aparece repartido entre varias líneas.

3.12. 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
```

3.13. 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");
}
```

3.14. 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/) {
```

3.15. 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>)
```

3.16. 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.
```

3.17. Algunas extensiones

3.17.1. Comentarios

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

3.17.2. 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
```

3.17.3. 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 3.15.

3.17.4. 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 $‘ !~ /foo$/
```

3.18. Secuencias de números de tamaño fijo

El siguiente problema y sus soluciones se describen en el libro de J.E.F. Friedl [5]. 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.

3.19. 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 3.18:

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

3.20. 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
```

3.21. 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
```

3.22. 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
```

3.23. 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
```

3.24. 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
```

3.25. 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
```

3.26. 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/;
```

3.27. 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
```

3.28. 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 4

Referencias

4.1. Referencias a variables ya existentes

4.1.1. 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
```

4.1.2. 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:

```
main::(-e:1):   0
DB<1> $ra = \10
DB<2> p $$ra
10
DB<3> p $ra
SCALAR(0xc0b840)
```

```
DB<4> x $ra
0  SCALAR(0xc0b840)
   -> 10
DB<5> $$ra = 20
Modification of a read-only value attempted at (eval 18)[/usr/share/perl/5.8/perl5db.pl:628] line 2.
DB<6> use Scalar::Util qw{readonly}
DB<7> x readonly($$ra)
0  8388608
DB<8> x readonly($ra)
0  0
DB<9> @b = 5..10
DB<10> $rb = \@b
DB<11> print "rb = $rb; rb-> = @{$rb}"
rb = ARRAY(0xc5dfc0); rb-> = 5 6 7 8 9 10
```

Observe la forma en la que se imprime una referencia.

4.1.3. Contextos y Referencias

El Operador \ en un Contexto Escalar

Un aviso sobre los contextos en 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.

El Operador \ en un Contexto de Lista

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.

El operador \ y el Aplanamiento de Listas

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
```

Primero \ actúa sobre la sublista. Luego las sublistas 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
```

4.1.4. Ambiguedad en el De-referenciado

¿Como se debe interpretar la expresión `$$a[1]`? ¿Cómo `${$a[1]}` o bien como `${$a}[1]`? ¿Se está hablando de un array `@a` cuyo elemento `$a[1]` es una referencia a un escalar o es `$a` una referencia a una lista `@$a` cuyo segundo elemento esta siendo accedido?

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 $$a[1]
SCALAR(0x81046f4)
  DB<5> p @a
SCALAR(0x81046d0)SCALAR(0x81046f4)SCALAR(0x81046ac)
  DB<6> p ${$a[1]}
5
  DB<7> p ${$a}[1]                      # $$a[1] es ${$a}[1]
SCALAR(0x81046f4)
```

Regla de Evaluación de los Prefijos

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*.

El Significado de las LLaves

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
```

4.1.5. 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 *flecha* `->` 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->[2]}"
0 0 1
  DB<9> p $rm->[1,2]
ARRAY(0x810479c)
  DB<10> p $rm->[1]->[1]
1
  DB<11> p $rm->[1][1] # La segunda flecha puede omitirse
  1
```

¿Puede explicar la conducta observada en la línea 9 (`p $rm->[1,2]`)?

La razón es que el operador de flecha fuerza un contexto escalar en la evaluación del índice. 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);
```

4.2. 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`.

`ref` sobre Referencias a Objetos

Sobre referencias a objetos (véase 6), `ref` devuelve el nombre de la clase:

```
$obj  = bless {}, "Foo";
$type = reftype $obj;    # "Foo"
```

`ref` y la Coomprobación de Tipos

La función `ref` puede ser utilizada para mejorar los mensajes de error:

```
die "Expected scalar reference" unless ref($slr_ref) eq "SCALAR";
```

Evaluación de `ref` en un Contexto de Cadena

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)
```

La Función `reftype`

La función `reftype` de Scalar::Util funciona de manera parecida a `ref EXPR`, con la diferencia de que sobre los objetos no retorna la clase del objeto:

```
$type = reftype "string";           # undef
$type = reftype \$var;              # SCALAR
$type = reftype [];                 # ARRAY

$obj  = bless {}, "Foo";
$type = reftype $obj;               # HASH
```

4.3. Paso de Listas 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 esta limitación consiste en pasar como argumentos las referencias a los arrays y los hashes implicados.

El siguiente ejemplo muestra una subrutina que permite realizar la suma, el producto, etc. de un número `n` de vectores. Para ello recibe una referencia a una subrutina implantando una operación binaria conmutativa y una lista de referencias a listas (una matriz en la que las filas pueden ser de distinta longitud). Por ejemplo, las siguientes llamadas son legales:

```
my @m  = operate( \&plus, \@b, \@a);
my @m2 = operate( \&times, \@a, \@b, \@c);
```

Donde las funciones `plus` y `times` vienen definidas por:

```
sub plus  {  $_[0]+$_[1] }
sub times {  $_[0]*$_[1] }
```

La siguiente ejecución ilustra el modo de funcionamiento:

```
lhp@nereida:~/Lperl/src$ vect3.pl
a = (3 2 1 9)
b = (5 6 7)
c = (1 2)
a+b:
8 8 8 9
a*b*c:
15 24 7 9
max(a, b, c):
5 6 7 9
```

La rutina `operate` devuelve un vector conteniendo el resultado de operar vectorialmente los diferentes vectores.

Si un vector es mas corto que otro se supone que es extendido a la longitud del mas largo con elementos iguales al elemento neutro de la operación usada. La operación se supone conmutativa:

```
lhp@nereida:~/Lperl/src$ cat -n vect3.pl
#!/usr/bin/perl  -w
use strict;
use List::Util qw(reduce max);

sub operate {
  my $rop = shift;
  die "Provide a subref\n" unless ref($rop) eq 'CODE';
  my @length = map { scalar(@$_) } @_; # lengths of the different lists

  # index of longest list
  our ($a, $b);
  my $longest = reduce { $length[$a] < $length[$b]?$b:$a } 0..$#_;
  my @res = @{splice @_, $longest, 1};

  for my $v (@_) {
    my $i = 0;
    @res[0..$#$v] = map { $rop->($_, $res[$i++]) } @$v;
  }
  return @res;
```

```
}

sub plus  {  $_[0]+$_[1] }
sub times {  $_[0]*$_[1] }

my @a = (3, 2, 1, 9); print "a = (@a)\n";
my @b = (5, 6, 7);    print "b = (@b)\n";
my @c = (1, 2);       print "c = (@c)\n";
my @m = operate( \&plus, \@b, \@a);
print "a+b:\n@m\n";
my @m2 = operate( \&times, \@a, \@b, \@c);
print "a*b*c:\n@m2\n";
my @m3 = operate( \&max, \@a, \@b, \@c);
print "max(a, b, c):\n@m3\n";
```

La función `reduce` aplica el bloque de código a la lista. Inicialmente las variables `$a` y `$b` contienen los dos primeros elementos. El bloque es llamado y el resultado devuelto por el bloque queda en `$a`. A continuación `$b` recorre los elementos de la lista siendo operado con `$a` y dejando siempre el resultado en `$a`. El valor final de `$a` es el valor devuelto por `reduce`.

4.4. Referencias a almacenamiento anónimo

Listas Anónimas

Una lista de expresiónes entre corchetes como:

```
[ 'e1', 'e2', 'e3'];
```

se denomina *array anónimo*. La expresión 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.

Hashes Anónimos

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};
```

En este caso se dice que tenemos un *hash anónimo*.

Subrutinas Anónimas

También puede hacerse con las subrutinas. Para tener una *subrutina anónima* basta con suprimir el nombre de la subrutina:

```
$subref = sub { my $x = shift; $x*$x; };
```

Ejercicio 4.4.1.

- *¿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`*?*

- *Explique el significado del siguiente código:*

  ```
  $association = [ cat =>"nap", dog=>"gone", mouse=>"ball"];
  $a = { @$association }->{mouse};
  ```

- *Explique el significado de las siguientes sentencias y que queda en los diferentes arrays después de ejecutarlas:*

  ```
    DB<1> opendir D,'./'
    DB<2> @n = readdir D
    DB<3> @nd = map { {N=>$_, D=>-M $_}} @n
    DB<4> @snd = sort { $b->{D} <=> $a->{D} } @nd
    DB<5> @sn = map { $_->{N} } @snd
  ```

  Recuerde que `-M` *retorna el tiempo desde la última modificación en días.*

4.5. 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
- Diferencia simétrica
- Convertir un conjunto en una cadena

La siguiente sesión con el depurador le ayudará a encontrar las respuestas a algunas de las preguntas que se le plantearán durante la resolución de la práctica:

```
  DB<1> @a{1..4} = (); @p{0, 2, 4, 6, 8} = ()
  DB<2> @u{keys %a, keys %p} = ()             # unión
  DB<3> @i{grep {exists $p{$_}} keys %a} = ()  # intersección
```

4.6. Estructuras anidadas

Listas Anidadas

Es posible crear estructuras de datos multidimensionales usando `listas` anónimas.

```
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];
```

Es opcional escribir la flecha entre dos corchetes:

```
print $table->[$x][$y];
```

Hashes Anidados

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";
```

En general las secuencias `]->{`, `}->[`, etc. pueden ser abreviadas omitiendo la flecha: `]{`, `}[`, etc.

4.7. Asignación Implícita de Memoria y Autovivificación

Partes Derechas

Cuando una variable de tipo lista se evalúa en una expresión se produce una copia de la misma. 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
```

En la línea 2 se crea una referencia a una copia del array `@a`. Si se quiere crear una referencia al array `@a` o a uno de sus elementos lo que debe ponerse en la parte derecha es la referencia al mismo:

```
lhp@nereida:~/Lperl/src$ perl -wde 0
main::(-e:1):   0
  DB<1> @a = 1..5
  DB<2> $b = \$a[3]
  DB<3> $$b = 9999
  DB<4> x @a
0  1
1  2
2  3
3  9999
4  5
  DB<5> $c = \@a
  DB<6> $c->[3] = -1
  DB<7> x @a
0  1
1  2
2  3
3  '-1'
4  5
```

Alojamiento Automático de Memoria

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.

Autovivificación

Se denomina *Autovivificación* o *Autovivification* a la creación automática de una referencia cuando una valor indefinido es de-referenciado. El siguiente ejemplo ilustra una autovivificación:

```
lhp@nereida:~$ perl -de 0
DB<1> $href->{Fruit}->{Bananas}->{Yellow} = "In Season"
DB<2> print "Red Apples are not in season.\n" unless exists $href->{Fruit}->{Apples}->{Red}
Red Apples are not in season.
DB<3> print $href->{Fruit}->{Apples}->{Red}

DB<4> print join "\n", keys %{$href->{Fruit}}
Bananas
Apples
```

Cuando un valor `undef` es de-referenciado siendo usado como *lvalue* se autovivifica:

```
>perl -Mstrict -wle 'my $r; my @x; @$r=@x; print "$r\n"'
ARRAY(0x225280)
```

Es en cierto modo razonable, ya que si está en el lado izquierdo es que se le quiere definir. Esto no ocurre si se usa como *rvalue*:

```
>perl -Mstrict -wle 'my $r; my @x; @x=@$r; print "$r\n"'
Can't use an undefined value as an ARRAY reference at -e line 1.
```

Distinguir cuando una expresión esta siendo usada como rvalue o lvalue puede ser sutíl. Por ejemplo:

```
  DB<1> x use strict; my $r = undef; my @x = map{ $_ } @{ $r }
  empty array
```

4.8. Impresión de Estructuras Anidadas

Para imprimir una estructura de datos compleja existen varias soluciones. La mas común es usar el módulo `Data::Dumper`.

La Función `dumpValue` de `dumpvar.pl`

El módulo forma parte de la distribución de Perl desde la versión `5.5`:

```
nereida:~> perl -MModule::CoreList -e \
   'print Module::CoreList->first_release("Data::Dumper")."\n"'
5.005
```

Si la versión de Perl con la que trabajamos es antigua, existe una solución se puede usar la función `dumpValue` que 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`

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.

```
nereida:~/alu/0607/alu2644> perl -wde 0
main::(-e:1):   0
  DB<1> use Data::Dumper
  DB<2> @a = (1, {A => ['AB', 'empty'], B => ['bB', 'empty']})
  DB<3> p Dumper(\@a)
$VAR1 = [
          1,
          {
            'A' => [
                     'AB',
                     'empty'
                   ],
            'B' => [
                     'bB',
                     'empty'
                   ]
```

```
          }
        ];
  DB<4> $Data::Dumper::Indent = 0
  DB<5> p Dumper(\@a)
$VAR1 = [1,{'A' => ['AB','empty'],'B' => ['bB','empty']}];
  DB<6> $Data::Dumper::Indent = 1
  DB<8> p Dumper(\@a)
$VAR1 = [
  1,
  {
    'A' => [
      'AB',
      'empty'
    ],
    'B' => [
      'bB',
      'empty'
    ]
  }
];
  DB<9> $Data::Dumper::Indent = 3
  DB<10> p Dumper(\@a)
$VAR1 = [
          #0
          1,
          #1
          {
            'A' => [
                     #0
                     'AB',
                     #1
                     'empty'
                   ],
            'B' => [
                     #0
                     'bB',
                     #1
                     'empty'
                   ]
          }
        ];
```

4.9. Ejemplo: El Producto de Matrices

El siguiente ejemplo muestra una subrutina `matrixProd` que realiza el producto de matrices. Veamos un ejemplo de llamada:

```
lhp@nereida:~/Lperl/src$ sed -ne '58,64p' matrixproduct.pl | cat -n
     1  my $A = [[1,2,3],[2,4,6],[3,6,9]];
     2  my $B = [[1,2],[2,4],[3,6]];
     3  my $C = matrixProd($A,$B);
     4
     5  print "Matriz A \n"; printMat($A);
     6  print "\nMatriz B \n"; printMat($B);
```

```
     7  print "\nMatriz C \n"; printMat($C);
```

La ejecución del programa produce la siguiente salida:

```
lhp@nereida:~/Lperl/src$ matrixproduct.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
```

Producto de Matrices Usando Referencias

Sigue el código e la función `matrixProd`:

```
lhp@europa:~/projects/perl/src/perltesting$  sed -ne '1,26p' matrixproduct.pl | cat -n
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use List::Util qw(sum);
 4  use List::MoreUtils qw(any);
 5  use Scalar::Util qw{reftype};
 6
 7  sub matrixProd {
 8    my ($A, $B) = @_;
 9    die "Error. Se esperaban dos matrices" if any { !defined($_) or reftype($_) ne 'ARRAY' } ($A, $B, @$
10
11    my $nrA = @$A;
12    my $nrB = @$B;
13    my $ncB = @{$B->[0]};
14    my @C;
15
16    die ("Las matrices no son multiplicables\n") if any { @$_ != $nrB } @$A;
17
18    for(my $i = 0; $i < $nrA; $i++)  {
19      for(my $j = 0; $j < $ncB; $j++) {
20        my $k = 0;
21        $C[$i][$j] = sum(map { $A->[$i][$k++] * $_ } map { $_->[$k] } @$B);
22      }
23    }
24
25    return \@C;
26  }
```

El Módulo PDL

Todo queda mucho mas simple usando el módulo *PDL* (PDL significa *Perl Data Language*) el cual da soporte al cálculo científico en Perl:

```
#!/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.

4.10. Ejercicio: Identificadores entre LLaves

En general, en un contexto en el que se espera un identificador de variable, Perl interpreta un identificador entre llaves como una cadena literal. Explique la conducta de Perl ante el siguiente código:

```
lhp@nereida:~/Lperl/src/testing$ cat -n shift.pl
     1  use strict;
     2
     3  my @puf = (1..10);
     4  toto(\@puf);
     5
     6  sub toto {
     7    my @a = @{shift};
     8    print "@a\n";
     9  }
lhp@nereida:~/Lperl/src/testing$ perl -c shift.pl
Global symbol "@shift" requires explicit package name at shift.pl line 7.
shift.pl had compilation errors.
```

¿Que ocurre si sustituimos la línea 5 por `my @a = @{shift()};`?

4.11. 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 4.1

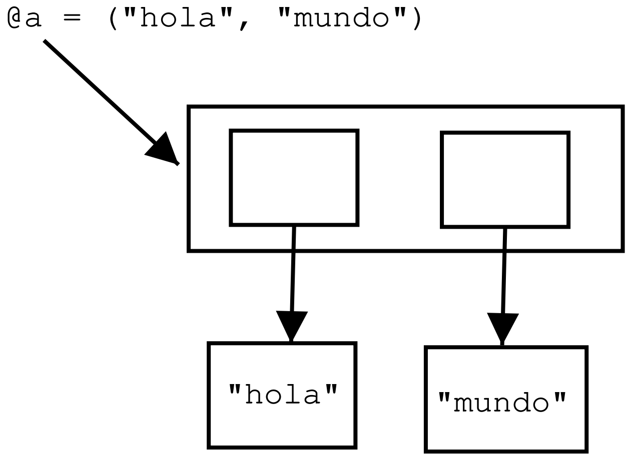

Figura 4.1: Un array es una colección de ligaduras lógicas a valores escalares

El Contador de Referencias de un Valor

- 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 dicho valor.

Véase la figura 4.2. Puesto que hay una ligadura entre la variable `$a` y su valor, este contador es al menos uno. 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.

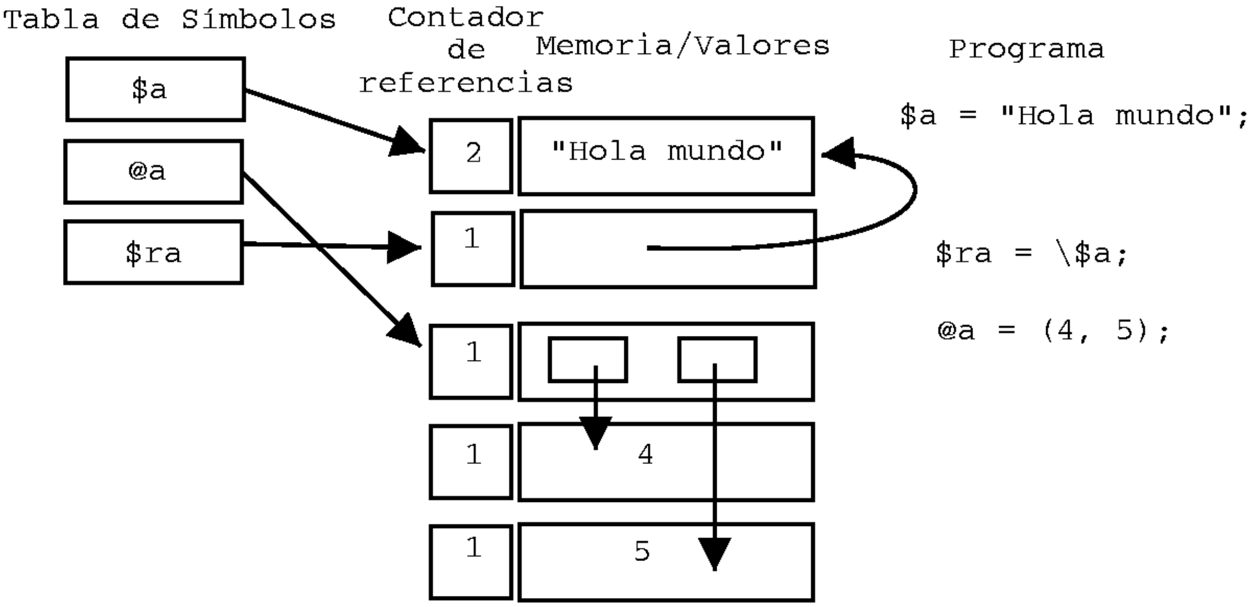

Figura 4.2: Contadores de referencia y asignaciones

Ejemplo

El siguiente ejemplo, debido a Ikegami (PerlMonks), está tomado de una respuesta en PerlMonks. Suponga el siguiente código:

```
1 my $array_ref = [];
2 for(1 .. 2) {
3     my $hash_ref = { foo => foo, bar => bar };
4     push(@{$array_ref}, $hash_ref);
5 }
6
7 my $hash_ref = $array_ref->[0];
8 $array_ref = [];
```

Estado después de la línea 7:

```
 $array_ref                                    $hash_ref
+--------------+                              +--------------+
| array ref    |                              | hash ref     |
```

```
+--------------+                              +--------------+
| refcount = 1 |                              | refcount = 1 |
+--------------+    +--------------------+    +--------------+
|           o=====> | array              |    |           o  |
+--------------+    +--------------------+    +---------- : -+
                    | refcount = 1       |                :
                    +--------------------+                :
                    | element 0          |                :
                    |  +--------------+  |                :
                    |  | hash ref     |  |                :
                    |  +--------------+  |                :
                    |  | refcount = 1 |  |                V
                    |  +--------------+  |    +--------------+
                    |  |           o========> | hash         |
                    |  +--------------+  |    +--------------+
                    +--------------------+    | refcount = 2 |
                    | element 1          |    +--------------+
                    |  +--------------+  |    | keys & vals  |
                    |  | hash ref     |  |    +--------------+
                    |  +--------------+  |
                    |  | refcount = 1 |  |
                    |  +--------------+  |    +--------------+
                    |  |           o========> | hash         |
                    |  +--------------+  |    +--------------+
                    +--------------------+    | refcount = 1 |
                                              +--------------+
                                              | keys & vals  |
                                              +--------------+
```

Despúes que se sobreescriba la referencia al array (línea 8):

```
+--------------+                              +--------------+
| array ref    |                              | hash ref     |
+--------------+                              +--------------+
| refcount = 1 |                              | refcount = 1 |
+--------------+    +--------------------+    +--------------+
|           o  |    | array              |    |           o  |
+---------- : -+    +--------------------+    +---------- : -+
            :       | refcount = 0       |                :
            :       +--------------------+                :
            V       | element 0          |                :
+--------------+    |  +--------------+  |                :
| array        |    |  | hash ref     |  |                :
+--------------+    |  +--------------+  |                :
| refcount = 1 |    |  | refcount = 1 |  |                V
+--------------+    |  +--------------+  |    +--------------+
| no elements  |    |  |           o========> | hash         |
+--------------+    |  +--------------+  |    +--------------+
                    +--------------------+    | refcount = 2 |
                    | element 1          |    +--------------+
                    |  +--------------+  |    | keys & vals  |
                    |  | hash ref     |  |    +--------------+
                    |  +--------------+  |
                    |  | refcount = 1 |  |
```

```
                    |  +--------------+  |    +--------------+
                    |  |           o========> | hash         |
                    |  +--------------+  |    +--------------+
                    +--------------------+    | refcount = 1 |
                                              +--------------+
                                              | keys & vals  |
                                              +--------------+
```

El array original es liberado automáticamente puesto que su contador de referencias alcanza cero. Las estructuras de los contadores de los elementos alcanzan cero:

```
+--------------+                              +--------------+
| array ref    |                              | hash ref     |
+--------------+                              +--------------+
| refcount = 1 |                              | refcount = 1 |
+--------------+                              +--------------+
|           o  |                              |           o  |
+---------- : -+                              +---------- : -+
            :                                             :
            :                                             :
            V                                             :
+--------------+       +--------------+                   :
| array        |       | hash ref     |                   :
+--------------+       +--------------+                   :
| refcount = 1 |       | refcount = 0 |                   V
+--------------+       +--------------+       +--------------+
| no elements  |       |           o========> | hash         |
+--------------+       +--------------+       +--------------+
                                              | refcount = 2 |
                                              +--------------+
                       +--------------+       | keys & vals  |
                       | hash ref     |       +--------------+
                       +--------------+
                       | refcount = 0 |
                       +--------------+       +--------------+
                       |           o========> | hash         |
                       +--------------+       +--------------+
                                              | refcount = 1 |
                                              +--------------+
                                              | keys & vals  |
                                              +--------------+
```

Los contadores de referencias de los hashes alcanzan cero y son liberados. Ahora el contador del antiguo segundo elemento alcanza cero:

```
+--------------+                              +--------------+
| array ref    |                              | hash ref     |
+--------------+                              +--------------+
| refcount = 1 |                              | refcount = 1 |
+--------------+                              +--------------+
|           o  |                              |           o  |
+---------- : -+                              +---------- : -+
            :                                             :
            :                                             :
            V                                             :
```

```
+--------------+                                          :
| array        |                                          :
+--------------+                                          :
| refcount = 1 |                                          V
+--------------+                              +--------------+
| no elements  |                              | hash         |
+--------------+                              +--------------+
                                              | refcount = 1 |
                                              +--------------+
                                              | keys & vals  |
                                              +--------------+


                                              +--------------+
                                              | hash         |
                                              +--------------+
                                              | refcount = 0 |
                                              +--------------+
                                              | keys & vals  |
                                              +--------------+
```

y es liberado:

```
+--------------+                              +--------------+
| array ref    |                              | hash ref     |
+--------------+                              +--------------+
| refcount = 1 |                              | refcount = 1 |
+--------------+                              +--------------+
|           o  |                              |           o  |
+---------- : -+                              +---------- : -+
            :                                             :
            :                                             :
            V                                             :
+--------------+                                          :
| array        |                                          :
+--------------+                                          :
| refcount = 1 |                                          V
+--------------+                              +--------------+
| no elements  |                              | hash         |
+--------------+                              +--------------+
                                              | refcount = 1 |
                                              +--------------+
                                              | keys & vals  |
                                              +--------------+
```

Liberación de Memoria en Estructuras de Datos Cíclicas

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` y las funciones `weaken` e `isweak` en el módulo `Scalar::Util`.

```
pp2@nereida:~$ perl -wde 0
main::(-e:1):   0
  DB<1> use Scalar::Util qw(weaken isweak)
  DB<2> $var = "hola\n"
```

```
  DB<3> $ref = \$var
  DB<4> weaken($ref)
  DB<5> p isweak($ref)
1
  DB<6> $c = $ref
  DB<7> p "<".isweak($c).">"
<>
```

La función `weaken`

La función `weaken` recibe como argumento una referencia. Hace que no se incremente el contador de referencias del valor asociado con dicha referencia. Vea el siguiente ejemplo:

```
hp@nereida:~/Lperl/src$ cat -n noweaken.pl
1  use strict;
2  use Scalar::Util qw(weaken);
3  my $ref;
4  {
5    my $var = "hola\n";
6    $ref = \$var;
7  }
8  print $$ref;
```

```
lhp@nereida:~/Lperl/src$ cat -n weaken.pl
 1  use strict;
 2  use Scalar::Util qw(weaken);
 3  my $ref;
 4  {
 5    my $var = "hola\n";
 6    $ref = \$var;
 7    weaken($ref);
 8  }
 9  print $$ref;
```

```
lhp@nereida:~/Lperl/src$ perl noweaken.pl
hola
```

```
lhp@nereida:~/Lperl/src$ perl weaken.pl
Can’t use an undefined value as a SCALAR reference ...
```

Una referencia se dice *débil* (weak) si no cuenta en el objeto al que referencia. Cuando el contador del objeto al que referencia alcanza cero la referencia contiene `undef`.

Tenga en cuenta que la copia de una referencia débil (creada con `weaken`) es una referencia fuerte:

```
lhp@nereida:~/Lperl/src$ cat -n weakenwithcopy.pl
 1  use strict;
 2  use Scalar::Util qw(weaken);
 3  my ($ref, $copy);
 4  {
 5    my $var = "hola\n";
 6    $ref = \$var;
 7    weaken($ref);
 8    $copy = $ref;
 9  }
10  print $$ref;
11
lhp@nereida:~/Lperl/src$ perl weakenwithcopy.pl
hola
```

La función `isweak`

La función `isweak` retorna `TRUE` si la expresión pasada como parámetro es una referencia débil.

Detectando Ciclos El módulo Devel::Cycle permite la detección de ciclos:

```
lhp@nereida:~/Lperl/src/testing$ cat -n ciclos.pl
     1  #!/usr/local/bin/perl -w
     2  use strict;
     3  use Devel::Cycle;
     4  my $test = {
     5    fred   => [qw(a b c d e)],
     6    ethel  => [qw(1 2 3 4 5)],
     7    george => {martha => 23, agnes  => 19}
     8  };
     9  $test->{george}{phyllis} = $test;
    10  $test->{fred}[3]      = $test->{george};
    11  $test->{george}{mary} = $test->{fred};
    12  find_cycle($test);
    13  exit 0;
lhp@nereida:~/Lperl/src/t
```

Al ejecutar produce una salida como esta:

```
lhp@nereida:~/Lperl/src/testing$ ./ciclos.pl
Cycle (1):
                          $A->{'fred'} => \@B
                               $B->[3] => \%C
                          $C->{'mary'} => \@B

Cycle (2):
                          $A->{'fred'} => \@B
                               $B->[3] => \%C
                       $C->{'phyllis'} => \%A

Cycle (3):
                        $A->{'george'} => \%C
                          $C->{'mary'} => \@B
                               $B->[3] => \%C

Cycle (4):
                        $A->{'george'} => \%C
                       $C->{'phyllis'} => \%A
```

El módulo permite comprobar la existencia de ciclos con referencias débiles mediante la función `find_weakened_cycle`. El programa:

```
lhp@nereida:~/Lperl/src/testing$ cat -n ciclosweaken.pl
 1  #!/usr/local/bin/perl -w
 2  use strict;
 3  use Scalar::Util qw(weaken);
 4  use Devel::Cycle;
 5  my $test = {
 6    fred   => [qw(a b c d e)],
 7    ethel  => [qw(1 2 3 4 5)],
 8    george => {martha => 23, agnes  => 19}
 9  };
10  $test->{george}{phyllis} = $test;
11  $test->{fred}[3]      = $test->{george};
12  $test->{george}{mary} = $test->{fred};
13  weaken($test->{george}->{phyllis});
```

```
find_weakened_cycle($test);
exit 0;
```

Produce la salida:

```
lhp@nereida:~/Lperl/src/testing$ ciclosweaken.pl
Cycle (1):
                          $A->{'fred'} => \@B
                               $B->[3] => \%C
                          $C->{'mary'} => \@B

Cycle (2):
                          $A->{'fred'} => \@B
                               $B->[3] => \%C
                   w-> $C->{'phyllis'} => \%A

Cycle (3):
                        $A->{'george'} => \%C
                          $C->{'mary'} => \@B
                               $B->[3] => \%C

Cycle (4):
                        $A->{'george'} => \%C
                   w-> $C->{'phyllis'} => \%A
```

El Módulo `Devel::Peek`

Estudie la siguiente ejecución con el depurador. El módulo `Devel::Peek` nos permite observar la estructura interna de las variables en el intérprete:

```
lhp@nereida:~/Lperl/src$ perl -MDevel::Peek -wd weakenwithcopy.pl
main::(weakenwithcopy.pl:3):    my ($ref, $copy);
main::(weakenwithcopy.pl:4):    {
  DB<1> c 10
main::(weakenwithcopy.pl:10):   print $$ref;
  DB<2> -
1:      use strict;
2:      use Scalar::Util qw(weaken);
3:      my ($ref, $copy);
4       {
5:        my $var = "hola\n";
6:        $ref = \$var;
7:        weaken($ref);
8:        $copy = $ref;
9       }
10==>   print $$ref;
  DB<2> Dump $ref
SV = RV(0x817b328) at 0x823a050
  REFCNT = 2
  FLAGS = (PADBUSY,PADMY,ROK,WEAKREF)
  RV = 0x814f7e0
  SV = PVMG(0x83dcb80) at 0x814f7e0
    REFCNT = 1
    FLAGS = (PADBUSY,PADMY,RMG,POK,pPOK)
    IV = 0
```

```
    NV = 0
    PV = 0x8238fa8 "hola\n"\0
    CUR = 5
    LEN = 8
    MAGIC = 0x83e3368
      ............................
  DB<3> Dump $copy
SV = RV(0x817b324) at 0x814f810
  REFCNT = 2
  FLAGS = (PADBUSY,PADMY,ROK)
  RV = 0x814f7e0
  SV = PVMG(0x83dcb80) at 0x814f7e0
    REFCNT = 1
    FLAGS = (PADBUSY,PADMY,RMG,POK,pPOK)
    IV = 0
    NV = 0
    PV = 0x8238fa8 "hola\n"\0
    CUR = 5
    LEN = 8
    MAGIC = 0x83e3368
      ............................
```

Contadores de Referencias de Listas

El siguiente ejemplo muestra como los elementos de un array tienen sus propios contadores de referencias:

```
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.

4.12. 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 una variable de paquete. Esto se conoce como *referenciado simbolico*. Su uso es mas eficiente que la alternativa de usar `eval`. Veamos un ejemplo (obsérvese la ausencia de `-w` y de `use strict`):

```
lhp@nereida:~/Lperl/src/testing$ cat -n symbolicref.pl
 1  #!/usr/bin/perl
 2  sub data {
 3    my ($fa, $sa) = @_;
 4    print "Inside data\n";
 5    print "First list: @$fa\n";
 6    print "Second list: @$sa\n";
 7  }
```

```

$data = 4;
@data = (1,2,3,4);
@tutu = 5..8;
$name = "data";

print "${$name}\n";
push  @{$name}, @data;
print "@data\n";
&{$name}(’data’, ’tutu’);
```

La ejecución da como resultado:

```
lhp@nereida:~/Lperl/src/testing$ symbolicref.pl
4
1 2 3 4 1 2 3 4
Inside data
First list: 1 2 3 4 1 2 3 4
Second list: 5 6 7 8
```

El Pragma `strict` y el Referenciado Simbólico

El pragma `strict` impide el uso de referencias simbólicas:

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

use strict;

our $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.
```

`use strict ’refs’`

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 se ha usado our
$a = "x"; # $a tampoco es 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.
```

`no strict ’refs’`

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`:

```
lhp@nereida:~/Lperl/src$ cat -n symbol_ref3.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3
 4  our ($x, $y, $z) = (4, 5, 6);
 5  my $a = "x";
 6  {
 7    no strict 'refs';
 8    $a = <>;
 9    chomp($a);
10    $$a = 10;
11  }
12
13  print "x=$x, y=$y, z=$z\n";
lhp@nereida:~/Lperl/src$ ./symbol_ref3.pl
z
x=4, y=5, z=10
```

Referenciado Simbólico y Ámbito de las Variables

- 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.
- *Toda referencia simbólica se refiere a una variable de paquete. Las variables léxicas no pueden ser referenciadas a través de referencias simbólicas.* Las referencias simbólicas acceden a la tabla de símbolos y las variables léxicas no se almacenan en las tablas de símbolos - solo las declaradas con `our` lo son. *Las referencias simbólicas 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
  ```

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;
```

El Operador Flecha y las Referencias Simbólicas

Las referencias simbólicas tambien pueden usarse a la izquierda del operador `->`

```
$symref = "set";
$symref->{type} = "discrete"; # Lo mismo que: $set->{type} = "discrete";
```

El pragma `strict` no protesta del uso de referenciados simbólicos cuando este se hace mediante el operador `->`.

Un Ejemplo de uso de Referenciado Simbólico: La función `can`

El ejemplo que sigue muestra un pequeño intérprete. El usuario llama al programa con un comando y un glob que especifica un conjunto de ficheros.

```
lhp@nereida:~/Lperl/src/perl_testing_adn_examples/chapter_09$ symbolic.pl dirs /p*
/pendriver
/proc
lhp@nereida:~/Lperl/src/perl_testing_adn_examples/chapter_09$ symbolic.pl latest *
symbolic.pl
lhp@nereida:~/Lperl/src/perl_testing_adn_examples/chapter_09$ symbolic.pl sort_by_time *.pl
symbolic.pl
simplecalc.pl
make_test_files.pl
filefilter.pl
lhp@nereida:~/Lperl/src/perl_testing_adn_examples/chapter_09$ symbolic.pl chuchu *.pl
Unknown command 'chuchu'
```

Para cada comando el programa implanta una función de filtro e imprime el resultado del filtro.

```
lhp@nereida:~/Lperl/src/perl_testing_adn_examples/chapter_09$ cat -n symbolic.pl
 1  #!/usr/local/bin/perl
 2  use strict;
 3  use warnings;
 4
 5  main( @ARGV );
 6
 7  sub main {
 8      die "Usage:\n$0 <command> [file_pattern]\n" unless @_;
 9      my $command = shift;
10
11      die "Unknown command '$command'\n" unless  main->can( $command );
12
13      no strict 'refs';
14      my @r = $command->( @_ );
15      local $" = "\n";
16      print "@r\n";
17  }
18
19  sub sort_by_time {
20      map  { $_->[0] }
21      sort { $a->[1] <=> $b->[1] }
22      map  { [ $_, -M $_ ] } @_
23  }
24
25  sub latest {
26      (sort_by_time( @_ ) )[0];
```

```
27  }
28
29  sub dirs {
30      grep { -d $_ } @_;
31  }
```

La principal novedad en este código se refiere al control (línea 11) de que comandos están disponibles. La función `can` puede ser llamada de la forma:

```
paquete->can('nombre_de_subrutina')
```

devuelve cierto si existe una subrutina con ese paquete y falso en caso contrario. De hecho devuelve una referencia (dura/no simbólica) a la subrutina si esta existe.

Recuerde

- 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, las léxicas, 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 `::`.

4.12.1. Práctica: Referenciado Simbólico

A menudo el uso de referencias simbólicas puede ser sustituido mediante el uso de un hash de ámbito léxico. Reescriba el ejemplo anterior *Un Ejemplo de uso de Referenciado Simbólico: La función* `can` utilizando un hash de referencias a funciones.

4.13. 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");
```

4.14. 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.

```
lhp@nereida:~/Lperl/src$ cat -n readfunction.pl
#!/usr/local/bin/perl5.8.0 -w
use strict;

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
```

4.14.1. 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. Aqui tiene una indicación de como hacerlo:

```
lhp@nereida:~/Lperl/src$ cat switch.pl
#!/usr/local/bin/perl -w
use strict;
use warnings;

sub switch {
  my $exp = shift or die "switch error: expecting an expression\n";
  die "switch error. Empty cases\n" unless @_;
  my %cases = @_;

  if (exists($cases{$exp})) {
    die "switch error. Expected code for <$exp>\n" unless ref($cases{$exp}) eq 'CODE';
    ........................
  }
  .........................
}

my $x = 4;
switch $x,
  2 => sub { print "two\n" },
  3 => sub { print "three\n" },
  4 => sub { print "four\n" },
  default => sub { print "Other value\n" }
;
```

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
     1  #!/usr/local/bin/perl5.8.0 -w
     2
     3  use Switch;
     4
     5  $val = shift;
     6
     7  switch ($val) {
     8    case 1 { print "number 1\n" }
     9    case "hello" { print "string hello\n" }
    10    else { print " none of these two\n" }
    11  }
```

4.15. Typeglobs

4.15.1. 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 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*.

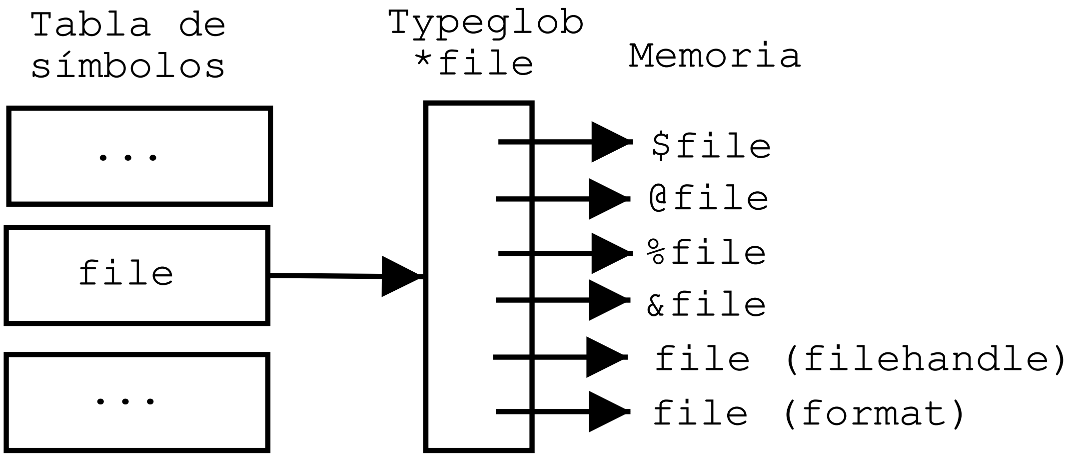

Figura 4.3: Un typeglob nombra a la estructura que se interpone entre la tabla de símbolos y la memoria

Perl provee una sintáxis especial, denominada ”typeglob” para referirse conjuntamente a los diferentes tipos de variable: `*file`. (Piensa en `*` como en un comodín).

4.15.2. 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, ”’ash”, etc.). Cuando se asigna un ”typeglob” a otro se copia dicho vector de referencias.

4.15.3. 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*.

Es un error declarar una variable 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.
```

Ejercicio 4.15.1. *Explique el mensaje de advertencia en la compilación del siguiente programa:*

```
lhp@nereida:~/Lperl/src$ cat -n mytypeglob2.pl
 1  #!/usr/bin/perl  -w
 2  use strict;
 3
 4  our $a;
 5  my $b;
 6
 7  *a = *b;
 8  $a = 4;
 9  print "$b\n";
lhp@nereida:~/Lperl/src$ perl -c mytypeglob2.pl
Name "main::b" used only once: possible typo at mytypeglob2.pl line 7.
mytypeglob2.pl syntax OK
```

Ejercicio 4.15.2. *¿Que se imprimirá al ejecutar el siguiente código?*

```
$a = 3.14159;
my $a = 5;
*b = *a;
print $b;
```

4.15.4. `local` y typeglobs

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
```

4.15.5. Paso de parámetros a una subrutina por medio de typeglobs

Ya hemos visto que las referencias permiten el paso de estructuras anidadas a una subrutina, evitando el efecto del aplanamiento de listas. Las referencias simbólicas también proveen una forma alternativa (pero desaconsejada). Los typlegobs nos dan otra via (tampoco recomendable):

```
lhp@nereida:~/Lperl/src$ cat -n passbyref.pl
#!/usr/local/bin/perl
use strict;
use warnings;

sub sum {
  our (@a, @b);
  local (*a, *b) = @_;

  $a[$_] += $b[$_] for 0..$#a;
}

our @c = 1..4;
our @d= 5..8;

sum(*c, *d);
print "@c\n";
```

Cuando se ejecuta, el programa anterior produce la salida:

```
lhp@nereida:~/Lperl/src$ ./passbyref.pl
6 8 10 12
```

Paso de un Fichero sin Prefijo como Typeglob

Los manejadores de ficheros (filehandles) pueden ser usados sin declaración previa cuando se usan sin prefijo. En tal caso no se pueden asignar o pasar como parámetros, esto es, es ilegal hacer:

```
open(F,"fich1"); # Recomendado: open my $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:

```
lhp@nereida:~/Lperl/src$ cat -n fileparameter4.pl
#!/usr/bin/perl
use warnings;
use strict;

sub welcome {
  local *FHANDLE = shift;

  print FHANDLE "Welcome ...\n";
}

open(FILE, ">test.txt");
welcome(*FILE);
close(FILE);
```

Paso de un Fichero sin Prefijo como Referencia a un Typeglob

Otra forma de pasar un manejador de fichero sin prefijo es usar una referencia a un typeglob:

```
#!/usr/bin/perl -w
sub welcome {
  my $fh = shift;

  print $fh "Welcome ...\n";
}

open(FILE, ">test.txt");
$file = *FILE;
welcome($file);
close($file);
```

El Módulo `IO::File`

El modo recomendado de uso de ficheros consiste en tratarlos como objetos de la clase `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;
```

4.15.6. 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`:

```
lhp@nereida:~/Lperl/src$ cat -n benchtypeglobs3.pl
 1  #!/usr/bin/perl
 2  use strict;
 3  use Benchmark;
 4
 5  our ($i, $r, $b, $s);
 6
 7  local $i;
 8  *b = *i;
 9  $r = \$i;
10  $s = "i";
11
12  no strict 'refs';
13  timethese(4,
14    { reference => sub { for ($$r = 0; $$r < 1e6; $$r++) { $$r; } },
15      typeglob  => sub { for ($b = 0; $b < 1e6; $b++) { $b; } },
```

```
      symbolic  => sub { for ($$s = 0; $$s < 1e6; $$s++) { $$s; } },
    });
```

El resultado de la ejecución en `nereida` nos da la siguiente relación:

```
lhp@nereida:~/Lperl/src$ benchtypeglobs3.pl
Benchmark: timing 4 iterations of reference, symbolic, typeglob...
 reference:  1 wallclock secs ( 1.13 usr +  0.00 sys =  1.13 CPU) @  3.54/s (n=4)
  symbolic:  6 wallclock secs ( 6.33 usr +  0.00 sys =  6.33 CPU) @  0.63/s (n=4)
  typeglob:  1 wallclock secs ( 0.75 usr +  0.00 sys =  0.75 CPU) @  5.33/s (n=4)
```

4.15.7. Typeglobs Selectivos

Al asignar una referencia a un typeglob se obtiene un *Typeglob Selectivo*:

```
*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.

Del mismo modo `@args` es un sinónimo de `@ARGV` pero no de `$ARGV` y `do_it` es el nombre de una subrutina que inicialmente era anónima.

Referencias a Typeglobs

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
```

Veamos otro ejemplo:

```
lhp@nereida:~/Lperl/src$ perl -wde 0
main::(-e:1):   0
  DB<1> %variable = (v => "ven", a => "a", r=> "rumania")
  DB<2> sub variable { print "esto es una variable" }
  DB<3> $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:

```
  DB<4> x %{*$typeglob_ref}
0  'r'
1  'rumania'
2  'a'
3  'a'
```

```
4  'v'
5  'ven'
  DB<5> x &{*$typeglob_ref}()
0  1
  DB<6> x *$typeglob_ref->()
0  1
  DB<7> x *$typeglob_ref->{r}
0  'rumania'
```

Ejercicio 4.15.3. *¿Por que no sale por pantalla el mensaje* `"esto es una variable"` *en las líneas 5,6 y 7? ¿Por que la respuesta es* `"1"`*?*

Referencias a Ficheros (Barewords) En la sección 4.15.5 abordabamos el problema de los ficheros como argumentos de una subrutina. El código puede ser reescrito como sigue:

```
lhp@nereida:~/Lperl/src$ cat -n ./fileparameter2.pl
   1  #!/usr/bin/perl -w
   2  use strict;
   3
   4  sub welcome {
   5    my $fh = shift;
   6
   7    print $fh "Welcome ...\n";
   8  }
   9
  10  open(FILE, ">test.txt");
  11  my $file = \*FILE;
  12  welcome($file);
  13  close($file);
lhp@nereida:~/Lperl/src$ ./fileparameter2.pl
lhp@nereida:~/Lperl/src$ cat test.txt
Welcome ...
```

Ejercicio 4.15.4. *¿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();
```

4.15.8. 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. Las siguientes parejas de asignaciones son equivalentes:

```
$scalar_ref = *variable{SCALAR} # igual a $scalar_ref = \$variable
$array_ref  = *variable{ARRAY}  # igual a $scalar_ref = \@variable
$hash_ref   = *variable{HASH}   # igual a $scalar_ref = \%variable
$sub_ref    = *variable{CODE}   # igual a $scalar_ref = \&variable
```

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
```

Ejercicio 4.15.5. *Dada la asignación:*

`$h = *$t{HASH};`

¿Cuál es el tipo de `$t`*? ¿Que queda en* `$h`*?*

4.15.9. 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}`.

Creando un Envoltorio para una Subrutina

Supongamos que queremos crear una función `wrap` que establece un envoltorio (en la jerga un *wrapper*) a la llamada a una subrutina que se pasa como parámetro.

```
wrap ’mi_rutina’, sub { print "Ejecutando mi_rutina con args <@_>\n" }
```

Por ejemplo, al hacer el siguiente wrap:

```
#!/usr/local/bin/perl -w
use strict;
use Scalar::Util qw(looks_like_number);
use List::MoreUtils qw(all);

sub wrap {
  my ($subname, $wrapper) = @_;
  die "Error in ’wrap’: Provide a sub name\n" unless $subname =~ /[a-zA-Z_]\w+/;
  die "Error in ’wrap’: No wrapper provided\n" unless ref($wrapper) eq ’CODE’;

  .....................................................................
}

sub square {
  return map { $_ * $_ } @_;
}

```

```
wrap('square',
     sub {
       my $subname = shift;

         die "Error en $subname: se esperaban números\n"
       unless all { looks_like_number($_) } @_;

       return @_;
     }
);


my @a = square(5, 2);
print "square(5, 2) = (@a)\n";
@a = square(3, 'Juan', 9);
print "square(3, 'Juan', 9) = (@a)\n";
```

Obtenemos como resultado la ejecución:

```
lhp@nereida:~/Lperl/src$ wrap.pl
square(5, 2) = (25 4)
Error en main::square: se esperaban números
```

La subrutina `wrap` en el código que sigue comienza comprobando los parámetros. A continuación se construye el nombre completo de la función. La función `caller` devuelve el nombre del paquete desde el que se llamó a la función actual (véase la sección 1.15.10).

```
lhp@nereida:~/Lperl/src$ cat -n wrap.pl
 6  sub wrap {
 7    my ($subname, $wrapper) = @_;
 8    die "Error in 'wrap': Provide a sub name\n" unless $subname =~ /^((::)?\w+)+$/;
 9    die "Error in 'wrap': No wrapper provided\n" unless ref($wrapper) eq 'CODE';
10
11    my $fullspec = ($subname =~ /::/)? $subname : caller()."::$subname";
12
13    no strict 'refs';
14    my $subptr = *$fullspec{CODE};
15
16    no warnings 'redefine';
17    *{$fullspec} = sub { my @r = $wrapper->($fullspec, @_); return $subptr->(@r); };
18  }
```

En la línea 14 se hace un referenciado simbólico a un typeglob: `*$fullspec`. El typeglob resultante es indexado como hash en la entrada `CODE` (repase la sección 4.15.8). Esto nos deja en `$subptr` una referencia a la subrutina que se desea *envolver*.

En la línea 17 procedemos a sustituir la subrutina original por nuestra subrutina envoltorio. El compilador perl producirá un warning indicando que la subrutina ya existe. Esa es la razón para el `no warnings 'redefine'` de la línea 16. Para hacer la sustitución basta con usar un typeglob selectivo.

4.15.10. Práctica: Construcción de un wrapper

Complete la subrutina `wrap` para que admita envolver la entrada y la salida de una subrutina:

```
wrap 'mirutina',
```

```
  pre => sub { print "Ejecutando ".shift()."  con args <@_>\n"; @_  },
  post => sub { print "Saliendo de mi_rutina\n"; @_  }
;
```

El código `post` recibe como argumentos al resultado de la subrutina 'envuelta'.

4.15.11. Suprimiendo Subrutinas con Typeglobs y Referenciado Simbólico

En (raras) ocasiones queremos eliminar completamente una subrutina. Las ocasiones en las que he visto en la necesidad de hacerlo son aquellas en las que la existencia de una subrutina afecta de forma no deseada al comportamiento de otra (recuerde que en Perl una subrutina puede saber dinámicamente si cierta subrutina existe mediante el uso de `can`).

La distribución Parse::Eyapp provee el módulo Parse::Eyapp::Base el cual provee las funciones `delete_method` e `insert_method` para suprimir y bautizar subrutinas. Vea un ejemplo de uso:

```
pl@nereida:~/LEyapp/examples$ cat -n localwithinsert.pl
 1  #!/usr/local/bin/perl -w
 2  use strict;
 3
 4  package Tutu;
 5  use Parse::Eyapp::Base qw(:all);
 6
 7  my $tutu = sub {
 8    "inside tutu\n"
 9  };
10
11  sub plim {
12    insert_method(_tutu => $tutu);
13
14    print _tutu();
15
16    delete_method('_tutu');
17  }
18
19  package main;
20
21  Tutu::plim();
22  print(main->can('Tutu::_tutu')? (Tutu::_tutu()."\n") : "Can't do tutu\n");
23  Tutu::plim();
```

Cuando se ejecuta el código anterior produce la salida:

```
pl@nereida:~/LEyapp/examples$ localwithinsert.pl
inside tutu
Can't do tutu
inside tutu
```

La subrutina `delete_method('subname', qw{class1 class2 class3})` elimina la subrutina `subname` en los paquetes `Class1`, `Class2` y `Class1`:

```
pl@nereida:~/LEyapp/lib/Parse/Eyapp$ sed -ne '144,172p' Base.pm | cat -n
 1  sub delete_method {
 2    my $name = pop;
 3    $name = '' unless defined($name);
 4    croak "Error in delete_method: Illegal method name <$name>\n" unless $name =~/^\w+$/;
```

```
  my @classes = @_;

  @classes = scalar(caller) unless @classes;
  no strict 'refs';
  for (@classes) {
    croak "Error in delete_method: Illegal class name <$_>\n" unless /^[\w:]+$/;
    unless ($_->can($name)) {
      print STDERR "Warning in delete_method: No sub <$name> to delete in package <$_>\n";
      next;
    }
    my $fullname = $_."::".$name;

    # Temporarily save the other entries
    my @refs = map { *{$fullname}{$_} } qw{HASH SCALAR ARRAY GLOB};

    # Delete typeglob
    *{$fullname} = do { local *{$fullname} };

    # Restore HASH SCALAR ARRAY GLOB entries
    for (@refs) {
      next unless defined($_);
      *{$fullname} = $_;
    }
  }
}
```

Se salvan las referencias a los viejos valores de los diversos tipos con un typeglob selectivo:

```
my @refs = map { *{$fullname}{$_} } qw{HASH SCALAR ARRAY GLOB}
```

Después se elimina la entrada del typeglob usando `local`:

```
*{$fullname} = do { local *{$fullname} }
```

La expresión `local *{$fullname}` elimina temporalmente el typeglob (lo deja `undef`) y retorna el typeglob. Como queremos hacer permanente la eliminación del typeglob. Véase el siguiente ejemplo ilustrativo con el depurador:

```
lhp@nereida:~/Lperl/src/perl_networking/ch2$ perl -wde 0
main::(-e:1):   0
  DB<1> @x = 1..5; $x = 'hola'
  DB<2> $c = 'x'
  DB<3> x  local *{$c}
0  *main::x
  DB<4> p $x
hola
  DB<6> x *{$c} = { local *{$c} }
Odd number of elements in anonymous hash at (eval 10)[/usr/share/perl/5.8/perl5db.pl:628] line 2.
  DB<7> x *{$c} = do { local *{$c} }
0  *main::x
  DB<8> print $x
Use of uninitialized value in print at (eval 12)[/usr/share/perl/5.8/perl5db.pl:628] line 2.
```

y por último se restauran los viejos valores excepto el de código.

4.15.12. El Módulo `Symbol`

El módulo `Symbol` provee un buen número de funciones para la manipulación de typeglobs y símbolos. Por ejemplo, es posible

- Suprimir un paquete completo usando la función `Symbol::delete_package`.

4.15.13. Práctica: Inserción de una Subrutina

Escriba una subrutina `insert_method` que cuando sea llamada:

```
 insert_method( qw{CLASS1 CLASS2 ... }, 'subname', sub { ... } )
```

Inserte en la tabla de símbolos la subrutina referenciada con nombre `subname` en los paquetes `Class1`, `Class2` y `Class3`. Si no se especifican nombres de paquete se asume que se instala en el paquete llamador.

4.16. Prototipos

Los *prototipos* en Perl son un mecanismo que permite

- La activación de ciertas comprobaciones de tipos
- Escribir subrutinas cuyo comportamiento disfrute de privilegios similares a los de los operadores internos de Perl.

Introducción: El Problema

Como ejemplo, supongamos que queremos escribir una función `shift2` que retorna y elimina los dos primeros elementos del principio de un array. Nos gustaría usarla como `shift`:

```
@a = 1..10;
($f, $s) = shift2 @a;
```

Si queremos escribirla, no vale hacer:

```
sub shift2 { splice @_, 0, 2 }
```

Observe el comportamiento del siguiente programa.

```
$ cat -n shift2.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3
     4  sub shift2 { splice @_, 0, 2 }
     5
     6  my @a = 1..5;
     7  my ($f, $g) = shift2 @a;
     8
     9  local $" = ',';
    10  print "f = $f, g = $g  \@a = (@a)\n";
$ ./shift2.pl
f = 1, g = 2  @a = (1,2,3,4,5)
```

La llamada a `splice` en la línea 4 retira Los elementos de `@_` pero no del array `@a`.

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;
```

Control de la LLamada Mediante Prototipos

Si queremos que la interfaz de `shift2` sea similar a la de `shift` debemos hacer uso de *prototipos*. Podemos redeclarar la función como:

```
sub shift2(\@) { splice @{$_[0]}, 0, 2 }
```

El prototipo `\@` indica dos cosas:

- Que la rutina espera un argumento de tipo lista
- Que la lista será automáticamente convertido a una referencia a un array

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.

LLamada con Ampersand y Prototipos

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.

Formato de los Prototipos

Los prototipos se construyen a partir de *átomos prototipo* los cuáles estan hechos de un prefijo o de un escape y un prefijo.

Prototipo Referencia a un Escalar

Un prototipo como `\$` le indica a Perl que en la llamada se debe especificar una variable escalar la cuál será convertida en la rutina en una referencia a la variable. Véase la sesión que sigue:

```
lhp@nereida:~/Lperl/src$ perl -wde 0
main::(-e:1):   0
  DB<1> sub r(\$) { print "@_\n" }
  DB<2> r qw{a b c}
Type of arg 1 to main::r must be scalar (not list) at (eval 6)
  DB<3> r "hola"
Type of arg 1 to main::r must be scalar (not constant item) at (eval 7)
  DB<4> $x = 4
  DB<5> r $x
SCALAR(0x842f57c)
  DB<6> r \$x
Type of arg 1 to main::r must be scalar (not single ref constructor) at (eval 10)
  DB<7> r $x, qw{a b c}
Too many arguments for main::r at (eval 11)
  DB<8> r $x, 1..3
Too many arguments for main::r at (eval 12)
```

Cualquier *prototipo backslash* (\) representa a un argumento que debe comenzar con ese carácter. De ahí las protestas del intérprete en las líneas 2, 3 y 6.

El Prototipo `$`

Un prototipo de la forma `$` lo que hace es que fuerza un contexto escalar. Observe la conducta del siguiente programa:

```
$ cat -n ./dollarproto.pl
     1  #!/usr/bin/perl -w
     2  use strict;
     3
     4  sub t ($@) { my $a = shift; my @b = @_; print "a = $a, b = (@b)\n"; }
     5
     6  my ($a, $b, $c) = qw/uno dos tres/;
     7  t ':',$a, $b, $c;
     8
     9  my @r = 1..5;
    10  t @r;
    11
$ ./dollarproto.pl
a = :, b = (uno dos tres)
a = 5, b = ()
```

¿Podría explicar la salida?

Los Prototipos `@` y `%`

Un prototipo del tipo `@` o del tipo `%` fuerza un contexto de lista y consume el resto de argumentos.

El Prototipo `&`

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 omitir en la llamada permitiendo un estilo similar al uso de los operadores `sort`, `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. Esta característica nos permite escribir funciones que siguen una sintáxis similar a la de `sort`.

El Prototipo *

El prototipo `*` fuerza una referencia a un typeglob.

Argumentos Requeridos y Argumentos Opcionales

Un prototipo de la forma `;` separa los argumentos requeridos de los argumentos opcionales.

Ejemplos

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 4.16.1. *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?*

Peligros en el Uso de Prototipos

El operador `mkdir` tiene un prototipo `$;$`. Véanse las consecuencias de semejante prototipo:

```
lhp@nereida:/tmp$ perl -wde 0
main::(-e:1):   0
  DB<1> @a = (’tutu’, ’077’)
  DB<2> x mkdir @a
0  1
  DB<3> !!ls -ltr | tail -1
drwxr-xr-x 2 lhp      lhp        4096 2008-04-16 14:05 2
  DB<4> x &mkdir @a
syntax error at (eval 7)[/usr/share/perl/5.8/perl5db.pl:628] line 2, near "&mkdir @a"
  DB<5> x mkdir $a[0], $a[1]
0  1
  DB<6> !!ls -ltr | tail -2
drwxr-xr-x 2 lhp      lhp        4096 2008-04-16 14:05 2
d--x--xr-x 2 lhp      lhp        4096 2008-04-16 14:07 tutu
```

Ejercicio 4.16.2. *¿Podría explicar el resultado?*

La función `prototype`

La función `prototype` retorna una cadena describiendo el prototipo de la referencia a la función pasada como argumento:

```
lhp@nereida:~/Lperl/src$ perl -de 0
  DB<1> sub a($$)
  DB<2> p prototype \&a
$$
  DB<3> x prototype 'CORE::mkdir'
  0  '$;$'
```

Si el argumento es una cadena que comienza por `CORE::` se interpreta como el nombre de un operador Perl. Si el operador no se puede sobreescribir o sus argumentos no se pueden expresar como un prototipo retorna `undef`.

La función `set_prototype`

La función `set_prototype` en Scalar::Util permite establecer dinámicamente el prototipo de una función o borrarlo:

```
set_prototype CODEREF, PROTOTYPE
```

Establece el prototipo de `CODEREF` o lo borra si `PROTOTYPE` es `undef`. Returna `CODEREF`. Ejemplo:

```
set_prototype \&foo, '$$';
```

Reescribiendo `reduce` mediante Prototipos

El siguiente código implanta una función `reduce` similar a la proveída por el módulo `List::Util` presentado en la sección 1.13.2. Aunque la versión en `List::Util` es esencialmente similar en funcionalidad, es mas eficiente al estar escrita en C.

```
lhp@nereida:~/Lperl/src/hop/Chap7$ cat -n reduce.pl
#!/usr/bin/perl -w
use strict;
use Scalar::Util qw(looks_like_number);
use List::MoreUtils qw(all);

sub reduce (&@) {
  my $code = shift;
  my $val = shift;
  local ($a, $b);
  for (@_) {
    ($a, $b) = ($val, $_);
    $val = $code->()
  }
  return $val;
}

  die "\n$0 number number ...\n"
unless @ARGV and all { looks_like_number($_) } @ARGV;
my $s = reduce { $a+$b } @ARGV;
my $p = reduce { $a*$b } @ARGV;

print "Suma = $s, Prod = $p\n";
```

El uso del prototipo nos permite omitir el uso de la palabra reservada `sub` y la coma intermedia. La localización de las variables `$a` y `$b` nos permite evitar la posible contaminación de variables globales con el mismo nombre.

4.16.1. Práctica: Suma de Prefijos

Escriba una función `prefix` que funciona de manera parecida a `List::Util::reduce`: recibe una referencia a un código que describe una operación $\oplus$ y una lista $\mathcal{L} = \{x_1, \ldots x_n\}$ y devuelve una lista con los resultados de la operación sobre los prefijos de la lista: $x_1, x_1 \oplus x_2, x_1 \oplus x_2 \oplus x_3 \ldots x_1 \oplus \ldots x_n$. Por ejemplo, la llamada `prefix { $a + $b } (1, 2, 3)` retornará la lista `(1, 3, 6)`.

4.17. Las Cadenas como Ficheros

Es posible tratar una cadena como si de un fichero se tratara pasándole a `open` una referencia a la cadena:

```
open($fh, "<",  \$string);   # read only
open($fh, ">",  \$string);   # write only, discard original contents
open($fh, "+>", \$string);   # read and write, discard original contents
open($fh, "+<", \$string);   # read and write, preserve original contents
```

Si se pone `$/` a una referencia a un entero se intentará leer el número de registros referenciado por dicho entero. Por ejemplo:

```
 local $/ = \32768; # or \"32768", or \$var_containing_32768
 open my $fh, $myfile or die $!;
 local $_ = <$fh>;
```

leerá no mas de 32768 bytes desde el fichero.

El siguiente programa:

```
lhp@nereida:~/Lperl/src/testing$ cat -n stringasfile.pl
 1  use strict;
 2
 3  my $var = '*' x 100;
 4  my $max_length = 10;
 5
 6  $/ = \$max_length; # input record separator
 7  $\ = "\n";         # output record separator
 8  open my $FH, "<", \$var or die;
 9  print while <$FH>;
```

Cuando se ejecuta produce como salida:

```
lhp@nereida:~/Lperl/src/testing$ perl stringasfile.pl
**********
**********
**********
**********
**********
**********
**********
**********
**********
**********
```

4.18. Clausuras

El concepto de *clausura* es una noción bien conocida en el mundo de la programación funcional. Cuando se define una subrutina en un determinado contexto léxico, se ejecuta usando ese contexto incluso si es llamada fuera de ese contexto. Las clausuras aparecieron como concepto en los años 60 y fueron llevadas a la práctica por primera vez en un lenguaje que se llama *Scheme*.

Variables Léxicas Persistentes

En el momento de su creación una subrutina - sea anónima o no - toma los valores de las variables del ámbito en el que se encuentra. El ámbito léxico que engloba a la subrutina en el momento de su creación determina una *clausura*. Un ejemplo:

```
lhp@nereida:~/Lperl/src$ cat -n closure2.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  my $name = 1000;
 4  {
 5    my $name = $name--;
 6
 7    sub pr {
 8      $name++;
 9      return "closure: $name\n";
10    }
11  }
12
13  print $name,"\n";
14  print pr();
15  print pr();
16  print pr();
```

cuando se ejecuta produce la siguiente salida:

```
lhp@nereida:~/Lperl/src$ closure2.pl
999
closure: 1001
closure: 1002
closure: 1003
```

En condiciones normales Perl liberaría la memoria correspondiente a la variable `name` de la línea 5 al final de su ámbito. Sin embargo, la subrutina `pr` que la usa continúa visible y la variable `name` se conserva mientras la rutina este en uso. La única forma de acceder a la variable `name` de la línea 5 es a través de la subrutina `pr`.

Esta es una de las característica de una clausura: subrutinas que recuerdan el estado del entorno en el momento de su creación preservando las variables del ámbito léxico que les rodea, incluso después de la desaparición de dicho ámbito léxico.

Retornar una Subrutina

En el siguiente ejemplo la subrutina `tutu` recibe un número y retorna una subrutina que multiplica su argumento por dicho número. El programa principal aplica crea varias subrutinas y las aplica a una secuencia de números especificados:

```
lhp@nereida:~/Lperl/src$ closure3.pl 3 1 2 3 4
Multiplicar por 3: (3 6 9 12)
Multiplicar por -1: (-1 -2 -3 -4)
```

La subrutina creada dinámicamente recuerda el valor de la variable léxica `$n` en el momento de su creación.

El bucle de la línea 19 crea diferentes clausuras `$t` que acceden a diferentes instancias de la variable léxica `$n`.

```
lhp@nereida:~/Lperl/src$ cat -n closure3.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use Scalar::Util qw(looks_like_number);
 4
 5  sub tutu {
 6    my $n = shift;
 7
 8    sub {
 9      my $x = shift; die "Se esperaba un numero\n" unless looks_like_number($n);
10      return $x*$n;
11    }
12  }
13
14  ########### main #################################
15  die "Uso:\n$0 num1 num2 num3 ...\n" unless (@ARGV > 1);
16  my $n = shift; die "Se esperaba un numero\n" unless looks_like_number($n);
17
18
19  for my $f ($n, -1) {
20    my $t = tutu($f);
21    my @a = map { $t->($_) } @ARGV;
22    print "Multiplicar por $f: (@a)\n";
23  }
```

4.18.1. Clausuras y Generación de Funciones Similares

Las clausuras pueden ser un buen mecanismo para generar dinámicamente familias de funciones que se diferencian en algún parámetro.

En el siguiente ejemplo se quieren escribir funciones que faciliten la escritura de HTML, de manera que la llamada:

```
print HEAD("colores"),
      "¡Tenga ", red("cuidado"), "con esa ", green("luz!"),"\n",
      "¡Tenga ", RED("cuidado"), "con esa ", GREEN("luz!"),"\n",
      tail;
```

produciría la salida:

```
lhp@nereida:~/Lperl/src$ colors3.pl
<HTML>
<HEAD>
<TITLE>colores</TITLE>
</HEAD>
<BODY >
¡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>
</BODY>
</HTML>
```

Este es el código:

```
lhp@nereida:~/Lperl/src$ cat -n colors3.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3
 4  my @colors = qw(red blue green yellow orange purple violet);
 5
 6  for my $name (@colors) {
 7    no strict ’refs’; # permitir la manipulación de la tabla de símbolos
 8    *$name = *{uc $name} = sub { "<FONT COLOR=’$name’>@_</FONT>" };
 9  }
10
11  sub make_synonym {
12    my ($name, $sub) = @_;
13    die unless ($name =~ /[\w:]+/ && ref($sub) eq ’CODE’);
14
15    no warnings;
16    no strict ’refs’;
17    *$name = $sub;
18  }
19
20  sub tail {
21    return << ’COLA’;
22  </BODY>
23  </HTML>
24  COLA
25  }
26  make_synonym(TAIL => \&tail);
27
28  sub head {
29   my $title = shift;
30   return  << "CABEZA";
31  <HTML>
32  <HEAD>
33  <TITLE>$title</TITLE>
34  </HEAD>
35  <BODY >
36  CABEZA
37  }
38  make_synonym(HEAD => \&head);
39
40  ########### main ##############
41  print HEAD("colores"),
42        "¡Tenga ", red("cuidado"), "con esa ", green("luz!"),"\n",
43        "¡Tenga ", RED("cuidado"), "con esa ", GREEN("luz!"),"\n",
44        tail;
```

La clausura se forma en la línea 7 con la variable léxica `$name` declarada en la línea 5. La subrutina que se crea ”recuerda” el valor de `$name`. Se han usado typeglobs para instalar entradas en la tabla de símbolos a la función tanto en minúsculas como en mayúsculas.

Las funciones `head` y `tail` producen HTML para la cabecera y la cola usando *documentos aqui* (conocidos como *here-document* o *heredoc*):

4.18.2. Anidamiento de subrutinas

Ya vimos en la sección 1.15.1 que Perl no anida subrutinas como ocurre en Pascal. Sin embargo, el uso combinado de clausuras, typeglobs y `local` permite emular el efecto:

```
lhp@nereida:~/Lperl/src$ cat -n nestedsub.pl
 1  #!/usr/bin/perl
 2  use strict;
 3  use warnings;
 4
 5  sub interior {
 6    my $x = shift;
 7    print "sub marine: $x\n";
 8    $x*$x;
 9  }
10
11  sub exterior {
12    my $x = $_[0] + 3;
13    no strict ’refs’;
14    no warnings;
15    local *interior = sub { print "interior anidada: $x\n"; return $x - 1 };
16    return $x + interior();
17  }
18
19  print ’exterior(2) = ’,exterior(2),"\n";
20  print ’interior(3) = ’,interior(3),"\n"
```

En este ejemplo se ha creado en la línea 15 una subrutina denominada `interior` que ensombrece a la subrutina `interior` global definida en la línea 5 y que además tiene acceso al ámbito de `exterior`. Esta subrutina `interior` de la línea 15 es accesible solamente desde la subrutina `exterior`. El resultado de la ejecución es:

```
lhp@nereida:~/Lperl/src$ ./nestedsub.pl
interior anidada: 5
exterior(2) = 9
sub marine: 3
interior(3) = 9
```

Obsérvese que hemos omitido la opción `-w` del intérprete y hemos usado el módulo `warnings`.

Ejercicio 4.18.1. *¿Cuál es el ámbito exacto de visibilidad de una subrutina contruida como la subrutina* `interior` *de la línea 15? ¿Sería visible desde una subrutina que fuera llamada desde* `exterior` *en un punto posterior a su creación?*

4.18.3. Clausuras e Iteradores

Un *iterador* es un objeto que provee una interfaz para recorrer una lista. El iterador debe guardar la información sobre la lista y sobre la posición actual dentro de la lista.

Cuando se llama a un constructor de iteradores, el constructor retorna una subrutina que es capaz de iterar sobre los elementos de la lista.

Son ejemplos de iteradores

- Los ficheros (donde `open` crea el iterador y el operador de lectura `<>` nos da el siguiente),
- Los directorios (`opendir` es el constructor, `readdir` el iterador)

- Las cadenas cuando son sometidas a los operadores de emparejamiento y sustitución con la opción `g` (la posición del siguiente lugar de búsqueda puede ser consultada mediante el operador `pos`):

```
  DB<1 $x = 'un ejemplo sencillo'
  DB<2> print "$&(".pos($x).") " while $x =~ m{[aeiou]}g
u(1) e(4) e(6) o(10) e(13) i(16) o(19)
```

Un Ejemplo Simple de Iterador

El siguiente ejemplo muestra la construcción de un *iterador*. El generador podria usarse como sigue:

```
16  my $row = iterator 1, 4;    # creación de un iterador de 1 a 4
17  my $col = iterator 1, 5, 2; # de 1 a 5 pero con paso 2
18  my ($r, $c);
19  while ($row->($r)) {
20    print "$r: ";
21    while ($col->($c)) {
22      print "$c ";
23    }
24    print "\n";
25  }
```

Cada llamada a `iterators` 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:

```
lhp@nereida:~/Lperl/src$ perl iterators.pl
1: 1 3 5
2: 1 3 5
3: 1 3 5
4: 1 3 5
lhp@nereida:~/Lperl/src$
```

El Constructor del Iterador

El código del constructor `iterator` retorna la subrutina de iteración. Esta última controla el agotamiento de la lista en la línea 9. En tal caso se vuelve a iniciar el contador (`$next = $from-$step`) y se retorna `undef` (obsérvese el uso de la coma).

```
lhp@nereida:~/Lperl/src$ cat -n iterators.pl
 1  use strict;
 2
 3  sub iterator {
 4    my ($from, $to, $step) = @_;
 5    $step = 1 unless defined($step);
 6    my $next = $from-$step;  # iniciar contador
 7    my $closure_ref = sub {
 8      $next += $step;
 9      $next = $from-$step, return if $next > $to;
10      $_[0] = $next;
11      return 1;
12    };
13    return $closure_ref;
14  }
```

Referencias

Para saber mas sobre iteradores consulte el libro Higher Order Perl [13].

Subconjuntos de un Conjunto

Introducción

El siguiente ejemplo muestra un iterador sobre los subconjuntos de un conjunto dado. El conjunto viene definido por los argumentos en línea de comando. Sigue un ejemplo de ejecución:

```
lhp@nereida:~/Lperl/src$ perl allsubsets.pl A B C D
()
(A)
(B)
(A, B)
(C)
(A, C)
(B, C)
(A, B, C)
(D)
(A, D)
(B, D)
(A, B, D)
(C, D)
(A, C, D)
(B, C, D)
(A, B, C, D)
```

Número en Binario = Conjunto

La idea en la que se basa el iterador de conjuntos es que la representación binaria de un número puede verse como la definición de un conjunto. Los unos indican pertenencia y los ceros no pertenencia. Observe la misma ejecución anterior pero prefijado con la representación binaria del número de orden:

```
~/Lperl/src$ perl allsubsets.pl A B C D | perl -ne 'printf "%2d:%4b %s",$k,$k,$_; $k++'
 0:   0 ()
 1:   1 (A)
 2:  10 (B)
 3:  11 (A, B)
 4: 100 (C)
 5: 101 (A, C)
 6: 110 (B, C)
 7: 111 (A, B, C)
 8:1000 (D)
 9:1001 (A, D)
10:1010 (B, D)
11:1011 (A, B, D)
12:1100 (C, D)
13:1101 (A, C, D)
14:1110 (B, C, D)
15:1111 (A, B, C, D)
```

La `opción -n` de Perl

La `opción -n` de Perl inserta el siguiente bucle alrededor de nuestro programa:

```
LINE:
  while (<>) {
```

```
              ...               # nuestro programa va aqui
          }
```

Asi pues el comando `perl -ne 'printf "%2d:%4b %s",$k,$k,$_; $k++'` se convierte en:

```
        LINE:
          while (<>) {
              printf "%2d:%4b %s",$k,$k,$_; $k++
          }
```

LLamadas al Iterador

Los conjuntos son representados mediante listas. Dado que el conjunto vacío es un subconjunto de cualquier conjunto y que el conjunto vacío es representado mediante la lista vacía no podemos usar como criterio de finalización el retorno de la lista vacía.

```
lhp@nereida:~/Lperl/src$ sed -ne '20,$p' allsubsets.pl | cat -n
     1  my @S;
     2  FOREVER: {
     3    @S = $s->();
     4
     5    local $" = ', ';
     6    print "(@S)\n";
     7
     8    last if (@S == @ARGV);
     9
    10    redo FOREVER;
    11  }
```

Constructor

Sigue el código:

```
lhp@nereida:~/Lperl/src$ cat -n allsubsets.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3
 4  sub subsets {
 5    my @set = @_;
 6    my $powern = 1 << @set;
 7    my $n = -1;
 8
 9    my $s = sub {
10      $n++;
11      return map {  $n & (1 << $_)? ($set[$_]) : () } 0..$#set;
12    };
13
14    return $s;
15  }
16
17  die "$0 el1 el2 el3 ...\n" unless @ARGV;
18  my $s = subsets(@ARGV);
19
20  my @S;
21  FOREVER: {
22    @S = $s->();
23
```

```
24    local $" = ', ';
25    print "(@S)\n";
26
27    last if (@S == @ARGV);
28
29    redo FOREVER;
30  }
```

Práctica: Iterador de Archivos

Construya un iterador `dir_walk` que recorra los arboles de directorios de sus argumentos generando un fichero de cada vez. Observe el modo de uso y el resultado de una ejecución:

```
lhp@nereida:~/Lperl/src/hop/Chap4$ cat -n dirwalk.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3
 4  sub dir_walk {
 .  ..............
19  }
20
21  my $d = dir_walk(@ARGV);
22
23  my $f;
24  while ($f = $d->()) {
25    print "$f\n";
26  }
lhp@nereida:~/Lperl/src/hop/Chap4$ ./dirwalk.pl /tmp/Parallel-Simple-Pipe-0.01
/tmp/Parallel-Simple-Pipe-0.01
/tmp/Parallel-Simple-Pipe-0.01/script
/tmp/Parallel-Simple-Pipe-0.01/t
/tmp/Parallel-Simple-Pipe-0.01/META.yml
/tmp/Parallel-Simple-Pipe-0.01/lib
/tmp/Parallel-Simple-Pipe-0.01/Changes
/tmp/Parallel-Simple-Pipe-0.01/MANIFEST
/tmp/Parallel-Simple-Pipe-0.01/Makefile.PL
/tmp/Parallel-Simple-Pipe-0.01/README
/tmp/Parallel-Simple-Pipe-0.01/script/test2.pl
/tmp/Parallel-Simple-Pipe-0.01/script/test.pl
/tmp/Parallel-Simple-Pipe-0.01/t/Parallel-Simple-Pipe.t
/tmp/Parallel-Simple-Pipe-0.01/lib/Parallel
/tmp/Parallel-Simple-Pipe-0.01/lib/Parallel/Simple
/tmp/Parallel-Simple-Pipe-0.01/lib/Parallel/Simple/Pipe.pm
/tmp/Parallel-Simple-Pipe-0.01/lib/Parallel/Simple/Communicator.pm
lhp@nereida:~/Lperl/src/hop/Chap4$
```

Nota: para que el operador `-d` funcione es necesario que los nombres de fichero especifiquen también el camino al archivo. No basta con el nombre de base. Sin embargo, lo que devuelve `readdir` es el nombre base.

```
lhp@nereida:~/Lperl/src$ perl -de 0
main::(-e:1):   0
  DB<1> opendir $d,'/tmp/Parallel-Simple-Pipe-0.01'
  DB<2> x scalar(readdir $d)
0  '.'
```

```
  DB<3> x scalar(readdir $d)
0  '..'
  DB<4> x scalar(readdir $d)
0  'script'
  DB<5> x scalar(readdir $d)
0  't'
  DB<6> x scalar(readdir $d)
0  'META.yml'
  DB<7> x -d 't'
0  undef
  DB<8> x -d '/tmp/Parallel-Simple-Pipe-0.01/t'
0  1
```

Como se ve en las líneas 2 y 3 de la sesión `readdir` retorna las referencias al directorio actual y al directorio padre. Tenga esto en cuenta apartando estos dos directorios para evitar la posibilidad de una recursión infinita.

Si lo considera necesario, repase las secciones sobre el manejo de directorios y ficheros 2.5, 2.9 y 2.10.

4.18.4. 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. Si una función pura es llamada muchas veces con los mismos argumentos es posible acelerar la ejecución de la misma utilizando una estrategia denominada *memoizing*: Se usa una cache para guardar la correspondencia entre los argumentos y los resultados. Habitualmente esta cache es un hash, en el que las claves son los argumentos y los valores los resultados. La variable usada como cache se sitúa en clausura con la subrutina. Para saber mas sobre memoizing consulte el libro de Mark Jason Dominus Higher Order Perl [13]. El módulo `Memoize`, escrito por Dominus, proporciona memoización automática de una función.

El siguiente ejemplo *memoiza* la función que computa los números de Fibonacci:

```
lhp@nereida:~/Lperl/src$ cat -n memoize2.pl
 1  #!/usr/bin/perl -w
 2  use Benchmark;
 3  use Memoize;
 4
 5  sub fib {
 6    my $n = shift;
 7    if ($n < 2) { $n }
 8    else { fib($n-1)+fib($n-2) }
 9  }
10
11  {
12
13  my @fib = (0, 1);
14
15    sub fibm {
16      my $n = shift;
17
18      return $fib[$n] if defined $fib[$n];
19      $fib[$n] = fibm($n-1)+fibm($n-2);
20    }
21  }
22
23  sub fibM {
```

```
24    my $n = shift;
25    if ($n < 2) { $n }
26    else { fib($n-1)+fib($n-2) }
27  }
28
29  memoize 'fibM';
30
31  my ($r, $m, $a);
32  timethese(2, {
33    recursivo => sub { $r = &fib(30) },
34    memoized  => sub { $m = &fibm(30) },
35    automemoized  => sub { $a = &fibM(30) }
36    }
37  );
38
39  print "recursivo = $r, memoized = $m, automemoized = $a\n";
```

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:

```
lhp@nereida:~/Lperl/src$ memoize2.pl
Benchmark: timing 2 iterations of automemoized, memoized, recursivo...
automemoized:  3 wallclock secs ( 2.98 usr +  0.00 sys =  2.98 CPU) @  0.67/s (n=2)
            (warning: too few iterations for a reliable count)
  memoized:  0 wallclock secs ( 0.00 usr +  0.00 sys =  0.00 CPU)
            (warning: too few iterations for a reliable count)
 recursivo:  6 wallclock secs ( 5.97 usr +  0.00 sys =  5.97 CPU) @  0.34/s (n=2)
            (warning: too few iterations for a reliable count)
recursivo = 832040, memoized = 832040, automemoized = 832040
```

Práctica: Memoización de un Divide y Vencerás

El problema de la mochila 0-1 se enuncia asi: Se tiene una mochila de de capacidad C 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.

Denotemos por $knap_{k,c,b}$ el valor óptimo de la solución para el subproblema considerando los objetos $\{1, \ldots k\}$, capacidad c y beneficio inicial b. El problema consiste en encontrar el valor de $knap_{N,C,0}$. Se cumple la siguiente fórmula:

$$knap_{k,c,b} = \text{máx}\{knap_{k-1,c,b}, knap_{k-1,c-w_k,b+p_k}\} \tag{4.1}$$

si $c > w_k$ y $knap_{\emptyset,c,b} = b$ en otro caso.

Esta fórmula nos dice que la solución para el problema $knap_{k,c,b}$ con objetos $\{1, \ldots k\}$ puede obtenerse a partir de las soluciones de los dos subproblemas de mochila con objetos $\{1, \ldots k-1\}$ que resultan de tomar las decisiones de incluir o no el objeto k.

El siguiente programa recibe el número de objetos y genera un problema aleatorio de mochila 0-1 con ese número de objetos:

```
lhp@nereida:~/Lperl/src$ time ./memoizeknap.pl 20
C = 115; w = 17 18 4 14 13 13 8 16 17 15 10 9 3 14 9 3 8 15 7 18
115

real    0m3.043s
user    0m3.050s
```

```
sys     0m0.010s
```

El programa usa la ecuación 4.1 para resolver el problema. El diseño recursivo de la subrutina da lugar a un esquema *divide-y-vencerás*:

```
lhp@nereida:~/Lperl/src$ cat -n memoizeknap.pl
#!/usr/bin/perl -w
use strict;
use List::Util qw(max sum);

my $N = shift || 10;
my @w = map { 3 + int(rand($N-3)) } 1..$N;
my @p = @w;
my $C = int(sum(@w)/2);

  sub knap {
    my ($k, $c, $b) = @_;
    my @s;

    return $b if $k < 0;
    $s[0] = knap($k-1, $c, $b);
    if ($w[$k] <= $c) {
      $s[1] = knap($k-1, $c-$w[$k], $b+$p[$k]);
    }
    return max(@s);
  }

my $s = knap($N-1, $C, 0);

print "C = $C; w = @w\n";
print "$s\n";
```

Responda a las siguientes preguntas:

1. ¿Cual es el tamaño del árbol de búsqueda explorado por `knap` para un problema de la mochila con N objetos?. La pregunta se refiere al peor caso posible.
2. Para una mochila de capacidad C y N objetos de pesos w_k y beneficios p_k ¿Cuál es el máximo cardinal del conjunto de ternas de entrada `(k, c, b)` de la subrutina `knap`? ¿Que valor es mayor. Este o el calculado en el apartado anterior?
3. Memoize usando el módulo `Memoize` la función `knap`. Use el módulo Benchmark para comparar rendimientos. ¿Puede explicar las diferencias? ¿Como cambia el árbol de búsqueda explorado al memoizar la función?
4. Memoize manualmente la función `knap`. Use el módulo Benchmark para comparar los rendimientos de las tres versiones.

Práctica: La Función `memoize`

El siguiente código muestra una forma de obtener un ”memoizador” de propósito general:

```
sub memoize {
  my $func = shift;

  my %cache;
  my $stub = sub {
```

```
    my $key = join ',', @_;
    $cache{$key} = $func->(@_) unless exists $cache{$key};
    return $cache{$key};
  }
  return $stub;
}
```

En las secciones 4.15.9 y 4.15.10 vimos como usando typeglobs selectivos podemos instalar en la tabla de símbolos un *wrapper* de una función dada. Nótese que la funcionalidad del módulo `Memoize` cae dentro de la categoría del *wrapping*. Utilice la misma técnica para sustituir una función por su memoizada.

4.18.5. Currying

Se denomina *currying* 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`. Para saber mas sobre currying consulte el libro Higher Order Perl [13]. El término currying hace alusión al lógico americano *Haskell Curry* (1900-1982) que popularizó el concepto en 1930. Al parecer *Gottlob Frege* ya había introducido la idea en 1893. *Moses Schönfinkel* la redescubrió en 1924.

Ejercicio 4.18.2. *Describa en palabras los dominios inicial y final de las funciones:*

1. *La función* `map`*.*
2. *La función* `splice`
3. *La función* `first`*: Consulte la documentación del módulo* `List::Util`*.*

Una Función `curry`

El siguiente ejemplo muestra una función `curry` que recibe como argumento una función

$$f : D_1 \times \ldots \times D_n \to E$$

y retorna una función que currifica esa función:

$$curry(f) : D_1 \to \mathcal{F}(D_2 \times \ldots \times D_n, E)$$

donde $\mathcal{F}(D_2 \times \ldots \times D_n, E)$ denota las funciones con argumentos en $D_2 \times \ldots \times D_n$ y valores en E.

Cuando a la función $curry(f)$ se le pasa un argumento adicional x se obtiene una nueva función:

$$curry(f)(x) : D_2 \times \ldots \times D_n \to E$$

Sigue un ejemplo de uso:

```
die "Error\n" unless @ARGV;
my $cprod = curry(sub { $_[0]*$_[1] });
my $doble = $cprod->(2);
my @dobles = map { $doble->($_) } @ARGV;
print "@dobles\n";

my $cfirst = curry(\&first);
my $fodd = $cfirst->( sub { $_ % 2 } ); # Primer impar
print $fodd->(@ARGV)."\n";
```

```
lhp@nereida:~/Lperl/src$ currying2.pl 4 8 12 9 23 25
8 16 24 18 46 50
9
```

Veamos el código y una ejecución:

```
lhp@nereida:~/Lperl/src$ cat -n currying2.pl
#!/usr/bin/perl -w
use strict;
use List::Util qw(first);

sub curry {
  my $f = shift; # Referencia a una función de dos o mas argumentos

  return sub {  # Retorna función como f pero con un arg. menos
    my $x = shift;
    my $fx = sub { $f->($x, @_) };
    return $fx;
  };
}
```

La técnica de currificación transforma una función ordinaria en una *fábrica de funciones* (*function factory*). Al llamar a la fábrica con un parámetro nos devuelve una nueva función.

Ejercicio 4.18.3. *¿Que describe la expresión* `curry(\&List::Util::sum)->(4)->(1..5)`*? Si tiene dudas lea la siguiente sesión con el depurador:*

```
lhp@nereida:~/Lperl/src$ perl -wd currying2.pl
main::(currying2.pl:15):        die "Error\n" unless @ARGV;
  DB<1>  $f = \&List::Util::sum
  DB<2> $g = curry($f)
  DB<3> $h = $g->(4)
  DB<4> x $h->(1..5)
0  19
```

Currificar `map`

Veamos otro ejemplo de currificación. La función `map` es una función de dos argumentos: el primero es código y el segundo una lista. La función retorna una lista. Matemáticamente:

$$Map : \mathcal{F}(D,E) \times \mathcal{L}(D) \to \mathcal{L}(E)$$

donde $\mathcal{F}(D,E)$ denota las funciones de D en E y $\mathcal{L}(D)$ denota las listas de objetos de tipo D.

Si la currificamos la convertiremos en una función de un sólo argumento que retorna funciones:

$$Cmap : \mathcal{F}(D,E) \to \mathcal{F}(\mathcal{L}(D), \mathcal{L}(E))$$

Así pues $Cmap(f)$ - para una función f dada - es una función que actúa sobre una lista y retorna una lista. Podríamos usarla como sigue:

```
lhp@nereida:~/Lperl/src/hop/Chap7$ sed -ne ’11,$p’ cmap2.pl | cat -n
my $sizes  = cmap { -s $_ };
my $rand20 = cmap { int(rand(20)) };
local $a = 0;
my $pref   = cmap { $_ += $a; $a = $_ };

my @s = $sizes->(@ARGV);
my @r = $rand20->(1..5);
my @p = $pref->(1..5);

print "Tamaños = @s\n";
```

```
11  print "Aleatorio = @r\n";
12  print "Suma de prefijos = @p\n";
```

que al ejecutra resulta en:

```
lhp@nereida:~/Lperl/src/hop/Chap7$ cmap2.pl *.pm
Tamaños = 650 1933 591 1713
Aleatorio = 18 11 1 8 12
Suma de prefijos = 1 3 6 10 15
```

Veamos una implementación:

```
lhp@nereida:~/Lperl/src/hop/Chap7$ sed -ne '1,9p' cmap2.pl | cat -n
1  #!/usr/bin/perl -w
2  use strict;
3
4  sub cmap (&) {
5    my $f = shift;
6    return sub {
7      map { $f->($_) } @_
8    };
9  }
```

Práctica: Currificación de `grep`

Siguiendo la estrategia de currificación presentada para la función `map` escriba el código para la currificación de la función `grep`.

4.18.6. Listas Perezosas

La *evaluación perezosa* es una estrategia de cómputo que intenta retrasar los cálculos de las expresiones hasta que sean realmente necesarios. Los iteradores vistos en la sección 4.18.3 pueden considerarse una forma de evaluación perezosa: en vez de generar una lista potencialmente enorme de objetos (permutaciones, subconjuntos, o cualesquiera que sean) sólo se genera un item de cada vez. Los items que no sean usados por el programa no serán nunca generados y no se malgastará tiempo y espacio en ellos.

Otro ejemplo de evaluación perezosa lo constituyen las *listas perezosas*. Una lista enlazada perezosa es casi igual que una lista enlazada ordinaria. Cada nodo esta formado por una parte de información *y un enlace que permite acceder al siguiente nodo de la lista*. No hay diferencia señalable de las subrutinas `node` y `head` del siguiente código con el que escribiríamos para implantar una lista enlazada ordinaria. La diferencia está en la cola de la lista (accedida mediante la subrutina `tail`). La cola de la lista es una referencia a un código que - cuando la cola es accedida - permite calcular el siguiente elemento de la lista. Una cola se denomina una *promesa* o *promise*. La función `tail` comprueba si la cola es una promesa (i.e. una referencia a código) y si es así pasa a computar el siguiente nodo mediante la función referenciada.

El siguiente código crea una lista (perezosa) conteniendo las líneas de un fichero hasta la primera línea que casa con una expresión regular dada como argumento.

La subrutina `file2lazylist` abre el fichero y retorna la función que será usada como promesa. Esta función trabaja en clausura guardando el acceso al manejador de fichero `$fh` (que también es retornado).

```
lhp@nereida:~/Lperl/src/hop/Chap6$ cat -n useStreams1.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3  use Carp;
 4  use IO::File;
```

```

sub node($$) {
  my ($h, $t) = @_;
  [$h, $t];
}

sub head {
  my ($s) = @_;
  $s->[0];
}

sub tail {
  my ($s) = @_;
  if (is_promise($s->[1])) {
    $s->[1] = $s->[1]->();
  }
  $s->[1];
}

sub is_promise {
  ref($_[0]) eq 'CODE';
}

sub show_until_match {
  my ($s, $regexp) = @_;
  while ($s and head($s) !~ /$regexp/) {
    print head($s);
    $s = tail($s);
  }
  print head($s) if $s;
}

sub file2lazylist {
  my $fn = shift;
  my $fh = new IO::File;
  $fh->open("< $fn") or carp "No se pudo abrir $fn: $!";

  my $nl;
  $nl = sub {
    return unless defined($_ = <$fh>);
    return node($_, $nl);
  };
  return ($fh, $nl)
}

my $file = shift;
my $regexp = shift;
my ($fh, $fl) = file2lazylist($file);
my $node = $fl->();
show_until_match($node, $regexp);
close($fh);
```

Al ejecutar el código anterior obtendremos una salida similar a esta:

```
lhp@nereida:~/Lperl/src/hop/Chap6$ useStreams1.pl useStreams.pl '\(.*\)'
```

```
#!/usr/bin/perl -w
use strict;
use Carp;
use IO::File;

sub node($$) {
```

Para saber mas sobre listas perezosas consulte el libro Higher Order Perl [13].

Práctica: Listas Perezosas

Dados dos números n y m con $m > n$, escriba una función que genera una lista perezosa con los números entre n y m.

Capítulo 5

Módulos

5.1. Los packages: Repaso

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";
}

# ...salimos del paquete C110
package D110;
# ahora estamos en el espacio de nombres D110

$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`

5.2. 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
```

Las Variables Especiales Pertenecen a `main`

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
```

Variables Léxicas y Scratchpads

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*. 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.

Ejercicio 5.2.1. *Explique la salida del siguiente programa:*

```
lhp@nereida:~/Lperl/src$ cat -n privacy.pl
#!/usr/bin/perl -w
use strict;
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 = $a\n";
```

5.3. Subrutinas Privadas

Ejercicio 5.3.1.

- *Estudie el programa y explique la salida a la que da lugar. La subrutina* `peer` *definida en la línea 22 llama a la subrutina* `interior`*. ¿Podría identificar que subrutina es llamada? ¿La de la línea 5 o la de la línea 14?*

```
lhp@nereida:~/Lperl/src$ cat -n localsub.pl
 1  #!/usr/bin/perl
 2  use strict;
 3  use warnings;
 4
 5  sub interior {
 6    my $x = shift;
 7    print "En sub 'interior global': $x\n";
 8  }
 9
10  sub exterior {
11    my $x = $_[0];
12    no strict 'refs';
13    no warnings;
14    local *interior = sub {
15      print "En sub interior anidada\n";
16      return shift() - 1
17    };
18    print "En sub exterior. Llamo a la sub 'peer(5)'\n";
19    peer(5);
20  }
21
22  sub peer {
23    print "En sub peer. Llamo a 'interior'\n";
24    interior(shift())."\n";
25  }
26
27  exterior(2);
28  interior(3);
```

Al ejecutar el programa `localsub.pl` *se produce la siguiente salida:*

```
lhp@nereida:~/Lperl/src$ localsub.pl
En sub exterior. Llamo a la sub 'peer(5)'
En sub peer. Llamo a 'interior'
```

```
En sub interior anidada
En sub 'interior global': 3
```

- *Tengo una subrutina en un package que no quiero que sea visible desde otros paquetes. ¿Que puedo hacer para conseguirlo? (Recuerde que las subrutinas son siempre globales).*
- *Explique cual será la salida del siguiente programa (la macro* `__PACKAGE__` *retorna el nombre del paquete actual):*

```
lhp@nereida:~/Lperl/src$ cat -n  ./privatesub.pl
 1  #!/usr/bin/perl
 2  use strict;
 3  use warnings;
 4
 5  package One;
 6  sub tutu {
 7    my $x = shift;
 8    print "En sub 'One:tutu ': $x\n";
 9  }
10
11  package Two;
12  sub tutu {
13    my $package = caller;
14      die "Attempt to call private function Two:tutu"
15    unless ($package eq __PACKAGE__);
16
17    my $x = shift;
18    print "En sub 'Two:tutu ': $x\n";
19  }
20
21  package main;
22  One::tutu(3);
23  Two::tutu(5);
```

5.4. Paquetes y Ficheros

¿Un Paquete por Fichero?

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.

Sufijos

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`.

Cargando una Librería con `require`

Los programas que quieren usar las subrutinas en la librería pueden invocar el fichero con `require`

```
lhp@nereida:/tmp$ perl -wde 0
main::(-e:1):   0
  DB<1> require GRID::Machine    # Cargada

  DB<2> require "GRID::Machine"  # Error
Can't locate GRID::Machine in @INC (@INC contains:
/etc/perl
```

```
/usr/local/lib/perl/5.8.8
/usr/local/share/perl/5.8.8
/usr/lib/perl5
/usr/share/perl5
/usr/lib/perl/5.8
/usr/share/perl/5.8
/usr/local/lib/site_perl
/usr/local/lib/perl/5.8.7
/usr/local/share/perl/5.8.7
/usr/local/lib/perl/5.8.4
/usr/local/share/perl/5.8.4.)
at (eval 17)[/usr/share/perl/5.8/perl5db.pl:628] line 2.

  DB<3> require "GRID/Machine.pm" # Cargada

  DB<4> x %INC
0  're.pm'
1  '/usr/lib/perl/5.8/re.pm'
2  'attributes.pm'
3  '/usr/share/perl/5.8/attributes.pm'
..  ................................................
42  'GRID/Machine.pm'
43  '/usr/local/share/perl/5.8.8/GRID/Machine.pm'
..  ................................................
60  'GRID/Machine/MakeAccessors.pm'
61  '/usr/local/share/perl/5.8.8/GRID/Machine/MakeAccessors.pm'
62  'Module/Which.pm'
63  '/usr/local/share/perl/5.8.8/Module/Which.pm'
64  'GRID/Machine/Result.pm'
65  '/usr/local/share/perl/5.8.8/GRID/Machine/Result.pm'
..  ................................................
110  'Term/ReadLine/Gnu.pm'
111  '/usr/lib/perl5/Term/ReadLine/Gnu.pm'
```

la función `require` lee el archivo si este no ha sido leido antes.

En el hash especial `%INC` están las librerías cargadas. Obsérvense las líneas 42 y 43.

El módulo Module::Util proporciona la función `module_path` que devuelve el camino relativo para un módulo. Así si se quiere saber donde está un módulo:

```
  DB<1> use GRID::Machine
  DB<2> use Module::Util qw( :all );
  DB<3> x $INC{module_path('GRID::Machine')}
0  '/soft/perl5lib/share/perl/5.8.8//GRID/Machine.pm'
```

o bien:

```
  DB<4> x module_is_loaded('GRID::Machine')
0  '/soft/perl5lib/share/perl/5.8.8//GRID/Machine.pm'
```

El archivo cargado no tiene que necesariamente estar asociado con un `package`. Su evaluación por `require` debe devolver verdadero. Por ello la última sentencia ejecutada deberá devolver una valor cierto. De aquí que un paquete normalmente termine con:

`return 1;`

Si se omiten las comillas y el sufijo, se asume una extensión `.pm`:

```
lhp@nereida:/tmp$ perl -wde 0
main::(-e:1):   0
  DB<1> require "CGI"    # Error
Directory /usr/local/share/perl/5.8.8/CGI not allowed in require at (eval 5)[/usr/share/perl/5.8/perl5db.p
  DB<2> require CGI      # OK
  DB<3> require "CGI.pm" # OK
```

”`use`” Ocurre Durante la Compilación

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`

5.5. Búsqueda de Librerías y Módulos

La Lista de Caminos de Módulos: `@INC` Cuando se compila una sentencia `use module` se produce la búsqueda por el fichero conteniendo `module`.

El fichero conteniendo el módulo o librería debe colocarse en uno de varios directorios estándar en los que el compilador busca. 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 Significado de `::`

El compilador sustituye cada `::` por el separador de caminos. Asi la orden: `use Text::ParseWords;` se traduce por el fichero `Text/ParseWords.pm`. En cierta máquina el directorio exacto podría ser algo similar a `/usr/lib/perl5/5.00503/Text/ParseWords.pm`

La Evaluación de `use Text::ParseWords`

El compilador Perl abre el primer fichero en el camino de búsqueda 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`.

Añadir Directorios de Búsqueda

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 .
   ```

Modificar `@INC` en Tiempo de Compilación

Si se quiere garantizar que el camino en cuestión esta disponible antes de que se ejecute ninguna sentencia se puede usar una de las opciones 1 2 o 4. También podemos hacerlo rodeando la opción 3 de un `BEGIN`:

```
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.

Ejercicio 5.5.1. *Explique la siguiente salida:*

```
casiano@beowulf:~/src/perl$ perl -de 0
DB<1> require Data::Dumper

DB<2> require "Data::Dumper"
Can't locate Data::Dumper in @INC (@INC contains: /etc/perl
/usr/local/lib/perl/5.8.8 /usr/local/share/perl/5.8.8
/usr/lib/perl5 /usr/share/perl5 /usr/lib/perl/5.8
/usr/share/perl/5.8 /usr/local/lib/site_perl .) at (eval
18)[/usr/share/perl/5.8/perl5db.pl:628] line 2.
```

Observe que el primer `require` *no produce ninguna queja, el segundo si. Consulte* `perldoc -f require` *para entender este detalle.*

5.6. 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 6.5). La rutina por defecto comprueba el valor en la variable `$VERSION` del paquete en cuestión.

Para conocer la versión de un módulo podemos escribir:

```
$ perl -MMemoize -e 'print UNIVERSAL::VERSION('Memoize')."\n"'
1.01
```

o bien:

```
nereida:~/Lparrot/examples/shootout> perl -MMemoize -e 'print "$Memoize::VERSION\n"'
1.01
```

La `opción -M` permite cargar el módulo especificado como argumento.

5.7. Importación

Que es Importar

A menudo queremos *importar* ciertos símbolos de un módulo en nuestro espacio de nombres, para ahorrarnos pulsaciones: queremos escribir `sqrt` y no `math::sqrt`.

La Subrutina `import`

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; module::import(module::import, 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.

Argumentos de `import`

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");
```

Aprenda a Decir `no`

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`.

Ejemplo

Supongamos que 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
 1  #!/usr/bin/perl -w -I.
 2  use strict;
 3  use myimport;
 4
 5  my $titi = 4;
 6  my @titi = (1,2,3);
 7  &titi();
 8
 9  print"$titi\n";
10  print"@titi\n";
```

de lugar a la salida:

```
$ ./usemyimport.pl
Hola
4
1 2 3
```

Para ello se tiene encuenta los siguientes puntos:

1. Hay que instalar en la tabla de símbolos del paquete llamador (que no forzosamente es el paquete `main`) una entrada `titi`.
2. Para instalar `titi` hay que averiguar el nombre del paquete llamador. La función `caller` devuelve el "package" desde el cuál fue llamada la subrutina (sección 1.15.10).
3. Para instalar la entrada `titi` en la tabla de símbolos del paquete llamador hay que usar typeglobs y referenciado simbólico.
4. Minimizamos el periodo de desactivación de control estricto del referenciado simbólico (`no strict 'refs'`).
5. 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`.

```
lhp@nereida:~/Lperl/src$ cat -n myimport.pm
 1  package myimport;
 2  use strict;
 3
 4  sub titi {
 5    print "Hola\n";
 6  }
 7
 8  sub import {
 9
10    my ($caller_package) = caller;
11    {
12      no strict 'refs';
13      *{$caller_package."::titi"} = \&titi;
14    }
15  }
16
17  1;
```

Ejercicio 5.7.1. *Explique que ocurre si cambiamos la línea 13 por*

```
*{"$caller_package::titi"} = \&titi;
```

¿Seguirá funcionando?

5.8. Acceso a la tabla de símbolos

¿Que es un Stash?

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 en inglés 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.

Ejemplo

Consideremos el siguiente código que contiene dos paquetes: `main` y `toto`:

```
$ cat -n 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";
```

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
```

5.8.1. 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*.

Por ejemplo, dado el paquete `Test`:

```
lhp@nereida:~/Lperl/src/advanced_perl_programming/Typeglob$ cat -n dumpvar.pl
41  package Test;
42  our $x = 10;
43  our @y = (1,3,4);
44  our %z = (1,2,3,4, 5, 6);
45  our $z = 300;
46  our $f = sub { 1; };
```

la llamada:

```
48  DumpPackage::dumppackage("Test");
```

producirá una salida similar a:

```
lhp@nereida:~/Lperl/src/advanced_perl_programming/Typeglob$ dumppackage.pl
y
        Lista @y: [
          1,
          3,
          4
        ]
=============================
```

```
__ANON__
=============================
x
        Escalar $x:     10
=============================
f
        Escalar $f: sub {
            package Test;
            use strict 'refs';
            1;
        }
=============================
z
        Escalar $z:     300
        Hash %z: {
          1 => 2,
          3 => 4,
          5 => 6
        }
=============================
```

Para imprimir una estructura de datos compleja existen varias soluciones. Puede usar el módulo `Data::Dumper` para volcar las estructuras de datos resultantes. Consulte la documentación del módulo.

Puede partir del esqueleto que sigue. Las líneas de puntos indican lugares en los que deberá insertar el código apropiado.

```
lhp@nereida:~/Lperl/src/advanced_perl_programming/Typeglob$ cat -n dumppackage.pl
 1  #!/usr/local/bin/perl -w
 2  use strict;
 3
 4  package DumpPackage;
 5  use Data::Dumper;
 6
 7  $Data::Dumper::Terse = 1;
 8  $Data::Dumper::Deparse = 1;
 9
10  sub dumpackage {
11      my ($packageName) = @_;
12      my $rPackage;
13      {
14        .................
15        $rPackage = \%{"${packageName}::"};  # Obtener la referencia al stash
16      }
17      $, = " ";
18      while (my ($varName, $globValue) = each %$rPackage) {
19          print "$varName\n";
20          our ($..., @..., %...);
21          *var = $globValue; # Crear un alias
22          if (defined ($var)) {
..              .................................
25          }
26          if (defined (@var)) {
..              .................................
30          }
```

```
31          if (defined (%var)) {
..              ..................................
35          }
36          print "============================ \n";
37      }
38  }
```

5.9. AUTOLOAD: Captura de LLamadas

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.

Jerarquía de Ficheros

La jerarquía de directorios de nuestra aplicación es:

```
lhp@nereida:~/Lperl/src/systemcommand$ pwd
/home/lhp/Lperl/src/systemcommand
lhp@nereida:~/Lperl/src/systemcommand$ tree
.
|-- lib
|   `-- System
|       `-- Commands.pm
`-- script
    `-- usesystemcommand.pl

3 directories, 2 files
```

El Módulo

El módulo `System::Commands` proporciona una interfaz funcional a los comandos del sistema operativo:

```
lhp@nereida:~/Lperl/src/systemcommand$ cat -n lib/System/Commands.pm
 1  package System::Commands;
 2  use warnings;
 3  use strict;
 4  use File::Which qw(which);
 5  use List::Util qw{first};
 6
 7  my @ALLOWED;
 8
 9  our $AUTOLOAD;
10  sub AUTOLOAD {
11    $AUTOLOAD =~ m{.*::(\w+)$};
12    my $command = $1;
13
14    die "Error! sub $AUTOLOAD does not exists!\n" unless first { $command eq $_ } @ALLOWED;
15    no strict 'refs';
16    *{$AUTOLOAD} = sub {
```

```
    return `$command @_`;
  };

  goto &{$AUTOLOAD};

}

sub import {
  my $mypackage = shift;
  push @ALLOWED, @_;

  my ($caller_package) = caller;
  {
    no strict 'refs';
    for my $command (@_) {
      # Comprobar si existe el comando
      die "Error! '$command' command does not exists!\n" unless which($command);
      *{$caller_package."::".$command} = \&{$command};
    }
  }
}

1;
```

El Cliente

Veamos el programa cliente:

```
lhp@nereida:~/Lperl/src/systemcommand$ cat -n script/usesystemcommand.pl
#!/usr/bin/perl -I../lib -w
use strict;
use System::Commands qw{ls};

my $f = shift || 'pe*.pl';

print "\n******Contexto de lista*********\n";
my @files = ls("-l", "-t", "-r", $f);
print @files;

print "\n******Contexto escalar*********\n";
my $files = ls("-l", "-t", "-r", $f);
print $files;

print "\n******No existe*********\n";
my @r = chuchu("a");
```

Ejecución

Sigue una ejecución:

```
lhp@nereida:~/Lperl/src/systemcommand$ cd script/
lhp@nereida:~/Lperl/src/systemcommand/script$ usesystemcommand.pl

******Contexto de lista*********
ls: pe*.pl: No existe el fichero o el directorio
```

```
******Contexto escalar*********
ls: pe*.pl: No existe el fichero o el directorio

******No existe*********
Undefined subroutine &main::chuchu called at ./usesystemcommand.pl line 16.
```

Clausura y Memoria

La siguiente sesión con el depurador muestra el interés de la línea

```
push @ALLOWED, @_;
```

y de tener `@ALLOWED` declarada como una variable léxica en el ámbito del fichero:

```
lhp@nereida:~/Lperl/src/systemcommand/script$ perl -wde 0
main::(-e:1):   0
  DB<1> push @INC, '../lib'
  DB<2> use System::Commands qw{ls}
  DB<3> x ls('*.pl')
0  'usesystemcommand.pl
'
  DB<4> use System::Commands qw{ps}
  DB<5> p ps
  PID TTY          TIME CMD
 1632 pts/18   00:00:00 su
 1633 pts/18   00:00:00 bash
20981 pts/18   00:00:00 perl
20988 pts/18   00:00:00 ps
  DB<6> p ls
usesystemcommand.pl
  DB<7> p echo('Good morning')
Undefined subroutine &main::echo called at (eval 15)[/usr/share/perl/5.8/perl5db.pl:628] line 2.
```

5.10. Práctica: AUTOLOAD

Escriba un módulo `HTML::Tags` y su programa cliente `usehtmltags.pl`. Para cada `tag` HTML de la forma

```
<tag attr1="val1" attr2="val2"> texto </tag>
```

el módulo provee una función `tag` que cuando es llamada como sigue:

```
tag('texto', attr1 => 'val1', attr2 => 'val2')
```

retorna el texto HTML anterior. Así por ejemplo, la llamada:

```
A("Visite nuestra página", href=>"nereida.deioc.ull.es");
```

debería devolver la cadena:

```
<A href="nereida.deioc.ull.es">Visite nuestra página</A>
```

La llamada:

```
font("Alerta!", color=>red);
```

devolverá:

```
<font color="red">Alerta!</font>
```

etc.

Necesitará una función `AUTOLOAD` para realizar esta práctica.

Para la subrutina generada utilize la estrategia de paso de parámetros a través de un hash para tener argumentos con nombres (véase la sección 1.15.7)

Las funciones creadas serán insertadas en el paquete/espacio de nombres especificado como argumento de la directiva `use HTML::Tags`, como se muestra en el siguiente ejemplo:

```
use HTML::Tags 'MyHTML';

{
  package MyHTML;
    my $t =
      html(
        head(title('Hello World!),
        body(
          h1('Hello World!'),
          p,hr,p,
        )
     )
}
# de nuevo en package main

print "$t\n";
```

Si no se especifica un argumento a `use HTML::Tags` se utilizará el espacio de nombres `HTML::Tags`.

5.11. El Pragma `use subs`

Referenciado de Operadores No es posible referenciar los operadores pre-construidos en Perl:

```
lhp@nereida:~/Lperl/src$ cat reftooperator.pl
#!/usr/local/bin/perl -w
use strict;

my $a = \&glob;
print $a->('*');
lhp@nereida:~/Lperl/src$ perl reftooperator.pl
Undefined subroutine &main::glob called at reftooperator.pl line 5.
```

Los nombres de los operadores no están en una tabla de símbolos de un paquete particular.

El Pragma `use subs`

Se produce una ambiguedad cuando en un paquete se declara una subrutina cuyo nombre coincide con el de un operador Perl. Si posteriormente dicho nombre es usado sin usar una especificación completa Perl optará por interpretar que se llama al operador Perl y no a la subrutina. Para forzar la llamada a la subrutina se puede usar un ampersand o usar el nombre completo de la subrutina.

El pragma `use subs` permite una solución alternativa. Los operadores originales de Perl serán sustituidos por nuestras propias versiones en el ámbito del paquete. En el siguiente programa sustituímos `glob` en el ámbito del paquete `Glob::Regexp`:

```
lhp@nereida:~/Lperl/src$ cat -n usesubs.pl
 1  #!/usr/local/bin/perl -w
 2  package Glob::Regexp;
```

```
 3  use strict;
 4  use subs qw(glob);
 5
 6  sub glob {
 7    my $regexp = shift;
 8
 9    opendir my $DIR, "." or die "$!";
10    my @files = grep /$regexp/, readdir $DIR;
11    close($DIR);
12    return @files;
13  }
14
15  sub tutu {
16    my @a = glob(’^.[cd][ba].*\.pl’); # LLamada al nuevo glob
17    local $" = "\n";
18    my @b = <^.[cd][ba].*\.pl>; # El diamante es una llamada a glob
19    return <<"EOI";
20  ---glob(’^.[cd][ba].*\.pl’)---
21  @a
22  ---<^.[cd][ba].*\.pl>---
23  @b
24  EOI
25  }
26
27  package main;
28
29  sub tutu {
30    my @a = glob(’^.[cd][ba].*\.pl’);
31    local $" = "\n";
32    return "@a\n";
33  }
34
35  print Glob::Regexp::tutu;
36  print "---------main-----------\n";
37  print tutu;
```

Al ejecutar el programa vemos que la subrutina `Glob::Regexp::tutu` (llamada desde la línea 35) usa la nueva versión de `glob` mientras que los usos de `glob` dentro del paquete `main` usan la versión original.

La ejecución también muestra que el uso de diamantes en ”modo directorio” como en `<^.[cd][ba].*\.pl>` implica una llamada implícita a `glob`.

```
lhp@nereida:~/Lperl/src$ usesubs.pl
---glob(’^.[cd][ba].*.pl’)---
idbetcurly.pl
pcap.pl
scalar_sets.pl
scalar_sets2.pl
sdbm.pl
sdbm2.pl
---<^.[cd][ba].*.pl>---
idbetcurly.pl
pcap.pl
scalar_sets.pl
```

```
scalar_sets2.pl
sdbm.pl
sdbm2.pl
---------main-----------
```

5.12. Los Paquetes `CORE` y `CORE::GLOBAL`

El Prefijo `CORE`

Supongamos que estamos en un paquete que ha sobreescrito un operador Perl. Por ejemplo `glob`. En tal caso cualquier alusión a `glob` dentro del paquete se referirá a la nueva versión de `glob`.

¿Como podemos llamar al ”viejo” glob dentro del paquete?:

Escribiendo `CORE::glob`. El prefijo `CORE::` se refiere a los operadores Perl pre-construidos. Los operadores Perl no residen en ninguna tabla de símbolos y por tanto no existe un módulo `CORE::`. La construcción `CORE::operador` es una mera notación para poder referirnos a las funciones del núcleo de Perl.

El Paquete `CORE::GLOBAL` Como vimos en la sección 5.11 el pragma `use subs` permite la sustitución de un operador en el ámbito de un paquete. Si se quiere sustituir cualquier referencia (desde el paquete o fuera de él) al operador por la nueva versión es necesario declarar la nueva versión dentro del paquete `CORE::GLOBAL`:

```
$ perl -e ’BEGIN { *CORE::GLOBAL::system = sub { print "hello $_[0]\n" } } system "foo"’
hello foo
```

Sigue un ejemplo en el que reeemplazamos gobalmente el operador `glob`:

```
lhp@nereida:˜/Lperl/src$ cat -n coreglob.pl
#!/usr/local/bin/perl -w
package Tutu;
use strict;

{
  no warnings;
  *CORE::GLOBAL::glob = sub {
    my $regexp = shift;

    opendir my $DIR, "." or die "$!";
    my @files = grep /$regexp/, readdir $DIR;
    close($DIR);
    return @files;
  };
}
1;

package main;
use strict;

sub tutu {
  my @a = glob(’ˆ.[cd][ba].*\.pl’);
  local $" = "\n";
  return "@a\n";
}

print "---------main-----------\n";
print tutu;
```

ahora la llamada en el paquete `main` a `glob` encuentra la nueva versión:

```
lhp@nereida:~/Lperl/src$ coreglob.pl
---------main-----------
idbetcurly.pl
pcap.pl
scalar_sets.pl
scalar_sets2.pl
sdbm.pl
sdbm2.pl
```

5.13. 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 esté 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.

Heredando de `Exporter`

El módulo `Exporter` provee diferentes mecanismos para realizar la interfaz pública del módulo que estamos desarrollando.

En el ejemplo que sigue, la inicialización del vector especial `@ISA` en la línea 5 hace que (junto que el `use Exporter` de la línea 3) el módulo `Modexample::HopsExport` ”herede” de `Exporter` los métodos que nos hacen falta como `import`.

```
lhp@nereida:~/projects/perl/src$ cat -n Modexample/HopsExport.pm
 1  package Modexample::HopsExport;
 2  use strict;
 3  use Exporter;
 4
 5  our @ISA = (’Exporter’);
 6  our @EXPORT = qw(&hop_along);
 7
 8  sub hop_along {
 9    my ($from, $to, $step) = @_;
10    my $next = $from-$step;  # incializar contador
11    my $closure_ref = sub {
12      $next += $step;
13      $next = $from-$step, return if $next > $to;
14      $_[0] =$next;
15      return 1;
16    };
17    return $closure_ref;
18  }
19
20  1;
```

Aún cuando no hemos visto objetos, puede dar una ojeada a la sección 6.5 que trata sobre la herencia. La herencia indica que los métodos definidos y exportados por los paquetes en el array `@ISA` estan disponibles en el módulo cliente.

EXPORT y EXPORT_OK

El método `import` que proporciona `Exporter` examina la lista de cadenas en `@EXPORT` para determinar que funciones y variables se exportan por defecto.

Si tenemos variables o rutinas que sólo deben ser exportadas bajo demanda del cliente (como `foo` en `use Tutu qw(foo)`) debemos escribir sus nombres 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`.

```
lhp@nereida:~/Lperl/src$ cat -n usehopsexport.pl
#!/usr/bin/perl -w -I.
use strict;
use Modexample::HopsExport;

my ($r, $c);
my $row = hop_along 1, 5, 1;
my $col = hop_along 1, 5, 1;
while ($row->($r)) {
  while ($col->($c)) {
    print("($r, $c)\t");
  }
  print "\n";
}
```

Reglas de Exportación El módulo `Exporter` permite definir de manera precisa la interfaz 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` usa un conjunto de variables en el paquete que gobiernan la conducta de exportación del módulo. 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`.

Puesto que las variables `@EXPORT`, `@EXPORT_OK` y `%EXPORT_TAGS` son del paquete cuyo nombre casa con el del fichero[1], 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.

Usando `use`

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`.

---

[1] El fichero sin embargo puede contener varios `packages`

- `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`.

Visbilidad de las Funciones

Observe que un programa cliente siempre puede acceder a las variables de paquete del módulo y a sus funciones, incluso si no figuran como símbolos exportables, sin mas que escribir su nombre completo (por ejemplo `Trivial::Tutu::siete`).

La Etiqueta `:DEFAULT`

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)
```

Uso de etiquetas: el hash `EXPORT_TAGS`

El ”hash” `%EXPORT_TAGS` es usado por módulos que proveen un gran número de funciones, como es el caso de 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,
```

```
    hr;

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.

Referencias

Recuerde consultar la documentación de Exporter con `perldoc Exporter`.

5.14. CPAN: The Comprehensive Perl Archive Network

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/

Para ver el estado de los mirrors CPAN visite http://mirrors.cpan.org/

Véase The Zen of Comprehensive Archive Networks para una explicación mas detallada de la estructura de CPAN.

- Es posible consultar el estado de actualización de un mirror en la página http://www.cs.uu.nl/stats/mirmon/cpan.html#es.
- The Perl Authors Server: PAUSE se encuentra en https://pause.perl.org/
- El reastreo de bugs en los módulos de CPAN se hace con RT: https://rt.cpan.org/
- Toda la historia de distribuciones de CPAN se almacena en BackPan: http://backpan.perl.org/
- Las distribuciones son probadas en una variedad de plataformas por los CPAN testers: http://testers.cpan.org/
- El módulo CPAN provee el software necesario para la instalación automática de módulos desde CPAN

5.14.1. 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 5.1).

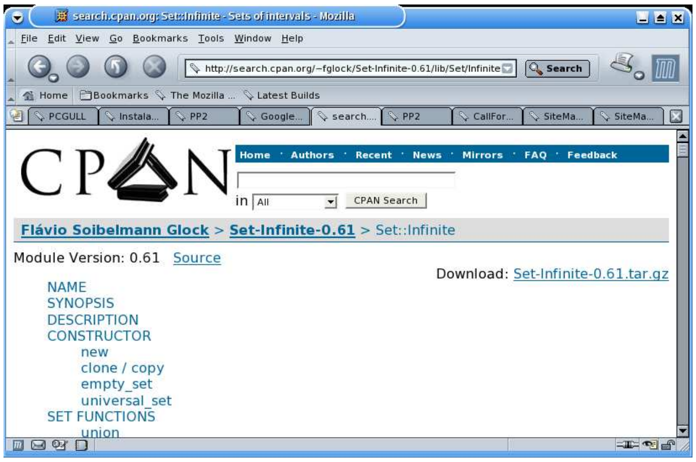

Figura 5.1: Buscando en CPAN

- En la parte superior encontrará enlaces al autor y la distribución. Algo asi:

  ```
  David Robins >  Net-SSH-Perl-1.30 >  Net::SSH::Perl
  ```

  El enlace de distribución esta formado a partir del nombre y el número de versión (`Net-SSH-Perl-1.30` en el ejemplo).

- 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.
- En la página de la distribución, presione sobre la opción `download` para descárgar el módulo en el directorio que desea. La opción download suele estar en la parte superior de la página:

  ```
  This Release    Net-SSH-Perl-1.30         [Download] [Browse]    18 Mar 2006
  ```

- Después - supuesto que esta en un sistema Unix - 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 INSTALL_BASE=/path/to/your/home/dir
```

o bien:

```
> perl Makefile.PL PREFIX=~/perl5
```

En ambos casos asegúrate que los directorios de búsqueda en los que las librerías quedan instalados (`~/perl5/...`) serán encontrados por Perl.

- Despues de esto deberás ejecutar el clásico *mantra de instalación*:

```
> make
> make test
> make install
```

Referencias

Véase `perldoc` perlmodinstall. El documento describe como instalar módulos en otros Sistemas Operativos (Windows, Macintosh, etc.). Consulte tambien http://www.cpan.org/misc/cpan-faq.html.

5.14.2. Práctica: Instalar un Módulo

Instale en un directorio local el módulo PAR. ¿De qué módulos depende? ¿Cuáles de ellos no pertenecen al núcleo de la distribución de Perl?

El programa corelist que viene con el módulo Module::CoreList permite saber a partir de que versión un módulo pertenece al núcleo de la distribución de Perl

```
$ corelist Data::Dumper

Data::Dumper was first released with perl 5.005
```

Recuerde pasar la opción adecuada cuando ejecute `perl Makefile.PL`. 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.

5.14.3. Saber que Módulos están Instalados

Existen varias soluciones al problema de conocer si un módulo está instalado:

`Module::Util`

Si Module::Util está instalado, use pm_which:

```
lhp@europa:~$ pm_which GRID::Machine
/usr/local/share/perl/5.8.8/GRID/Machine.pm
```

Use `perldoc`

Para saber si un módulo está instalado y donde se encuentra (supuesto que el módulo contiene documentación en su interior) use `perldoc -l`:

```
lhp@nereida:~/LHPbook$ perldoc -l Parse::Yapp
/usr/local/share/perl/5.8.4/Parse/Yapp.pm
lhp@nereida:~/LHPbook$ perldoc -l Parse::Yapp::Output
No documentation found for "Parse::Yapp::Output".
lhp@nereida:~/LHPbook$ ls /usr/local/share/perl/5.8.4/Parse/Yapp/Output.pm
/usr/local/share/perl/5.8.4/Parse/Yapp/Output.pm
lhp@nereida:~/LHPbook$
```

Simplemente: Use el Módulo

También puede comprobar la existencia cargando el módulo:

```
lhp@nereida:~/Lperl/src/ExtUtils$ perl -MNoexiste -e 1
Can't locate Noexiste.pm in @INC (@INC contains: /etc/perl ...
BEGIN failed--compilation aborted.
lhp@nereida:~/Lperl/src/ExtUtils$ perl -MYAML -e 1
lhp@nereida:~/Lperl/src/ExtUtils$
```

Use el módulo `ExtUtils::Installed` El módulo ExtUtils::Installed proporciona diversos métodos para manejar los módulos Perl instalados. Escriba el siguiente guión:

```
lhp@nereida:~/Lperl/src/ExtUtils$ cat -n pminst
 1  #!/usr/local/bin/perl
 2  use strict;
 3  use ExtUtils::Installed;
 4
 5  my $instmod = ExtUtils::Installed->new();
 6
 7  foreach my $module ($instmod->modules()) {
 8    my $version = $instmod->version($module) || "???";
 9    print "$module $version\n";
10  }
```

Ahora podemos averiguar que módulos instalados contienen la palabra ”soap”:

```
lhp@nereida:~/Lperl/src/ExtUtils$ pminst | grep -i soap
SOAP 0.28
SOAP::Lite 0.60
```

El Conjunto de Utilidades `pmtools` Instale pmtools y use el comando `pminst`. Se puede obtener en http://search.cpan.org/ mlfisher/pmtools/.

Sigue un ejemplo de uso:

```
pp2@nereida:~/src/perl/IPC-PerlSSH/lib/IPC$ pmdesc IPC::PerlSSH # descripción
IPC::PerlSSH (0.05) - a class for executing remote perl code over an SSH link
pp2@nereida:~/src/perl/coro$ pminst moize
Aspect::Library::Memoize
pp2@nereida:~/src/perl/IPC-PerlSSH/lib/IPC$ pmload IPC::PerlSSH  # Que modulos carga
/usr/share/perl/5.8/Carp.pm
/usr/share/perl/5.8/IPC/Open3.pm
/usr/share/perl/5.8/IPC/Open2.pm
/usr/share/perl/5.8/Exporter.pm
```

```
/usr/share/perl/5.8/strict.pm
/usr/local/share/perl/5.8.8/IPC/PerlSSH.pm
/usr/share/perl/5.8/Symbol.pm
```

Módulos en el Núcleo de Perl En ocasiones no basta con saber que el módulo está instalado. Necesitamos saber si pertenece o no al núcleo de Perl. El método `first_release` del módulo Module::CoreList da respuesta a esta pregunta:

```
nereida:˜> perl -MModule::CoreList -e \
   ’print Module::CoreList->first_release("Data::Dumper")."\n"’
5.005
```

Mejor aún, use el programa corelist que acompaña a dicho módulo:

```
$ corelist Data::Dumper

Data::Dumper was first released with perl 5.005
```

5.14.4. Suprimir un Módulo Instalado

El siguiente guión `pmrm` muestra como eliminar los ficheros de una distribución:

```
nereida:/usr/sbin# cat -n pmrm
     1  #!/usr/local/bin/perl -w
     2  use strict;
     3  use ExtUtils::Packlist;
     4  use ExtUtils::Installed;
     5
     6  $ARGV[0] or die "Usage: $0 Module::Name\n";
     7
     8  my $mod = $ARGV[0];
     9
    10  my $inst = ExtUtils::Installed->new();
    11
    12  foreach my $item (sort($inst->files($mod))) {
    13    print "removing $item\n";
    14    unlink $item;
    15  }
    16
    17  my $packfile = $inst->packlist($mod)->packlist_file();
    18  print "removing $packfile\n";
    19  unlink $packfile;
```

Al ejecutarlo (con los privilegios adecuados) podemos eliminar los ficheros implicados:

```
nereida:/usr/sbin# pminst | grep -i flex
Parse::Flex 0.11
nereida:/usr/sbin# pmrm Parse::Flex
removing /usr/bin/makelexer.pl
removing /usr/local/man/man3/Parse::Flex.3pm
removing /usr/local/man/man3/Parse::Flex::Generate.3pm
removing /usr/local/share/perl/5.8.8/Parse/Flex.aux
removing /usr/local/share/perl/5.8.8/Parse/Flex.dvi
removing /usr/local/share/perl/5.8.8/Parse/Flex.idx
removing /usr/local/share/perl/5.8.8/Parse/Flex.log
```

```
removing /usr/local/share/perl/5.8.8/Parse/Flex.pdf
removing /usr/local/share/perl/5.8.8/Parse/Flex.pm
removing /usr/local/share/perl/5.8.8/Parse/Flex.tex
removing /usr/local/share/perl/5.8.8/Parse/Flex.toc
removing /usr/local/share/perl/5.8.8/Parse/Flex/Generate.pdf
removing /usr/local/share/perl/5.8.8/Parse/Flex/Generate.pm
removing /usr/local/lib/perl/5.8.8/auto/Parse/Flex/.packlist
nereida:/usr/sbin# ls -l /usr/local/share/perl/5.8.8/Parse/F*
total 0
```

En la documentación de `ExtUtils::Packlist` puede encontrar un ejemplo mas completo.

5.14.5. Usando el módulo `CPAN.pm` como Administrador

Instalar un módulo con `CPAN.pm`

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 install Mi::Modulo
```

o bien puedes iniciar una sesión interactiva llamando a la función `shell`:

```
%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.

Configuración de `CPAN`

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
```

o bien mas simple:

```
$ cpan
```

Obtendremos una respuesta similar a esta:

```
nereida:~/src/perl/Nmap-scanner# cpan
CPAN: File::HomeDir loaded ok (v0.69)

cpan shell -- CPAN exploration and modules installation (v1.9102)
ReadLine support enabled
```

El Comando `h`

```
cpan shell -- CPAN exploration and modules installation (v1.70)
ReadLine support enabled
```

Pedimos ayuda . . .

```
cpan[1]> h

Display Information                                               (ver 1.9102)
 command  argument          description
 a,b,d,m  WORD or /REGEXP/  about authors, bundles, distributions, modules
 i        WORD or /REGEXP/  about any of the above
 ls       AUTHOR or GLOB    about files in the author's directory
    (with WORD being a module, bundle or author name or a distribution
    name of the form AUTHOR/DISTRIBUTION)

Download, Test, Make, Install...
 get      download                     clean    make clean
 make     make (implies get)           look     open subshell in dist directory
 test     make test (implies make)     readme   display these README files
 install  make install (implies test)  perldoc  display POD documentation

Upgrade
 r        WORDs or /REGEXP/ or NONE    report updates for some/matching/all modules
 upgrade  WORDs or /REGEXP/ or NONE    upgrade some/matching/all modules

Pragmas
 force  CMD    try hard to do command  fforce CMD    try harder
 notest CMD    skip testing

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                recent        latest CPAN uploads
cpan[2]>
```

El Comando `a`

El comando `a` nos permite obtener información sobre los autores:

```
cpan> a /.*wall/
Author          CCWALLACE ("C. Chad Wallace" <cmdrwalrus@canada.com>)
Author          LWALL ("Larry Wall. Author of Perl. Busy man." <larry@wall.org>)
Author          PAWAL ("Patrik Wallstrom" <pawal@blipp.com>)
Author          SABREN ("Michal Wallace" <sabren@manifestation.com>)
Author          SHAWNPW ("Shawn P. Wallace" <shawn@as220.org>)
Author          THW ("Thomas Walloschke" <thw@cpan.org>)
Author          ZAPHAR ("Jeremy Wall" <Jeremy@marzhillstudios.com>)
7 items found
```

El Comando `b`

El comando `b` nos permite obtener información sobre los `bundles`. Los bundles son grupos de módulos relacionados. Los bundles son usados por `CPAN` para simplificar la instalación de un grupo de módulos. Una sóla `b` listará los bundles disponibles. Si se especifica un argumento se obtiene información sobre el bundle:

```
cpan> b /ssh/
Bundle id = Bundle::SSH
    CPAN_USERID  SZABGAB (Gabor Szabo <gabor@pti.co.il>)
    CPAN_VERSION 1.00
```

```
    CPAN_FILE    S/SZ/SZABGAB/Bundle-SSH-1.00.tar.gz
    MANPAGE      Bundle::SSH - A bundle to install modules to use SSH from Perl
    CONTAINS     Net::SSH Math::Pari Class::Loader Crypt::Random Digest::SHA1 \
                 Digest::HMAC Digest::BubbleBabble Digest::MD2 \
                 Convert::ASN1 Crypt::Rijndael Crypt::CBC Crypt::DES \
                 Crypt::DES_EDE3 Convert::PEM Data::Buffer Crypt::DSA \
                 Crypt::DH String::CRC32 Math::GMP Compress::Zlib \
                 Convert::ASCII::Armour Crypt::Blowfish Crypt::Primes \
                 Sort::Versions Tie::EncryptedHash Crypt::RSA \
                 Net::SSH::Perl \
    INST_FILE    /root/.cpan/Bundle/SSH.pm
    INST_VERSION 1.00
```

Evaluación de Expresiones

El comando `!` permite la evaluación de expresiones Perl:

```
cpan[2]>  !print 13*56,"\n"
728
cpan[2]> !use Perl6::Say
Subroutine IO::Handle::say redefined at /usr/local/share/perl/5.8.4/Perl6/Say.pm line 26.
cpan[2]> !say "hello"
hello
cpan[2]> !print "hello"
hellocpan[2]>
```

El Comando `r`

El comando `r` nos permite conocer que módulos necesitan actualización:

```
cpan[2]> r /Net/
CPAN: Storable loaded ok (v2.15)
Going to read /root/.cpan/Metadata
  Database was generated on Thu, 08 May 2008 14:29:50 GMT
CPAN: YAML loaded ok (v0.66)
Going to read /root/.cpan/build/
............................................................................DONE
Found 88 old builds, restored the state of 41

Package namespace         installed    latest  in CPAN file
Mail::Internet                 1.76      2.03  MARKOV/MailTools-2.03.tar.gz
Net::ARP                        0.8       1.0  CRAZYDJ/Net-ARP-1.0.2.tgz
Net::Amazon                    0.40      0.49  BOUMENOT/Net-Amazon-0.49.tar.gz
Net::Cmd                       2.27      2.29  GBARR/libnet-1.22.tar.gz
Net::DNS                       0.59      0.63  OLAF/Net-DNS-0.63.tar.gz
Net::Daemon                    0.38      0.43  MNOONING/Net-Daemon/Net-Daemon-0.43.tar.gz
Net::Domain::TLD               1.65      1.67  ALEXP/Net-Domain-TLD-1.67.tar.gz
Net::Netstat::Wrapper          0.02      0.03  MCANTONI/Net-Netstat-Wrapper-0.03.tar.gz
Net::Pcap                      0.04      0.16  SAPER/Net-Pcap-0.16.tar.gz
Net::Ping                      2.31      2.35  SMPETERS/Net-Ping-2.35.tar.gz
Net::Proxy                     0.08      0.12  BOOK/Net-Proxy-0.12.tar.gz
Net::SCP                       0.07      0.08  IVAN/Net-SCP-0.08.tar.gz
Net::SFTP::Foreign             0.61      1.36  SALVA/Net-SFTP-Foreign-1.36.tar.gz
Net::SSH::Expect               1.04      1.09  BNEGRAO/Net-SSH-Expect-1.09.tar.gz
Net::SSL                       2.77      2.84  DLAND/Crypt-SSLeay-0.57.tar.gz
Net::SSLeay                    1.30      1.32  FLORA/Net-SSLeay-1.32.tar.gz
```

```
Net::Server                    0.96      0.97  RHANDOM/Net-Server-0.97.tar.gz
300 installed modules have no parseable version number
```

El comando `upgrade`

Una vez hemos visto que un módulo necesita actualización podemos actualizarlo con `upgrade`.

```
cpan[3]> upgrade Net::Server

Package namespace         installed    latest  in CPAN file
Net::Server                    0.96      0.97  RHANDOM/Net-Server-0.97.tar.gz
Running install for module 'Net::Server'
Running make for R/RH/RHANDOM/Net-Server-0.97.tar.gz
CPAN: LWP::UserAgent loaded ok (v5.810)
CPAN: Time::HiRes loaded ok (v1.86)
Fetching with LWP:
  http://www.perl.org/CPAN/authors/id/R/RH/RHANDOM/Net-Server-0.97.tar.gz
CPAN: Digest::SHA loaded ok (v5.45)
Fetching with LWP:
  http://www.perl.org/CPAN/authors/id/R/RH/RHANDOM/CHECKSUMS
Checksum for /root/.cpan/sources/authors/id/R/RH/RHANDOM/Net-Server-0.97.tar.gz ok
Scanning cache /root/.cpan/build for sizes
........................................................................----DONE
DEL(1/10): /root/.cpan/build/Encode-2.24-MsvFPh
...................................................
DEL(3/10): /root/.cpan/build/Module-Starter-1.470-278pfH
Net-Server-0.97/
Net-Server-0.97/t/
Net-Server-0.97/t/Server_PreFork.t
Net-Server-0.97/t/UDP_test.t
....................................
Net-Server-0.97/README
CPAN: File::Temp loaded ok (v0.18)

  CPAN.pm: Going to build R/RH/RHANDOM/Net-Server-0.97.tar.gz

Checking if your kit is complete...
Looks good
Writing Makefile for Net::Server
...............................................................
Running make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib', 'blib/arch')"
t/Options.................ok
............................
All tests successful.
Files=12, Tests=190,  2 wallclock secs ( 1.10 cusr +  0.20 csys =  1.30 CPU)
  RHANDOM/Net-Server-0.97.tar.gz
  /usr/bin/make test -- OK
Running make install
Installing /usr/local/share/perl/5.8.8/Net/Server.pm
.....................................................
Appending installation info to /usr/lib/perl/5.8/perllocal.pod
  RHANDOM/Net-Server-0.97.tar.gz
  /usr/bin/make install  -- OK
```

```
cpan[4]>
```

El Comando `ls`

El comando `ls` nos permite listar los módulos desarrollados por un autor:

```
cpan> ls DCONWAY
Fetching with LWP:
  ftp://archive.progeny.com/CPAN/authors/id/D/CHECKSUMS
Fetching with LWP:
  ftp://archive.progeny.com/CPAN/authors/id/D/DC/CHECKSUMS
Fetching with LWP:
  ftp://archive.progeny.com/CPAN/authors/id/D/DC/DCONWAY/CHECKSUMS
    4299 2001-05-22 DCONWAY/Acme-Bleach-1.12.tar.gz
    2410 2002-05-01 DCONWAY/Acme-Don-t-1.00.tar.gz
    2659 2002-05-02 DCONWAY/Acme-Don-t-1.01.tar.gz
   11942 2001-06-02 DCONWAY/Attribute-Handlers-0.70.tar.gz
   12344 2001-09-02 DCONWAY/Attribute-Handlers-0.75.tar.gz
   12858 2001-11-14 DCONWAY/Attribute-Handlers-0.76.tar.gz
   ...................................... etc.etc.tg.gz
     347 2006-05-05 DCONWAY/Text-Balanced-1.98.meta
   29040 2006-05-05 DCONWAY/Text-Balanced-1.98.tar.gz
   18557 2003-04-02 DCONWAY/Text-Reform-1.08.tar.gz
   20493 2003-04-09 DCONWAY/Text-Reform-1.10.tar.gz
   20902 2003-05-07 DCONWAY/Text-Reform-1.11.tar.gz
    7871 1999-05-14 DCONWAY/Tie-SecureHash-1.02.tar.gz
    7994 1999-11-11 DCONWAY/Tie-SecureHash-1.03.tar.gz
     447 2005-08-03 DCONWAY/Toolkit-0.0.2.meta
    5491 2005-08-03 DCONWAY/Toolkit-0.0.2.tar.gz
```

También es posible usar comodines con el comando:

```
cpan> ls DCON*
```

La Clase `CPAN::Shell`

Los comandos que están disponibles en la interfaz de la shell son métodos del paquete CPAN::Shell. Cada vez que se introduce un comando shell la entrada es analizada por la rutina `Text::ParseWords::shellwords` de Text::ParseWords, la cual analiza la entrada como lo hacen la mayoría de las shells:

```
pp2@nereida:~/src/testing$ perl -MText::ParseWords -wde 0
main::(-e:1):   0
  DB<1> @w = shellwords('comando arg1 arg2 arg3');
  DB<2> x @w
0  'comando'
1  'arg1'
2  'arg2'
3  'arg3'
  DB<3> @v = shellwords('comando "arg1 arg2" arg3');
  DB<4> x @v
0  'comando'
1  'arg1 arg2'
2  'arg3'
```

La primera palabra se interpreta se interpreta como el nombre de un método a llamar mediante referenciado simbólico y las siguientes palabras como argumentos para ese método.

El Comando `look`

El comando `look` abre una shell en el directorio en el que se ha descargado la distribución especificada. Si no se ha descargado la distribución procederá a hacerlo:

```
nereida:~# cpan
CPAN: File::HomeDir loaded ok (v0.69)

cpan shell -- CPAN exploration and modules installation (v1.9102)
ReadLine support enabled

cpan[1]> look Net::Server
Trying to open a subshell in the build directory...
Working directory is /root/.cpan/build/Net-Server-0.97-6wI2wN
nereida:~/.cpan/build/Net-Server-0.97-6wI2wN# ls -l
total 132
drwxr-xr-x 8 root    root     4096 2008-05-09 07:11 blib
-rw-r--r-- 1 casiano casiano 21994 2007-07-25 17:17 Changes
drwxr-xr-x 2 casiano casiano  4096 2007-07-25 17:23 examples
drwxr-xr-x 3 casiano casiano  4096 2007-07-25 17:23 lib
-rw-r--r-- 1 root    root    27741 2008-05-09 07:11 Makefile
-rw-r--r-- 1 casiano casiano   612 2007-02-03 08:18 Makefile.PL
-rw-r--r-- 1 casiano casiano  2198 2007-02-03 08:05 MANIFEST
-rw-r--r-- 1 casiano casiano    97 2007-02-03 08:02 MANIFEST.SKIP
-rw-r--r-- 1 casiano casiano   418 2007-07-25 17:23 META.yml
-rw-r--r-- 1 root    root        0 2008-05-09 07:11 pm_to_blib
-rw-r--r-- 1 casiano casiano 47485 2007-02-03 08:19 README
drwxr-xr-x 2 casiano casiano  4096 2007-07-25 17:23 t
^D # retornamos a la shell de CPAN
```

Instalación de Versiones de Distribuciones

Para instalar una distribución con un número de versión específico hay que utilizar el identificador de distribución (*Distribution id*):

```
cpan[6]> d /Expect-Simple/
Distribution id = D/DJ/DJERIUS/Expect-Simple-0.04.tar.gz
    CPAN_USERID  DJERIUS (Diab Jerius <djerius@cpan.org>)
    CONTAINSMODS Expect::Simple
    UPLOAD_DATE  2008-05-06
```

La instalación de un módulo es manejada por el método `install` de la clase `CPAN::Module` mientras que el de una distribución lo es por la clase `CPAN::Distribution`:

```
cpan[7]> install D/DJ/DJERIUS/Expect-Simple-0.04.tar.gz
Running make for D/DJ/DJERIUS/Expect-Simple-0.04.tar.gz
Fetching with LWP:
  http://www.perl.org/CPAN/authors/id/D/DJ/DJERIUS/Expect-Simple-0.04.tar.gz
CPAN: Digest::SHA loaded ok (v5.45)
Checksum for /root/.cpan/sources/authors/id/D/DJ/DJERIUS/Expect-Simple-0.04.tar.gz ok
Scanning cache /root/.cpan/build for sizes
............................................................................DONE
Expect-Simple-0.04/
Expect-Simple-0.04/Makefile.PL
Expect-Simple-0.04/MANIFEST.SKIP
Expect-Simple-0.04/README
```

```
Expect-Simple-0.04/META.yml
Expect-Simple-0.04/Changes
Expect-Simple-0.04/MANIFEST
Expect-Simple-0.04/t/
Expect-Simple-0.04/t/testprog
Expect-Simple-0.04/t/Expect-Simple.t
Expect-Simple-0.04/LICENSE
Expect-Simple-0.04/lib/
Expect-Simple-0.04/lib/Expect/
Expect-Simple-0.04/lib/Expect/Simple.pm
Expect-Simple-0.04/ChangeLog
CPAN: File::Temp loaded ok (v0.18)

  CPAN.pm: Going to build D/DJ/DJERIUS/Expect-Simple-0.04.tar.gz

Checking if your kit is complete...
Looks good
Writing Makefile for Expect::Simple
cp lib/Expect/Simple.pm blib/lib/Expect/Simple.pm
Manifying blib/man3/Expect::Simple.3pm
  DJERIUS/Expect-Simple-0.04.tar.gz
  /usr/bin/make -- OK
Running make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" \
                "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/Expect-Simple....ok
All tests successful.
Files=1, Tests=6,  0 wallclock secs ( 0.12 cusr +  0.03 csys =  0.15 CPU)
  DJERIUS/Expect-Simple-0.04.tar.gz
  /usr/bin/make test -- OK
Running make install
Installing /usr/local/share/perl/5.8.8/Expect/Simple.pm
Installing /usr/local/man/man3/Expect::Simple.3pm
Writing /usr/local/lib/perl/5.8.8/auto/Expect/Simple/.packlist
Appending installation info to /usr/lib/perl/5.8/perllocal.pod
  DJERIUS/Expect-Simple-0.04.tar.gz
  /usr/bin/make install  -- OK

cpan[8]>
```

5.14.6. Opciones de Configuración

Lista de Opciones de Configuración

El comando `o conf` nos permite ver y modificar las opciones de configuración:

```
cpan> o conf
CPAN::Config options and /home/casiano/.cpan/CPAN/MyConfig.pm:
    commit             Commit changes to disk
    defaults           Reload defaults from disk
    init               Interactive setting of all options

    build_cache        1000
    build_dir          /scratch/casiano/build
    cache_metadata     1
```

```
    cpan_home          /scratch/casiano/.cpan
    dontload_hash
    ftp                /usr/bin/ftp
    ftp_proxy
    getcwd             cwd
    gpg                /usr/bin/gpg
    gzip               /bin/gzip
    histfile           /scratch/casiano/.cpan/histfile
    histsize           100
    http_proxy
    inactivity_timeout 0
    index_expire       1
    inhibit_startup_message 0
    keep_source_where  /scratch/casiano/.cpan/sources
    lynx               /usr/bin/lynx
    make               /usr/bin/make
    make_arg
    make_install_arg
    makepl_arg         PREFIX=/soft/perl5lib/
    ncftpget           /usr/bin/ncftpget
    no_proxy
    pager              /usr/bin/less
    prerequisites_policy ask
    scan_cache         atstart
    shell              /bin/bash
    tar                /bin/tar
    term_is_latin      1
    unzip              /usr/bin/unzip
    urllist
        ftp://cpan.ip.pt/pub/cpan/
        ftp://ftp.rediris.es/mirror/CPAN/
        ftp://ftp.etse.urv.es/pub/perl/
        ftp://ftp.ri.telefonica-data.net/CPAN
        http://cpan.imasd.elmundo.es/
    wget               /usr/bin/wget
```

Ayuda sobre `o conf`

El comando `o conf help` permite obtener una ayuda resumida:

```
cpan[14]> o conf help

Known options:
  commit    commit session changes to disk
  defaults  reload default config values from disk
  help      this help
  init      enter a dialog to set all or a set of parameters

Edit key values as in the following (the "o" is a literal letter o):
  o conf build_cache 15
  o conf build_dir "/foo/bar"
  o conf urllist shift
  o conf urllist unshift ftp://ftp.foo.bar/
  o conf inhibit_startup_message 1
```

Modificación de la Lista de Servidores CPAN

Es posible consultar una opción específica:

```
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 al proceder a una instalación la URL insertada será la primera en ser consultada:

```
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
....
```

Para ver el estado de los mirrors puede consultar cualquiera de estas direcciones:

- http://www.cs.uu.nl/stats/mirmon/cpan.html#es.
- http://mirrors.cpan.org/. En esta página es posible buscar por países. Véase una posible salida para 'España':

  ```
  There are 3 mirrors matching your query: 'Spain'
  Name:   uva.es
  Organisation:   Grupo Universitario de Informática (Universidad de Valladolid)
  Location:       Valladolid, Spain, Europe
  Latitude/Longitude and Timezone         41.6770148220322/-4.728240966796875, +0
  Name:   osl.ugr.es
  Organisation:   Oficina de Software Libre de la Universidad de Granada
  Location:       Granada, Andalucía, Spain, Europe
  Latitude/Longitude and Timezone         37.18638888/-3.77749999, +1
  Name:   rediris.es
  Organisation:   Red Académica y de Investigación Nacional Española (Spanish Academic Network for Resea
  Location:       Madrid, Spain, Europe
  Latitude/Longitude and Timezone         40.42031/-3.70562, +1
  ```

- Lea el nodo What Does CPAN Mirror ”Freshness Date”Mean? en PerlMonks

Directorio en el que se Guardan las Distribuciones
El directorio en el que se guardan las distribuciones puede obtenerse asi:

```
cpan[10]> o conf keep_source_where
    keep_source_where  [/root/.cpan/sources]
Type 'o conf' to view all configuration items



cpan[11]> ^Z                     # <-- ponemos el proceso en background ...
[1]+  Stopped                 cpan
nereida:~/src/perl/Nmap-scanner# ls -l /root/.cpan/sources
total 320
drwxr-xr-x 3 root root   4096 2008-05-09 07:07 authors
-rw-r--r-- 1 root root 163409 2006-03-13 13:57 MIRRORED.BY
-rw-r--r-- 1 root root 145453 2004-11-03 13:26 MIRRORED.BY.bak
drwxr-xr-x 2 root root   4096 2008-05-09 07:07 modules
nereida:~/src/perl/Nmap-scanner  # fg  <-- y lo volvemos a poner en "foreground"
cpan[11]>
```

Estructura del Directorio de Distribuciones
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
```

Modificar la Configuración Inicial

En cualquier momento es posible reiniciar el *diálogo de configuración* mediante el commando `o conf init`:

```
cpan> o conf init

/etc/perl/CPAN/Config.pm initialized.

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]
...
```

Modificación Guiada de Opciones

También es posible especificar que opciones se quieren cambiar restringiendo el diálogo a estas:

```
cpan[1]>  o conf init makepl_arg mbuildpl_arg prefs_dir


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 or a personal directory.



CPAN.pm can store customized build environments based on regular
expressions for distribution names. These are YAML files where the
default options for CPAN.pm and the environment can be overridden and
dialog sequences can be stored that can later be executed by an
Expect.pm object. The CPAN.pm distribution comes with some prefab YAML
files that cover sample distributions that can be used as blueprints
to store one own prefs. Please check out the distroprefs/ directory of
the CPAN.pm distribution to get a quick start into the prefs system.

Directory where to store default options/environment/dialogs for
building modules that need some customization? [~/.cpan/prefs]


Every Makefile.PL is run by perl in a separate process. Likewise we
run 'make' and 'make install' in separate processes. If you have
any parameters (e.g. PREFIX, LIB, UNINST or the like) you want to
pass to the calls, please specify them here.

If you don't understand this question, just press ENTER.

Parameters for the 'perl Makefile.PL' command?
Typical frequently used settings:

    PREFIX=~/perl    # non-root users (please see manual for more hints)

Your choice:  [INSTALL_BASE=~/perl] INSTALL_BASE=~/personalmodules


The next questions deal with Module::Build support.

A Build.PL is run by perl in a separate process. Likewise we run
'./Build' and './Build install' in separate processes. If you have any
parameters you want to pass to the calls, please specify them here.

Parameters for the 'perl Build.PL' command?
Typical frequently used settings:

    --install_base /home/xxx             # different installation directory

Your choice:  [--install_base ~/perl] --install_base ~/personalmodules


Please remember to call 'o conf commit' to make the config permanent!
```

```
cpan[2]> o conf commit
commit: wrote '/home/pp2/.cpan/CPAN/MyConfig.pm'
```

Las opciones `INSTALL_BASE` y `--install_base`

La opción `INSTALL_BASE` pasada a `makefile.PL` permite especificar en que lugar quedarán instalados los módulos. Esta opción es preferible a `PREFIX` pues es mas fácil predecir donde acabaran los módulos.

Al igual que `PREFIX` permite establecer varios atributos `INSTALL*` de una sola tacada. El patrón de instalación resultante sigue este esquema:

```
  INSTALLARCHLIB     INSTALL_BASE/lib/perl5/$Config{archname}
  INSTALLPRIVLIB     INSTALL_BASE/lib/perl5
  INSTALLBIN         INSTALL_BASE/bin
  INSTALLSCRIPT      INSTALL_BASE/bin
  INSTALLMAN1DIR     INSTALL_BASE/man/man1
  INSTALLMAN3DIR     INSTALL_BASE/man/man3
```

Observe la instalación de un módulo después de haber establecido `INSTALL_BASE`:

```
cpan[3]> install GRID::Machine
CPAN: Storable loaded ok (v2.15)
Going to read /home/pp2/.cpan/Metadata
  Database was generated on Sun, 18 May 2008 05:29:48 GMT
CPAN: YAML loaded ok (v0.66)
Going to read /home/pp2/.cpan/build/
.............................................................................
  CPAN.pm: Going to build C/CA/CASIANO/GRID-Machine-0.091.tar.gz
Checking if your kit is complete...
Looks good
Writing Makefile for GRID::Machine
.............................................................................
  CASIANO/GRID-Machine-0.091.tar.gz
  /usr/bin/make -- OK
Running make test
.............................................................................
  /usr/bin/make test -- OK
Running make install
Installing /home/pp2/personalmodules/lib/perl5/GRID/Machine.pm
Installing /home/pp2/personalmodules/lib/perl5/GRID/Machine.pod
.............................................................................
Installing /home/pp2/personalmodules/man/man1/remotetest.pl.1p
Installing /home/pp2/personalmodules/man/man3/GRID::Machine::Result.3pm
Installing /home/pp2/personalmodules/man/man3/GRID::Machine.3pm
.............................................................................
Installing /home/pp2/personalmodules/bin/remotetest.pl
Writing /home/pp2/personalmodules/lib/perl5/i486-linux-gnu-thread-multi/auto/GRID/Machine/.packlist
Appending installation info to /home/pp2/personalmodules/lib/perl5/i486-linux-gnu-thread-multi/perllocal.p
  CASIANO/GRID-Machine-0.091.tar.gz
  /usr/bin/make install  -- OK
cpan[4]>
```

Vemos que los módulos (p.ej. `GRID::Machine`) fueron instalados en `/home/pp2/personalmodules/lib/perl5/`, los manuales en `/home/pp2/personalmodules/man/` y los guiones en `/home/pp2/personalmodules/bin/`.

Para tener operativo nuestro repositorio tendremos que añadir estos caminos a las correspondientes variables de entorno. Lo aconsejable es incluir estos comandos en `.bashrc` o en`.cshrc`, según sea nuestra shell, para que se ejecute cada vez que creamos una nueva shell:

- `PATH`
  Establecer la variable de entorno `PATH` para encontrar los guiones/ejecutables. En una `bash`:

  ```
  $ ps
    PID TTY          TIME CMD
  12174 pts/21   00:00:00 bash
  21386 pts/21   00:00:00 ps
  $ export PATH=$PATH:/home/pp2/personalmodules/bin/
  $ echo $PATH
  /usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games:/usr/local/mpich-1.2.5/ch_p4/bin:
  /usr/local/j2sdk1.4.2_04/bin:.:/home/pp2/bin/:/usr/local/cc/bin:.:/home/pp2/bin/:
  /home/pp2/personalmodules/bin/
  $ which remotetest.pl
  /home/pp2/personalmodules/bin//remotetest.pl
  ```

- `PERL5LIB` Para que los módulos instalados y la documentación `.pod` puedan ser encontrados por Perl añadimos:

  ```
  $ export PERL5LIB=/home/pp2/personalmodules/lib/perl5:$PERL5LIB
  $ perldoc -l GRID::Machine
  ```

- `MANPATH` para que `man` encuentre las páginas de los manuales asociados con los módulos deberemos establecer la variable de entorno `MANPATH`:

  ```
  $ export MANPATH=/home/pp2/personalmodules/man/:$MANPATH
  $ manpath
  manpath: aviso: la variable $MANPATH está asignada, insertando /etc/manpath.config.
  /home/pp2/personalmodules/man/:/usr/local/man:/usr/local/share/man:/usr/share/man:/usr/local/mpich-1.
  $ man GRID::Machine
  Dando formato a GRID::Machine(3pm); aguarde, por favor...
  ```

La opción `install_base` juega un papel similar para el módulo alternativo a ExtUtils::MakeMaker mas usado: Module::Build.

El Módulo `CPAN/Config.pm`

Los valores por defecto vienen definidos en el fichero `CPAN/Config.pm`. El lugar en el que se guarda este módulo que contiene la configuración de CPAN viene dado por la opción `cpan_home`:

```
cpan[16]> o conf cpan_home
    cpan_home          [/root/.cpan]
```

Alternativamente, también podríamos hallar su ubicación con:

```
nereida:~> perl -MCPAN::Config -e ’print $INC{"CPAN/Config.pm"}."\n"’
/etc/perl/CPAN/Config.pm
```

Sigue un ejemplo de los contenidos de `CPAN::Config.pm`:

```
nereida:~> cat -n /etc/perl/CPAN/Config.pm
 1
 2  # This is CPAN.pm's systemwide configuration file. This file provides
 3  # defaults for users, and the values can be changed in a per-user
 4  # configuration file. The user-config file is being looked for as
 5  # ~/.cpan/CPAN/MyConfig.pm.
 6
 7  $CPAN::Config = {
 8    'build_cache' => q[10],
 9    'build_dir' => q[/root/.cpan/build],
10    'bzip2' => q[/usr/bin/bzip2],
11    'cache_metadata' => q[1],
12    'cpan_home' => q[/root/.cpan],
13    'cpan_version_check' => q[1],
14    'curl' => q[/usr/bin/curl],
15    'dontload_hash' => {  },
16    'ftp' => q[/usr/bin/ftp],
17    'ftp_passive' => q[1],
18    'ftp_proxy' => q[],
19    'getcwd' => q[cwd],
20    'gpg' => q[/usr/bin/gpg],
21    'gzip' => q[/bin/gzip],
22    'histfile' => q[/root/.cpan/histfile],
23    'histsize' => q[100],
24    'http_proxy' => q[],
25    'inactivity_timeout' => q[0],
26    'index_expire' => q[1],
27    'inhibit_startup_message' => q[0],
28    'keep_source_where' => q[/root/.cpan/sources],
29    'lynx' => q[/usr/bin/lynx],
30    'make' => q[/usr/bin/make],
31    'make_arg' => q[],
32    'make_install_arg' => q[],
33    'make_install_make_command' => q[/usr/bin/make],
34    'makepl_arg' => q[INSTALLDIRS=site],
35    'mbuild_arg' => q[],
36    'mbuild_install_arg' => q[],
37    'mbuild_install_build_command' => q[./Build],
38    'mbuildpl_arg' => q[],
39    'ncftpget' => q[/usr/bin/ncftpget],
40    'no_proxy' => q[],
41    'pager' => q[/usr/bin/less],
42    'prefer_installer' => q[EUMM],
43    'prerequisites_policy' => q[ask],
44    'scan_cache' => q[atstart],
45    'shell' => q[/bin/bash],
46    'show_upload_date' => q[1],
47    'tar' => q[/bin/tar],
48    'term_is_latin' => q[1],
49    'unzip' => q[/usr/bin/unzip],
50    'urllist' => [q[ftp://ftp.rediris.es/mirror/CPAN/], q[ftp://archive.progeny.com/CPAN/], q[ftp://cpan
51    'wait_list' => [q[wait://ls6.informatik.uni-dortmund.de]],
52    'wget' => q[/usr/bin/wget],
```

```
};
1;
__END__
```

De hecho una forma de modificar la configuración de CPAN es editando este fichero.

El Hash Anónimo `$CPAN::Config`

En la versión instalada en el ejemplo el hash anónimo `$CPAN::Config` contiene las siguientes claves:

Clave	Significado
`build_cache`	Tamaño de la cache para la construcción de los módulos
`build_dir`	Directorio para la construcción de los módulos
`index_expire`	Recargar el índice despueś del número de días indicado
`cache_metadata`	Usar serializador para metadatos
`cpan_home`	Directorio local para este paquete
`cpan_version_check`	if true, warns you when the CPAN module is out of date.
`dontload_hash`	anonymous hash: modules in the keys will not be loaded by the `CPAN::has_inst()` routine
`gzip`	location of external program gzip
`histfile`	file to maintain history between sessions
`histsize`	maximum number of lines to keep in histfile
`inactivity_timeout`	breaks interactive Makefile.PLs after this many seconds inactivity. Set to 0 to never break.
`inhibit_startup_message`	if true, does not print the startup message
`keep_source_where`	directory in which to keep the source (if we do)
`make`	location of external make program
`make_arg`	arguments that should always be passed to 'make'
`make_install_arg`	same as `make_arg` for `make install`
`makepl_arg`	arguments passed to `perl Makefile.PL`
`pager`	location of external program more (or any pager)
`prerequisites_policy`	what to do if you are missing module prerequisites ('follow' automatically, 'ask' me, or 'ignore')
`proxy_user`	username for accessing an authenticating proxy
`proxy_pass`	password for accessing an authenticating proxy
`scan_cache`	controls scanning of cache ('atstart' or 'never')
`tar`	location of external program tar
`term_is_latin`	if true internal UTF-8 is translated to ISO-8859-1 (and nonsense for characters outside latin range)
`unzip`	location of external program unzip
`urllist`	arrayref to nearby CPAN sites (or equivalent locations)
`wait_list`	arrayref to a wait server to try (See CPAN::WAIT)

`ftp_proxy,`
`http_proxy,`
`no_proxy`
	the three usual variables for configuring proxy requests. Both as CPAN::Config variables and as environment variables configurable.

Un `CPAN/Config.pm` de Usuario

Estos valores por defecto pueden ser sobreescritos usando un fichero `CPAN/Config.pm` específico para el usuario. Lo mas conveniente es dejar ese fichero en `$HOME/.cpan/CPAN/MyConfig.pm` ya que el directorio `$HOME/.cpan` es añadido al camino de búsqueda del módulo `CPAN` antes de la ejecución de las sentencias `use` y/o `require`.

5.14.7. Bundles

Un `bundle` es un tipo de objeto CPAN que simplifica la instalación de un conjunto de módulos. El modulo CPAN detecta que nos estamos refiriendo a un bundle porque siempre van prefijados por `Bundle::`.

Construccion de un Bundle con `autobundle`

Una de las formas de construir un Bundle es mediante el comando `autobundle`. El comando `autobundle` crea un Bundle conteniendo la información necesaria para la instalación de los módulos especificados:

```
cpan[1]> autobundle CGI Crypt::Rot13 Date::Christmas Date::Manip
CPAN: Storable loaded ok (v2.15)
Going to read /root/.cpan/Metadata
  Database was generated on Tue, 08 May 2007 03:10:17 GMT

Package namespace         installed    latest  in CPAN file
CGI                            3.15      3.29  LDS/CGI.pm-3.29.tar.gz

Wrote bundle file
    /root/.cpan/Bundle/Snapshot_2007_05_08_00.pm
```

Ahora el fichero `"Snapshot_2001_05_09_01.pm"` puede ser usado en conjunción con `CPAN.pm` para instalar la familia de módulos descrita en el bundle:

```
perl -MCPAN -e 'install Bundle::Snapshot_2001_05_09_01'
```

En que Directorio queda un Bundle

El fichero se deja en el directorio `"$CPAN::Config->{cpan_home}/Bundle"`. En modo interactivo podemos consultar la opción `cpan_home`:

```
cpan> o conf cpan_home
    cpan_home          /home/chaos/monkey/.cpan
```

Transportando una Instalación de Perl con `autobundle`

Si no se especifica lista de módulos se construirá un bundle para la instalación actual de la máquina:

```
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 en otra máquina con los mismos módulos que esta, sólo tendremos que instalar el bundle.

5.14.8. 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 líneas del fichero que casan con la expresión regular. Ahora editamos el fichero y cambiamos las líneas que hacen alusión al `root`:

```
/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.
```

5.14.9. Construyendo un Mirror de CPAN

El módulo CPAN::Mini de Ricardo Signes (http://search.cpan.org/ rjbs/) permite crear un mirror reducido de CPAN.

La forma mas simple de usarlo es mediante el ejecutable `minicpan`. Es posible guardar las opciones del ejecutable en un fichero de configuración `~/.minicpanrc`:

```
~# cat .minicpanrc
local:  /usr/local/src/CPAN/
remote: ftp://perl.di.uminho.pt/pub/CPAN/
skip_perl: 1
trace: 1
```

Ahora puede ejecutarse sin opciones:

```
minicpan
```

Asegúrese que tiene espacio suficiente. Puede ver el espacio ocupado por un directorio usando `du`

```
# du -sh CPAN/
694M    CPAN/
```

Como puede verse - en el momento de la escritura de esta sección la versión "mini" de CPAN ocupaba 694 Megabytes. Para usar su copia local añada a `urllist` la localización de su mirror:

```
cpan[1]> o conf urllist unshift /usr/local/src/CPAN/
```

Ahora puede instalar módulos sin estar conectado a internet o usar ese mirror como servidor CPAN de la organización en la que trabaja.

5.14.10. Práctica: CPAN

Reconfigure `CPAN` para trabajar como usuario ordinario. Compruebe el buen funcionamiento descargando un módulo.

- Lea el nodo PerlMonks *Yes, even you can use CPAN*

5.15. `PAR`: The Perl Archive Toolkit

El módulo PAR permite usar ficheros `.zip` que siguen un formato denominado *Perl Archives*.

Ficheros `.par` Simples

En una primera aproximación podemos decir que un fichero `.par` es un fichero zip que contiene módulos.

Supongamos que estoy en una máquina (`orion`) que tiene instalado el módulo Parse::Eyapp:

```
casiano@orion:/usr/local/share/perl/5.8.8$ perldoc -l Parse::Eyapp
/usr/local/share/perl/5.8.8/Parse/Eyapp.pm
```

Para crear un fichero `.par` basta con llamar a zip:

```
casiano@orion:/usr/local/share/perl/5.8.8$ zip -r /tmp/orionparse.par Parse/
updating: Parse/ (stored 0%)
updating: Parse/Eyapp.pm (deflated 69%)
updating: Parse/Eyapp/ (stored 0%)
updating: Parse/Eyapp/Lalr.pm (deflated 74%)
updating: Parse/Eyapp/YATW.pm (deflated 64%)
updating: Parse/Eyapp/Treeregexp.pm (deflated 77%)
updating: Parse/Eyapp/Parse.pm (deflated 78%)
updating: Parse/Eyapp/Scope.pm (deflated 61%)
updating: Parse/Eyapp/Options.pm (deflated 63%)
updating: Parse/Eyapp/Output.pm (deflated 60%)
updating: Parse/Eyapp/Node.pm (deflated 72%)
updating: Parse/Eyapp/Grammar.pm (deflated 71%)
updating: Parse/Eyapp/Driver.pm (deflated 70%)
updating: Parse/Eyapp/Base.pm (deflated 52%)
updating: Parse/Eyapp/_TreeregexpSupport.pm (deflated 50%)
```

La opción `-r` hace que `zip` recorra la estructura de directorios recursivamente.

Ahora transferimos el fichero `/tmp/orionparse.par` a una máquina (`nereida`) en la cual no esta instalado Parse::Eyapp:

```
lhp@nereida:~/Lperl/src$ scp orion:/tmp/orionparse.par .
orionparse.par                        100%  105KB 105.0KB/s   00:00
```

Utilizando PAR podemos cargar el módulo Parse::Eyapp desde el fichero `/tmp/orionparse.par`

```
lhp@nereida:~/Lperl/src$ perl -wde 0
main::(-e:1):   0
  DB<1> use PAR 'orionparse.par'
  DB<2> use Parse::Eyapp
  DB<3> print $Parse::Eyapp::VERSION
1.082
  DB<4>
```

Tambien podemos hacerlo con un *one-liner*:

```
lhp@nereida:~/Lperl/src$ perl -MPAR=orionparse.par -MParse::Eyapp -e 'print "$Parse::Eyapp::VERSION\n"'
1.082
```

Ejecutables en un `.par`

Es posible añadir ficheros ejecutables a un fichero `.par`. El módulo Parse::Eyapp viene con el ejecutable `eyapp`. Añadámoslo al fichero `.par`:

```
casiano@orion:/usr/local/share/perl/5.8.8$ which eyapp
/usr/local/bin/eyapp
casiano@orion:/usr/local/share/perl/5.8.8$ cd /usr/local/bin/
casiano@orion:/usr/local/bin$ zip -r /tmp/orionparse.par eyapp
  adding: eyapp (deflated 59%)
casiano@orion:/usr/local/bin$
```

Transferimos de nuevo el fichero `.par` a la máquina `nereida`:

```
lhp@nereida:~/Lperl/src$ scp orion:/tmp/orionparse.par .
orionparse.par                          100%  108KB 108.0KB/s   00:00
```

Siempre es posible listar los ficheros que forman parte de una distribución usando unzip con la opción `-l`:

```
lhp@nereida:~/Lperl/src$ unzip -l orionparse.par
Archive:  orionparse.par
  Length     Date   Time    Name
 --------    ----   ----    ----
        0  11-02-07 12:40   Parse/
   135815  11-01-07 13:14   Parse/Eyapp.pm
        0  11-02-07 12:40   Parse/Eyapp/
    29574  09-17-07 16:38   Parse/Eyapp/Lalr.pm
     8178  09-17-07 16:17   Parse/Eyapp/YATW.pm
    51402  11-01-07 13:17   Parse/Eyapp/Treeregexp.pm
    52766  11-01-07 13:17   Parse/Eyapp/Parse.pm
    11069  09-17-07 16:40   Parse/Eyapp/Scope.pm
     6314  09-17-07 16:39   Parse/Eyapp/Options.pm
     8826  09-17-07 17:53   Parse/Eyapp/Output.pm
    23784  09-17-07 16:17   Parse/Eyapp/Node.pm
    13832  09-17-07 16:38   Parse/Eyapp/Grammar.pm
    17865  11-01-07 13:05   Parse/Eyapp/Driver.pm
     3673  09-17-07 16:37   Parse/Eyapp/Base.pm
     3296  09-17-07 16:40   Parse/Eyapp/_TreeregexpSupport.pm
     7102  11-02-07 12:40   eyapp
        0  07-10-08 10:22   Math/
        0  08-08-08 12:08   Math/Prime/
     5635  05-14-08 15:30   Math/Prime/XS.pm
 --------                   -------
   379131                   19 files
```

El programa par.pl permite ejecutar ficheros en un archivo `.par`:

```
lhp@nereida:~/Lperl/src$ par.pl orionparse.par eyapp -V
This is Parse::Eyapp version 1.082.
```

Por defecto `par.pl` busca por un ejecutable con nombre `main.pl`.

También puedo extraer el ejecutable:

```
lhp@nereida:~/Lperl/src/tmp$ unzip orionparse.par eyapp
Archive:  orionparse.par
  inflating: eyapp
```

y a continuación ejecutarlo usando el módulo PAR:

```
lhp@nereida:~/Lperl/src/tmp$ perl -MPAR=orionparse.par eyapp -V
This is Parse::Eyapp version 1.082.
```

Empaquetado de Modulos XS

PAR soporta la carga de módulos XS (véase `perlxs`). XS es un formato para la descripción de interfases entre código C y código Perl. Veamos un ejemplo. La sesión se inicia como administrador arrancando `cpan` para a continuación descargar el módulo Math::Prime::XS el cual, como su nombre indica, tiene partes escritas en C:

```
root@orion:~# cpan

cpan shell -- CPAN exploration and modules installation (v1.7602)
ReadLine support enabled
```

El comando

```
look Math::Prime::XS
```

que emitimos a continuación indica que queremos abrir una shell en el directorio de la distribución de Math::Prime::XS. Si el módulo no está actualizado, `cpan` procederá a descargarse la última versión antes de abrir una shell:

```
cpan> look Math::Prime::XS
CPAN: Storable loaded ok
Going to read /root/.cpan/Metadata
  Database was generated on Fri, 08 Aug 2008 03:02:59 GMT
CPAN: LWP::UserAgent loaded ok
Fetching with LWP:
  ftp://perl.di.uminho.pt/pub/CPAN/authors/01mailrc.txt.gz
Going to read /root/.cpan/sources/authors/01mailrc.txt.gz
CPAN: Compress::Zlib loaded ok
Fetching with LWP:
  ftp://perl.di.uminho.pt/pub/CPAN/modules/02packages.details.txt.gz
Going to read /root/.cpan/sources/modules/02packages.details.txt.gz
  Database was generated on Thu, 21 Aug 2008 02:03:21 GMT
Fetching with LWP:
  ftp://perl.di.uminho.pt/pub/CPAN/modules/03modlist.data.gz
Going to read /root/.cpan/sources/modules/03modlist.data.gz
Going to write /root/.cpan/Metadata
Running look for module Math::Prime::XS

Trying to open a subshell in the build directory...
CPAN: Digest::MD5 loaded ok
Checksum for /root/.cpan/sources/authors/id/S/SC/SCHUBIGER/Math-Prime-XS-0.20.tar.gz ok
Scanning cache /root/.cpan/build for sizes
Math-Prime-XS-0.20/
Math-Prime-XS-0.20/Changes
Math-Prime-XS-0.20/lib/
Math-Prime-XS-0.20/lib/Math/
Math-Prime-XS-0.20/lib/Math/Prime/
Math-Prime-XS-0.20/lib/Math/Prime/XS.pm
Math-Prime-XS-0.20/ppport.h
Math-Prime-XS-0.20/MANIFEST
Math-Prime-XS-0.20/XS.xs
Math-Prime-XS-0.20/t/
Math-Prime-XS-0.20/t/pod-coverage.t
Math-Prime-XS-0.20/t/calc_primes.t
Math-Prime-XS-0.20/t/pod.t
```

```
Math-Prime-XS-0.20/t/00-load.t
Math-Prime-XS-0.20/INSTALL
Math-Prime-XS-0.20/Build.PL
Math-Prime-XS-0.20/META.yml
Math-Prime-XS-0.20/Makefile.PL
Math-Prime-XS-0.20/README
Removing previously used /root/.cpan/build/Math-Prime-XS-0.20
Working directory is /root/.cpan/build/Math-Prime-XS-0.20
```

Aunque la sesión ha sido arrancada como `root` podría haberla hecho como un usuario ordinario. El único objetivo era descargar la distribución de Math::Prime::XS y posicionarse en el correspondiente directorio.

```
root@orion:~/.cpan/build/Math-Prime-XS-0.20# ls -l
total 204
-r--r--r-- 1 csegura csegura    786 2008-05-14 15:30 Build.PL
-r--r--r-- 1 csegura csegura   1445 2008-05-14 15:30 Changes
-r--r--r-- 1 csegura csegura    290 2008-05-14 15:30 INSTALL
drwxr-xr-x 3 csegura csegura   4096 2008-05-14 15:30 lib
-r--r--r-- 1 csegura csegura    498 2008-05-14 15:30 Makefile.PL
-r--r--r-- 1 csegura csegura    172 2008-05-14 15:30 MANIFEST
-r--r--r-- 1 csegura csegura    560 2008-05-14 15:30 META.yml
-r--r--r-- 1 csegura csegura 154956 2008-05-14 15:30 ppport.h
-r--r--r-- 1 csegura csegura   5033 2008-05-14 15:30 README
drwxr-xr-x 2 csegura csegura   4096 2008-05-14 15:30 t
-r--r--r-- 1 csegura csegura   4843 2008-05-14 15:30 XS.xs
```

Construimos la distribución siguiendo el procedimiento habitual:

```
root@orion:~/.cpan/build/Math-Prime-XS-0.20# perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Math::Prime::XS
root@orion:~/.cpan/build/Math-Prime-XS-0.20# make
cp lib/Math/Prime/XS.pm blib/lib/Math/Prime/XS.pm
/usr/bin/perl /usr/local/share/perl/5.8.8/ExtUtils/xsubpp  \
            -typemap /usr/share/perl/5.8/ExtUtils/typemap  \
                          XS.xs > XS.xsc && mv XS.xsc XS.c
cc -c   -D_REENTRANT -D_GNU_SOURCE -DTHREADS_HAVE_PIDS -DDEBIAN -fno-strict-aliasing -pipe \
        -I/usr/local/include -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -O2 \
        -DVERSION=\"0.20\" -DXS_VERSION=\"0.20\" -fPIC "-I/usr/lib/perl/5.8/CORE"   XS.c
Running Mkbootstrap for Math::Prime::XS ()
chmod 644 XS.bs
rm -f blib/arch/auto/Math/Prime/XS/XS.so
cc  -shared -L/usr/local/lib XS.o  -o blib/arch/auto/Math/Prime/XS/XS.so
chmod 755 blib/arch/auto/Math/Prime/XS/XS.so
cp XS.bs blib/arch/auto/Math/Prime/XS/XS.bs
chmod 644 blib/arch/auto/Math/Prime/XS/XS.bs
Manifying blib/man3/Math::Prime::XS.3pm
```

El aspecto mas notable de esta construcción es que se ha llamado al compilador de `C`. Las opciones pasadas al compilador han sido las mismas que se usaron en la instalación de Perl en la plataforma en uso. Ahora cambiamos al directorio de construcción `blib`:

```
root@orion:~/.cpan/build/Math-Prime-XS-0.20# cd blib
```

En el directorio `arch` está la librería `.so` (shared object). En el directorio `lib` esta el módulo Perl `.pm` que contiene la interfaz Perl a las funciones C:

```
root@orion:~/.cpan/build/Math-Prime-XS-0.20/blib# tree
.
|-- arch
|   `-- auto
|       `-- Math
|           `-- Prime
|               `-- XS
|                   |-- XS.bs
|                   `-- XS.so
|-- bin
|-- lib
|   |-- Math
|   |   `-- Prime
|   |       `-- XS.pm
|   `-- auto
|       `-- Math
|           `-- Prime
|               `-- XS
|-- man1
|-- man3
|   `-- Math::Prime::XS.3pm
`-- script
```

Para construir el fichero `.par` usamos `zip` añadiendo los directorios `arch` y `lib`. los contenidos de

```
root@orion:~/.cpan/build/Math-Prime-XS-0.20/blib# zip -r /tmp/primexs.par arch/ lib/
  adding: arch/ (stored 0%)
  adding: arch/.exists (stored 0%)
  adding: arch/auto/ (stored 0%)
  adding: arch/auto/Math/ (stored 0%)
  adding: arch/auto/Math/Prime/ (stored 0%)
  adding: arch/auto/Math/Prime/XS/ (stored 0%)
  adding: arch/auto/Math/Prime/XS/.exists (stored 0%)
  adding: arch/auto/Math/Prime/XS/XS.so (deflated 59%)
  adding: arch/auto/Math/Prime/XS/XS.bs (stored 0%)
  adding: lib/ (stored 0%)
  adding: lib/Math/ (stored 0%)
  adding: lib/Math/Prime/ (stored 0%)
  adding: lib/Math/Prime/.exists (stored 0%)
  adding: lib/Math/Prime/XS.pm (deflated 59%)
  adding: lib/auto/ (stored 0%)
  adding: lib/auto/Math/ (stored 0%)
  adding: lib/auto/Math/Prime/ (stored 0%)
  adding: lib/auto/Math/Prime/XS/ (stored 0%)
  adding: lib/auto/Math/Prime/XS/.exists (stored 0%)
root@orion:~/.cpan/build/Math-Prime-XS-0.20/blib#
```

Es habitual que un fichero `.par` este fundamentalmente constituído por los contenidos del directorio `blib/` construido a partir de una distribución CPAN.

Ahora transferimos el fichero `/tmp/primexs.par` a una máquina que carece de los módulos en dicho fichero, pero que es binario-compatible con la máquina en la que se realizó la construcción:

```
lhp@nereida:~/Lperl/src$ perldoc -l Math::Prime::XS
No documentation found for "Math::Prime::XS".
```

Después de la transferencia estamos en condiciones de usar el módulo Math::Prime::XS utilizando via PAR la distribucion `primexs.par`:

```
lhp@nereida:~/Lperl/src$ scp orion:/tmp/primexs.par .
primexs.par                             100%   11KB  11.4KB/s   00:00
lhp@nereida:~/Lperl/src$ vi prime3.pl
lhp@nereida:~/Lperl/src$ cat -n prime3.pl
     1  #!/usr/bin/perl -I../lib -w
     2  use PAR "primexs.par";
     3  use Math::Prime::XS qw{:all};
     4
     5  @all_primes   = primes(9);
     6  print "@all_primes\n";
     7
     8  @range_primes = primes(4, 9);
     9  print "@range_primes\n";
```

El programa se ejecuta sin errores produciendo la salida esperada:

```
lhp@nereida:~/Lperl/src$ prime3.pl
2 3 5 7
5 7
```

Por supuesto, el éxito de esta ejecución depende de la compatibilidad binaria de ambas plataformas (`orion` y `nereida`)

Uso de Módulos via Web

Es posible utilizar ficheros PAR cuya localización está en una máquina remota:

```
use PAR "http://orion.pcg.ull.es/~casiano/primexs.par";
```

En este escenario PAR cargará los módulos solicitados desde el repositorio PAR situado en `http://foo/bar/`. Por supuesto debemos el fichero `.par` deberá estar en la máquina remota (en el ejemplo que sigue `orion`). Copiamos el `.par` construido anteriormente en el directorio que contiene nuestros HTML publicos:

```
casiano@orion:~/public_html$ cp /tmp/primexs.par .
casiano@orion:~/public_html$ pwd
/home/casiano/public_html
casiano@orion:~/public_html$ ls -ltr | tail -1
-rw-r--r--  1 casiano casiano         11685 2008-08-23 12:57 primexs.par
```

Ahora podemos ejecutar en la máquina cliente `nereida` un programa que hace uso del repositorio situado en `orion`:

```
lhp@nereida:~/Lperl/src$ cat -n prime4.pl
   1  #!/usr/bin/perl -I../lib -w
   2  use PAR "http://orion.pcg.ull.es/~casiano/primexs.par";
   3  use Math::Prime::XS qw{:all};
   4
   5  @all_primes   = primes(9);
   6  print "@all_primes\n";
   7
   8  @range_primes = primes(4, 9);
   9  print "@range_primes\n";
```

El programa se ejecuta normalmente:

```
lhp@nereida:~/Lperl/src$ prime4.pl
2 3 5 7
5 7
```

5.16. Instalación de Ejecutables con `pp`

La utilidad `pp` (*Perl packager*) crea *ejecutables standalone* utilizando el empaquetador de paquetes `PAR` (*Perl Archive Toolkit*).

Veamos un ejemplo de uso. Disponemos de un programa `temp.pl` que convierte temperaturas entre diferentes escalas. El programa hace uso de la interfaz gráfica `Tk`:

```
lhp@mymachine:~/Lperl/src$ sed -ne 1,11p temp.pl
#!/usr/bin/perl -w
# tempcon version 0.1      for Perl/Tk
# by Alan Ford <alan@whirlnet.demon.co.uk> 27/03/1999
# Allows conversion of temperatures between different scales

require 5.002;
use English;
use Tk;
use Tk::DialogBox;
# use strict;
sub convert ;
```

para empaquetarlo en un ejecutable usamos `pp`:

```
lhp@mymachine:~/Lperl/src$ pp temp.pl -o temperature
lhp@mymachine:~/Lperl/src$ ls -ltr temp*
-rwxr-xr-x  1 lhp lhp    7082 2006-06-15 12:52 temp.pl
-rwxr-xr-x  1 lhp lhp 3343850 2007-05-09 09:23 temperature
```

Ahora podemos transferir el programa a una máquina en la que ni siquiera esta instalado el toolkit `Tk`:

```
lhp@mymachine:~/Lperl/src$ sftp mylogin@etsii
Connecting to etsii...
sftp> put temperature
Uploading temperature to /home/mylogin/temperature
temperature               100% 3265KB   3.2MB/s   00:01
sftp>
```

La imagen en el enlace `pp.png` de la página de estos apuntes muestra el resultado de la ejecución en una máquina en la que no estan instaladas los módulos necesarios para la buena ejecución del programa. Observe en la terminal de la izquierda como `Tk` no está disponible (llamada a `perldoc -l`). El menú de la derecha muestra al programa ejecutándose en modo gráfico[2].

5.17. Construcción de un Módulo con `h2xs`

Construcción de la Jerarquía Mínima para una Distribución

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:

---

[2] La conexión se ha realizado usando el plugin Java que permite el acceso en modo gráfico via web a las máquinas de esta escuela

```
$ 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 5.14.1).

El Programa `Makefile.PL`

El programa perl `Makefile.PL` generado por `h2xs` se encarga de crear el fichero `Makefile`:

```
$ cat -n Makefile.PL
 1  use 5.008004;
 2  use ExtUtils::MakeMaker;
 3  # See lib/ExtUtils/MakeMaker.pm for details of how to influence
 4  # the contents of the Makefile that is written.
 5  WriteMakefile(
 6   NAME              => ’Parse::Yard’,
 7   VERSION_FROM      => ’lib/Parse/Yard.pm’, # finds $VERSION
 8   PREREQ_PM         => {}, # e.g., Module::Name => 1.1
 9   ($] >= 5.005 ?     ## Add these new keywords supported since 5.005
10     (ABSTRACT_FROM  => ’lib/Parse/Yard.pm’, # retrieve abstract from module
11     AUTHOR         => ’Lenguajes y Herramientas de Programacion <lhp@ull.es>’) : ()),
12  );
```

Parámetros de `WriteMakefile`

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, resumen o abstract del módulo. `ExtUtils::MakeMaker` busca en la documentación POD de la distribución por una línea que case con la expresión regular `/^($package\s-\s)(.*)/`. La costumbre/convenio es que esta sea la primera línea en la sección `=head1 NAME`. El contenido de `$2` se interpreta como *abstract*. Por ejemplo en la distribución de `Parse::Yapp` podemos ver que contiene la línea:

  ```
  nereida:~> sed -ne '/NAME/,/SYNOPSIS/p' `perldoc -l Parse::Yapp`
  =head1 NAME

  Parse::Yapp - Perl extension for generating and using LALR parsers.

  =head1 SYNOPSIS

  nereida:~> pmdesc Parse::Yapp
  Parse::Yapp - Perl extension for generating and using LALR parsers.
  ```

- `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
```

El Fichero `MANIFEST`

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.

`make MANIFEST`

`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 Fichero del Módulo

El programa `h2xs` creó en `lib/Parse/Yapp.pm` un esqueleto inicial del módulo para nosotros. Estos son sus contenidos:

```
package Parse::Yard;
use 5.008004;              # Versión mínima de Perl para trabajar
use strict;                # Por defecto = a la de la instalación
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__
```

El marcador `__END__` indica el final del código. El esqueleto construído por `h2xs` también contiene define la estructura de la documentación:

```
# 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
```

La Construcción de `blib`

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
```

Este paso cobra importancia cuando el módulo contiene partes escritas en lenguajes externos (por ejemplo en C) y es necesaria la construcción de librerías dinámicas conteniendo código objeto.

`make test`

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 podemos construir una versión para su distribución mediante:

```
> perl Makefile.PL # Crea el Makefile
> make
> make dist # o bien: make tardist
```

La distribución resultante puede ser instalada siguiendo los pasos explicados en la sección 5.14.1.

5.18. 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 que siguen a `__END__` en el esqueleto generado por `h2xs` (véase la sección 5.17) muestran el uso de `pod`. Cuando Perl encuentra un comando `pod` (todos los comandos comienzan por el signo igual) ignora las líneas a las que afecta. Para indicar el final de una zona de documentación se usa el comando `=cut` al comienzo de un nuevo párrafo.

Conversión de `pod` a Otros Formatos

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 -prefile preamble.tex lib/Parse/Eyapp.pm`

Para convertir a `pdf` puedes usar `pod2pdf.pl`[3]:

```
pod2pdf.pl [options] filename.pod >filename.pdf
perldoc -u <Module::Name> | pod2pdf.pl [options] >filename.pdf
```

Una herramienta que permite el proceso inverso, esto es pasar de formato `man` a otros formatos es `rman` [4].

Si se ha instalado la documentación de Perl en tu sistema, puedes acceder a ella utilizando `info`, `man`, `tkpod`[5] 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.

Texto verbatim

Cuando un párrafo esta sangrado (comienza con un espacio o tabulador) se reproduce exactamente como aparece.

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`.

---

[3] http://perl.jonallen.info/projects/pod2pdf

[4] http://polyglotman.sourceforge.net/

[5] Si se instaló `Tk::Pod`

- `=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`.

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)

5.19. 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 [14]:

- 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 [14].

5.19.1. 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`?

5.19.2. Versiones posteriores a la 5.8

El Directorio `t/`

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.

Los Programas de Prueba

Estos ficheros de prueba deberán ser programas Perl con el tipo `.t`. 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
```

El árbol de directorios creado es el que sigue:

```
$ tree
.
‘-- PL-Tutu
    |-- Changes
    |-- MANIFEST
    |-- Makefile.PL
    |-- README
    |-- lib
    |   ‘-- PL
    |       ‘-- Tutu.pm
    ‘-- t
        ‘-- PL-Tutu.t

4 directories, 6 files
```

Ejecución de las Pruebas

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
```

Contenidos de un Programa de Prueba

Los contenidos del fichero de prueba muestran que se hace uso de `Test::More`. `Test::More` provee servicios para la esritura de pruebas:

```
$ 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.
```

Para mas información escriba `perldoc Test::More`.

Nombres de las Pruebas

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

Introduciendo una Prueba Adicional El programa `02lex.t` pone a prueba una subrutina `Lexical::Analysis::scanner` que hemos escrito. Se supone que su ejecución con entrada `$a` debe producir como resultado que la lista `@PL::Tutu::tokens` contenga una secuencia dada de caracteres:

```
$ 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)
```

5.20. Práctica: Construcción de una Distribución

Construya siguiendo los pasos descritos en las secciones 5.17 y 5.18 una distribución conteniendo un módulo `Sub::Wrap` que provea una función `wrap` como la descrita en las secciones 4.15.9 y 4.15.10. Escriba al menos una prueba adicional (consulte `perldoc Test::More`). Compruebe que puede crear la distribución (`make dist`) e instalarla (`make; make test; make install`) como se hace con cualquier distribución CPAN (relea si es necesario la sección 5.14.1.

5.21. 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.

5.21.1. 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

$$f_k(c) = \text{máx}\{f_{k-1}(c), f_{k-1}(c - w_k) + p_k\} \tag{5.1}$$

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 5.1.

5.21.2. 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 `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()};
    17    my @p = @{shift()};
    18
    19    my $N = @w;
    20    my @f;
    21
    22    croak "Profits and Weights don't have the same size"
                                unless scalar(@w) == scalar(@p);
    23
    24    for my $c (0..$M) {
    25      $f[0][$c] = ($w[0] <= $c)? $p[0] : 0;
    26    }
    27
    28    for my $k (1..$N-1) {
    29      for my $c (0..$M) {
    30        my $n = $f[$k-1][$c];
    31        if ($c >= $w[$k]) {
    32          my $y = $f[$k-1][$c-$w[$k]]+$p[$k];
    33          $f[$k][$c] = ($n < $y)? $y : $n;
    34        }
    35        else { $f[$k][$c] = $n; }
    36      }
    37    }
    38    return @f;
    39  }
    40
    41  sub ReadKnap {
    42    my $filename = shift;
    43
    44    my $file = IO::File->new("< $filename");
    45    croak "Can't open $filename" unless defined($file);
    46    my (@w, @p);
    47
    48    my $N = <$file>; chomp($N);
    49    my $M = <$file>; chomp($M);
```

```
  for (0..$N-1) {
    $w[$_] = <$file>;
    $p[$_] = <$file>;
  }
  chomp @w; chomp @p;
  return ($M, \@w, \@p);
}

1;
__END__
```

5.21.3. 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
```

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.

5.21.4. 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$
 5  CVS/[^/]+$
 6  ^Makefile$
 7  /Makefile$
 8  ^blib/
 9  \.swp$
10  \.bak$
11  \.pdf$
12  \.ps$
13  pm_to_blib
14  .pdf$
```

Ahora podemos construir el fichero `MANIFEST` sin mas que hacer `make manifest`:

```
hp@nereida:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ make manifest
/usr/bin/perl "-MExtUtils::Manifest=mkmanifest" -e mkmanifest
Added to MANIFEST: Changes
Added to MANIFEST: lib/Algorithm/Knap01DP.pm
Added to MANIFEST: Makefile.PL
Added to MANIFEST: MANIFEST
Added to MANIFEST: MANIFEST.SKIP
Added to MANIFEST: README
Added to MANIFEST: t/01alltests.t
Added to MANIFEST: t/02bench.t
Added to MANIFEST: t/example.pl
Added to MANIFEST: t/knap.dat
Added to MANIFEST: t/knap21.dat
Added to MANIFEST: t/knap22.dat
Added to MANIFEST: t/knap23.dat
Added to MANIFEST: t/knap25.dat
Added to MANIFEST: t/knapanderson.dat
Added to MANIFEST: t/kwalitee.t
Added to MANIFEST: t/usealknap.pl
```

```
Added to MANIFEST: TODO
lhp@nereida:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ cat MANIFEST
Changes
lib/Algorithm/Knap01DP.pm
Makefile.PL
MANIFEST                        This list of files
MANIFEST.SKIP
README
t/01alltests.t
t/02bench.t
t/example.pl
t/knap.dat
t/knap21.dat
t/knap22.dat
t/knap23.dat
t/knap25.dat
t/knapanderson.dat
t/kwalitee.t
t/usealknap.pl
TODO
```

5.21.5. 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 bórralo y haz `make` de nuevo.

5.21.6. 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` un lenguaje que permite la serialización de estructuras de datos y la interoperabilidad de dichas estructuras entre diferentes lenguajes.

Haga `make metafile` para construir el fichero `META.yml`:

```
pl@nereida:~/src/perl/YappWithDefaultAction$ make metafile
rm -rf Parse-Eyapp-1.067
/usr/bin/perl "-MExtUtils::Manifest=manicopy,maniread" \
                -e "manicopy(maniread(),'Parse-Eyapp-1.067', 'best');"
mkdir Parse-Eyapp-1.067
mkdir Parse-Eyapp-1.067/examples
mkdir Parse-Eyapp-1.067/t
mkdir Parse-Eyapp-1.067/lib
mkdir Parse-Eyapp-1.067/lib/Parse
mkdir Parse-Eyapp-1.067/lib/Parse/Eyapp
Generating META.yml
```

5.21.7. Las Pruebas

Testing versus Debugging

El tiempo invertido en *Comprobar* (*testing*) y el tiempo invertido en *Depurar* (*debugging*) mantiene una relación inversa. *Cuanto menor es el tiempo invertido en la preparación de pruebas mayor será nuestro tiempo de depuración.* Cuanto mejor sea nuestra suite de pruebas menor será el tiempo que necesitemos para fijar errores en nuestros programas.

Los Objetivos de las Pruebas

Las pruebas no aseguran la corrección de nuestro programa. Las pruebas descubren errores. Una prueba tiene éxito cuando falla y demuestra la presencia de un error.

En palabras de Bruce Eckel (*Thinking in Java*):

> *You can write all the prose, or create all the diagrams you want, describing how a class should behave and what it looks like, but nothing is as real as a set of tests. The former is a wish list, but the tests are a contract that is enforced by the compiler and the running program. It's hard to imagine a more concrete description of a class than the tests*

Cuando escriba las pruebas invierta su escala de valores: *Alégrese cuando una prueba descubre un error.* Es peor que el error esté ahí y no sea descubierto.

Que Comprobar

- Convertir cualquier fallo que encontremos en una prueba
- Comprobar el funcionamiento en mas de una plataforma
- Dar valores de entrada erróneos o en el límite. Por ejemplo:
  - Los valores máximo y mínimo posibles
  - Cadenas vacías
  - Cadenas multilínea
  - Entradas con caracteres de control o no ASCII (Unicode p.ej.)
  - Valores como `'0'`, `'0E0'`, listas vacías, hashes vacíos, etc.
  - Llamar sin argumentos a una rutina que los espera y viceversa
  - Llamar a una rutina con mas argumentos de los que espera
  - Llamar a una rutina con los argumentos en orden incorrecto
- Estudie las interacciones y dependencias entre recursos que se da por sentado que están (pero que a veces pueden no estar. Por ejemplo, un fichero con cuya existencia se cuenta)
- Estudie la dependencia de versiones del software instalado
- Compruebe que toda subrutina se prueba al menos una vez. Prepare para cada subrutina importante al menos una prueba que cubra el caso promedio y otra que compruebe su funcionamiento en casos límite.
- Estudie la escalabilidad de la distribución con valores tendiendo al límite. Ficheros grandes, números grandes, etc.

La Preparación de 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 [15] en el que aparece el correspondiente problema. Puede encontrar el texto completo del programa `01MartelloAndTothBook.t` en la sección 10.1.

Comprobar que el Módulo se Carga

Realizaremos 11 pruebas de las que la primera es comprobar que el módulo se carga correctamente (línea 8).

```
$ cat -n 01MartelloAndTothBook.t
 1 # Before ‘make install’ is performed this script should be runnable with
 2 # ‘make test’. After ‘make install’ it should work as ‘perl Algorithm-Knap01DP.t’
 3
 4 #########################
 5 use strict;
 6 use Test::More  tests => 11;
 7
 8 BEGIN { use_ok(’Algorithm::Knap01DP’, qw/Knap01DP ReadKnap/); }
```

Establecer un Correspondencia Entrada/Salida Correcta

Para el establecimiento de las pruebas para una subrutina dada es necesario definir las estructuras de datos para la llamada y las correspondientes que resultan de una correcta ejecución de dicha subrutina:

```
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];
```

En nuestro caso esa definición de la correspondencia entre una entrada y su correspondiente salida correcta se traduce en:

- La variable `@inputfiles` contiene los nombres de los ficheros de prueba.
- Las variables `$knap21` ... `$knap25` contienen estructuras de datos que definen los problemas: capacidad de la mochila, vector de pesos y vector de beneficios.
- La variable `@sol` las soluciones óptimas a esos problemas.

Uso de `Data::Dumper` y del Depurador para la Preparación de las Pruebas

En el proceso de elaboración de una prueba para una subrutina es necesario tener una descripción Perl de

- Los datos de entrada
- Los resultados devueltos

Esta descripción puede obtenerse usando Data::Dumper o el depurador o ambas cosas.

```
pp2@nereida:/tmp/Algorithm-Knap01DP-0.01/t$ perl -I../lib "-MAlgorithm::Knap01DP=Knap01DP,ReadKnap" -wde 0
main::(-e:1):   0
  DB<1> $x = ReadKnap("knap22.dat")
  DB<2> use Data::Dumper
  DB<3> Dumper($x)
  DB<4> p Dumper($x)
$VAR1 = [
          ’70’,
          ’20’,
          ’39’,
          ’37’,
          ’7’,
          ’5’,
          ’10’
        ];

  DB<5> x $x
0  ARRAY(0x840a11c)
   0  70
   1  20
   2  39
   3  37
   4  7
   5  5
   6  10
```

Cuando la función `ReadKnap` lee un fichero de datos devuelve una estructura como la descrita.

De hecho, es usando el depurador, cortando y pegando que hemos construido parte del código de pruebas definiendo las estructuras de datos `$knapXX`:

```
### main
my @inputfiles = qw/knap21.dat  knap22.dat  knap23.dat  knap25.dat/;
my @sol = (280, 107, 150, 900);
my $knap21 = [’102’, [ ’2’, ’20’, ’20’, ’30’, ’40’, ’30’, ’60’, ’10’ ],
                     [ ’15’, ’100’, ’90’, ’60’, ’40’, ’15’, ’10’, ’1’ ]];
my $knap22 = [’50’,  [ ’31’, ’10’, ’20’, ’19’, ’4’, ’3’, ’6’ ],
                     [ ’70’, ’20’, ’39’, ’37’, ’7’, ’5’, ’10’ ]];
my $knap23 = [’190’, [ ’56’, ’59’, ’80’, ’64’, ’75’, ’17’ ],
                     [ ’50’, ’50’, ’64’, ’46’, ’50’, ’5’ ]];
my $knap25 = [’104’, [ ’25’, ’35’, ’45’, ’5’, ’25’, ’3’, ’2’, ’2’ ],
                     [ ’350’, ’400’, ’450’, ’20’, ’70’, ’8’, ’5’, ’5’ ]];

my $knapsackproblem = [$knap21, $knap22, $knap23, $knap25];
```

Las Funciones `is_deeply` e `is`

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`.

```
my $i = 0;
my ($M, $w, $p);
my @f;

# Now 2*@inputfiles = 8 tests
for my $file (@inputfiles) {
  ($M, $w, $p) = ReadKnap((-e "t/$file")?"t/$file":$file);
  is_deeply($knapsackproblem->[$i], [$M, $w, $p], "ReadKnap $file");
  my $N = @$w;
  @f = Knap01DP($M, $w, $p);
  is($sol[$i++], $f[$N-1][$M], "Knap01DP $file");
}
```

Para mas funciones de *Comparación Profunda* véase el módulo Test::Deep.

Como Evitar la Muerte (de un Programa)

¿Cual es el comportamiento correcto de una función cuando es llamada con argumentos erróneos? Parece razonable que el comportamiento de tal función sea advertir al programador de la conducta anómala y en su caso detener la ejecución del programa llamando a `die` o a `croak`.

¿Que ocurre si dentro de un programa de prueba `.t` queremos comprobar el funcionamiento de tal subrutina bajo situaciones de "stress" como esta, por ejemplo, con argumentos erróneos?

La respuesta es que - si se hace una llamada convencional - la llamada a `die` dentro de la subrutina investigada provocará la "muerte prematura" del programa de pruebas.

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:

```
sub Knap01DP {
  my $M = shift;
  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);

  ....................................................
  return @f;
}

sub ReadKnap {
  my $filename = shift;

  my $file = IO::File->new("< $filename");
  croak "Can't open $filename" unless defined($file);
  ............

```

```
  return ($M, \@w, \@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`:

```
# test to check when weights and profits do not have the same size
$M = 100; @$w = 1..5; @$p = 1..10;
eval { Knap01DP($M, $w, $p) };
like $@, qr/Profits and Weights don’t have the same size/;
```

Ahora la llamada a `croak` dentro de `Knap01DP` sólo produce la finalización del `eval`. El mensaje emitido por `croak` o `die` queda en la variable especial `$@`.

La función `like` comprueba que el primer argumento casa con la expresión regular especificada en el segundo argumento.

El Módulo `Test::Exception`

Una alternativa es usar el módulo `Test::Exception` si está instalado:

```
BEGIN {
  $test_exception_installed = 1;
  eval { require Test::Exception };
  $test_exception_installed = 0 if $@;
}
```

para posteriormente aplicar la prueba:

```
SKIP: {
  skip "Test::Exception not installed", 1 unless $test_exception_installed;
  Test::Exception::lives_ok
              { Parse::Eyapp->new_grammar( input=>$translationscheme) }
              ’No errors in input translation scheme’;
}
```

Pruebas `SKIP`

El código anterior muestra una prueba `SKIP`. Una prueba `SKIP` declara un bloque de pruebas que - bajo ciertas circunstancias - 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).

Pruebas `TODO`

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.

```
TODO: {
  local $TODO = "Randomly generated problem";
  can_ok(’Algorithm::Knap01DP’, ’GenKnap’);
}
```

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.

`Devel::Cover`: Estudio del Cubrimiento de las Pruebas

Instale el módulo Devel::Cover. El módulo `Devel::Cover` ha sido escrito por Paul Johnson y proporciona estadísticas del cubrimiento alcanzado por una ejecución. Para usarlo siga estos pasos:

```
hp@nereida:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ cover -delete
Deleting database /home/lhp/projects/perl/src/threads/knapsack/Algorithm-Knap01DP/cover_db
lhp@nereida:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ make HARNESS_PERL_SWITCHES=-MDevel::Cover tes
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib', 'blib/arch')"
t/01alltests....ok
t/02bench.......ok
All tests successful.
Files=2, Tests=16, 18 wallclock secs (17.05 cusr +  0.75 csys = 17.80 CPU)

Writing HTML output to /home/lhp/projects/perl/src/threads/knapsack/Algorithm-Knap01DP/cover_db/coverage.h
done.
```

La ejecución toma ahora mas tiempo. Al ejecutar `cover` de nuevo obtenemos una tabla con las estadísticas de cubrimiento:

```
lhp@nereida:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ cover
Reading database from /home/lhp/projects/perl/src/threads/knapsack/Algorithm-Knap01DP/cover_db


---------------------------- ------ ------ ------ ------ ------ ------ ------
File                           stmt   bran   cond    sub    pod   time  total
---------------------------- ------ ------ ------ ------ ------ ------ ------
...lib/Algorithm/Knap01DP.pm  100.0   82.1   37.5  100.0    0.0  100.0   88.8
Total                         100.0   82.1   37.5  100.0    0.0  100.0   88.8
---------------------------- ------ ------ ------ ------ ------ ------ ------


Writing HTML output to /home/lhp/projects/perl/src/threads/knapsack/Algorithm-Knap01DP/cover_db/coverage.h
done.
```

El HTML generado nos permite tener una visión mas detallada de los niveles de cubrimiento.

```
lhp@nereida:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ cd cover_db/
lhp@nereida:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP/cover_db$ ls -l
total 68
-rw-r--r--  1 lhp lhp  3424 2007-05-16 15:48 blib-lib-Algorithm-Knap01DP-pm--branch.html
-rw-r--r--  1 lhp lhp  3127 2007-05-16 15:48 blib-lib-Algorithm-Knap01DP-pm--condition.html
-rw-r--r--  1 lhp lhp 34909 2007-05-16 15:48 blib-lib-Algorithm-Knap01DP-pm.html
-rw-r--r--  1 lhp lhp  2207 2007-05-16 15:48 blib-lib-Algorithm-Knap01DP-pm--subroutine.html
-rw-r--r--  1 lhp lhp  3345 2007-05-16 15:48 cover.12
-rw-r--r--  1 lhp lhp  2004 2007-05-16 15:48 coverage.html
-rw-r--r--  1 lhp lhp  1348 2007-05-16 15:48 cover.css
drwx------  2 lhp lhp  4096 2007-05-16 15:48 runs
drwx------  2 lhp lhp  4096 2007-05-16 15:48 structure
```

Para mejorar el cubrimiento de tu código comienza por el informe de cubrimiento de subrutinas. Cualquier subrutina marcada como no probada es un candidato a contener errores o incluso a ser *código muerto*.

Estudio del Perfil de Rendimiento

Se conoce con el nombre de *perfilado* o *profiling* de un programa al estudio de su rendimiento mediante un programa (conocido como *profiler*) que monitoriza la ejecución del mismo mediante una técnica que interrumpe cada cierto tiempo el programa para comprobar en que punto de la ejecución se encuentra. Las estadísticas acumuladas se vuelcan al final de la ejecución en un fichero que puede ser visualizado mediante la aplicación apropiada.

En Perl hay varios módulos que permiten realizar profiling. El mas antiguo es `Devel::DProf`. La aplicación para visualizar los resultados se llama `dprofpp`. Sigue un ejemplo de uso:

```
lhp@nereida:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP/t$ time perl -MDevel::Profiler usealknap.pl kn
14,5,11
14,5,3,8
14,2,11,3

real    0m0.065s
user    0m0.060s
sys     0m0.000s
```

La ejecución crea un fichero `tmon.out`:

```
lhp@nereida:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP/t$ ls -ltr | tail -1
-rw-r--r--  1 lhp lhp  9876 2007-05-16 16:10 tmon.out
```

Al ejecutar `dprofpp` se muestran las subrutinas ordenadas según su consumo de CPU:

```
lhp@nereida:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP/t$ dprofpp tmon.out
Total Elapsed Time = 0.002055 Seconds
  User+System Time =  5.5e-05 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 0.00   0.000  0.000     52   0.0000 0.0000  Algorithm::Knapsack::_knapsack
 0.00   0.000  0.000      1   0.0000 0.0001  Algorithm::Knapsack::compute
 0.00   0.000  0.000      1   0.0000 0.0000  Algorithm::Knapsack::solutions
 0.00   0.000  0.000      1   0.0000 0.0000  Algorithm::Knapsack::new
```

5.21.8. Formas de Ejecutar las Pruebas

Cada vez que hacemos `make test` observamos que se ejecuta el siguiente comando:

```
$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" \
                                "-e" "test_harness(0, 'blib/lib', 'blib/arch')" \
                                t/*.t
```

La Subrutina `test_harness`

El módulo `ExtUtils::Command::MM` de la distribución ExtUtils::MakeMaker 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)
```

La Subrutina `runtests`

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.

El Protocolo TAP

Las pruebas deben ajustar su salida al protocolo TAP: Test All Protocol 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`

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)
```

Si se quieren ejecutar las pruebas de un cierto programa de prueba `t/*.t` mientras se está editando con vim se puede poner el siguiente atajo en el fichero `~/.vimrc`[6]:

```
map ,t <Esc>:!prove -vl %<CR>
```

El objetivo `testdb`

El objetivo `testdb` del `Makefile` generado permite correr las pruebas bajo el control del depurador:

```
pp2@nereida:~/LGRID_Machine$ make testdb TEST_FILE=t/13pipes.t
PERL_DL_NONLAZY=1 /usr/bin/perl -d "-Iblib/lib" "-Iblib/arch" t/13pipes.t

Loading DB routines from perl5db.pl version 1.28
Editor support available.

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

---

[6] Se asume que estás editando el programa de prueba en la raíz del proyecto

```
1..15
ok 1 - use GRID::Machine;
main::(t/13pipes.t:6):  my $test_exception_installed;
main::(t/13pipes.t:7):  BEGIN {
  DB<1> c
Name "DB::single" used only once: possible typo at blib/lib/GRID/Machine/REMOTE.pm line 569.
ok 2 - No fatals creating a GRID::Machine object
ok 3 - No fatals opening not redirected output pipe
ok 4 - No fatals sending to pipe 10
ok 5 - No fatals sending to pipe 9
ok 6 - No fatals sending to pipe 8
ok 7 - No fatals sending to pipe 7
ok 8 - No fatals sending to pipe 6
ok 9 - No fatals sending to pipe 5
ok 10 - No fatals sending to pipe 4
ok 11 - No fatals sending to pipe 3
ok 12 - No fatals sending to pipe 2
ok 13 - No fatals sending to pipe 1
ok 14 - No fatals sending to pipe 0
0
1
2
3
4
5
6
7
8
9
10
ok 15 - No fatals closing pipe
Debugged program terminated.  Use q to quit or R to restart,
  use o inhibit_exit to avoid stopping after program termination,
  h q, h R or h o to get additional info.
  DB<1> q
pp2@nereida:~/LGRID_Machine$
```

5.21.9. 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`.

5.21.10. Profundizando en `Makefile.PL`

Interrogando al Usuario

`Makefile.PL` es un programa Perl y por tanto puede - en caso de duda - cuestionar al usuario sobre decisiones que se refieren a la instalación. El siguiente ejemplo pertenece a la distribución de Net::SSH::Perl:

```
# $Id: Makefile.PL,v 1.28 2005/02/20 19:56:02 dbrobins Exp $

use strict;
use 5.006;
use ExtUtils::MakeMaker qw(prompt WriteMakefile);

my %prereq = (
        'Digest::MD5'       => 0,
        'IO::Socket'        => 0,
);

my %SSH_PREREQ = (
    1 => {
        'String::CRC32'     => '1.2',
        'Math::GMP'         => '1.04',
        'Scalar::Util'      => 0,
    },

    2 => {
        'Digest::SHA1'      => '2.10',
        ..................  => ......,
    },
);
```

```
$SSH_PREREQ{3} = { map %{$SSH_PREREQ{$_}}, 1..2 };

print<<MSG;
This is Net::SSH::Perl.

As of version 1.00, Net::SSH::Perl supports both the SSH1 and
SSH2 protocols natively. The two protocols have different
module prerequisitives, so you need to decide which protocol(s)
................................................................

MSG

printf "    [%d] SSH%d\n", $_, $_ for 1..2;
printf "    [3] Both SSH1 and SSH2\n";

my $p = prompt("\nWhich protocol(s) do you plan to use?", 3);
print "\n";

@prereq{keys %{$SSH_PREREQ{$p}}} = values %{$SSH_PREREQ{$p}};

my @cryptmod = (
    [ 'IDEA', 'Crypt::IDEA' ],
    [ 'DES','Crypt::DES' ],
    [ 'DES3', 'Crypt::DES' ],
    [ 'Blowfish', 'Crypt::Blowfish' ],
    [ 'RC4', '' ],
);

print<<MSG;
Some of the Net::SSH::Perl ciphers depend on a Crypt:: module from
....................................................................

MSG

my $i = 1;
for my $ciph(sort { $a->[0] <=> $b->[0] } @cryptmod) {
        printf "    [%d] %s\n", $i++, $ciph->[0];
}
my $c = prompt("\nEnter your choices, separated by spaces:", 1);
print "\n";

for my $id(split /\s+/, $c) {
        next unless $cryptmod[$id-1]->[1];
        $prereq{$cryptmod[$id-1]->[1]} = '0';
}

print "Checking for optional modules\n\n";

unless (have_module('Digest::BubbleBabble', 0.01)) {
        print<<MSG, "\n";
Digest::BubbleBabble is required if you want to generate bubble babble
key fingerprints with pssh-keygen.
MSG
```

```
        if (read_yes_or_no("Would you like to install it? (y/n)", "y")) {
                $prereq{'Digest::BubbleBabble'} = '0.01';
        }
        print "\n";
}

unless (have_module('Crypt::RSA', 1.37)) {
        print<<MSG, "\n";
Crypt::RSA is required if you wish to use the ssh-rsa public
key algorithm (ssh-dss is used by default).
MSG
        if (read_yes_or_no("Would you like to install it? (y/n)", "y")) {
                $prereq{'Crypt::RSA'} = '1.37';
        }
        print "\n";
}

WriteMakefile(
    NAME => 'Net::SSH::Perl',
    DISTNAME => 'Net-SSH-Perl',
    VERSION_FROM => 'lib/Net/SSH/Perl.pm',
    PREREQ_PM    => \%prereq,
    AUTHOR => 'David Robins <dbrobins@cpan.org>',
    ABSTRACT => 'Perl client interface to SSH',
);

sub read_yes_or_no {
    my($prompt, $def) = @_;
    my $ans = prompt($prompt, $def);
    $ans =~ /^y/i;
}

sub have_module {
    my($name, $ver) = @_;
    eval("use $name" . ($ver ? " $ver;" : ";"));
    !$@;
}
```

El ejemplo muestra el uso de la función `prompt` que permite preguntar al usuario.

5.21.11. Comprobando la Distribución con `Test::Kwalitee`

Una vez que nuestro desarrollo esta completo podemos construir la distribución:

```
lhp@nereida:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ make dist
....
lhp@nereida:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ ls -ltr | tail -1
-rw-r--r--  1 lhp lhp  5653 2007-05-16 16:38 Algorithm-Knap01DP-0.25.tar.gz
```

¿Esta la distribución en condiciones de ser distribuída? El módulo `Test::Kwalitee` ayuda a comprobar algunos aspectos. Prepare un programa como este:

```
lhp@nereida:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ cat -n kwalitee
 1  #!/usr/bin/perl -w
```

```
use Test::More;

eval { require Test::Kwalitee; Test::Kwalitee->import() };

plan( skip_all => 'Test::Kwalitee not installed; skipping' ) if $@;
```

Ejecute el programa en el lugar en el que se encuentra su distribución:

```
hp@nereida:~/Lperl/src/threads/knapsack/Algorithm-Knap01DP$ perl kwalitee
1..13
ok 1 - extractable
ok 2 - has_readme
ok 3 - has_manifest
ok 4 - has_meta_yml
ok 5 - has_buildtool
ok 6 - has_changelog
ok 7 - no_symlinks
ok 8 - has_tests
ok 9 - proper_libs
ok 10 - no_pod_errors
ok 11 - use_strict
not ok 12 - has_test_pod
#   Failed test 'has_test_pod'
#   at /usr/local/share/perl/5.8.8/Test/Kwalitee.pm line 101.
# Add a test using Test::Pod to check for pod correctness.\
     Doesn't include a test for pod correctness (Test::Pod)
not ok 13 - has_test_pod_coverage
#   Failed test 'has_test_pod_coverage'
#   at /usr/local/share/perl/5.8.8/Test/Kwalitee.pm line 101.
# Add a test using Test::Pod::Coverage to check for POD coverage. \
  Doesn't include a test for pod coverage (Test::Pod::Coverage)
# Looks like you failed 2 tests of 13.
```

5.21.12. Comprobando la Portabilidad del Código

Para comprobar la portabilidad del código establezca un sistema de autentificación automática via `ssh` usando pares de claves privada-pública entre la máquina en la que desarrolla y cada una de las máquinas en las que desea probar su aplicación.

Autentificación Automática

SSH es un protocolo de red y un conjunto de estándares que permiten una conexión encriptada entre dos computadoras. Usa criptografía de clave pública para autentificar al computador y al usuario. Por defecto suele usar el puerto TCP 22.

SSH soporta autentificación basada en RSA. *RSA* fué el primer algoritmo publicado que permite el encriptado y la firma digital. Se cree que es seguro si las claves son lo suficientemente largas. Se usa aún en comercio electrónico. Una opción alternativa es *DSA* que corresponde a *Digital Signature Algorithm*. DSA es propiedad del gobierno de los Estados Unidos de America.

Hay criptosistemas -a los que RSA pertenece - en los cuales el encriptado y desencriptado se hace utilizando claves separadas y no es posible derivar la clave de desencriptado del conocimiento de la clave de encriptado. El cliente utiliza un par de claves pública/privada para la autentificación. El servidor sólo conoce la clave pública. Funciona como una pareja llave/cerradura en la que el conocimiento de la cerradura no permite deducir la forma de la llave.

Generación de una Pareja de Claves Pública-Privada

Para la generación de una pareja de claves pública-privada se realiza ejecutando en el cliente `ssh-keygen`:

```
$ ssh-keygen  -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/user/.ssh/id_rsa.
Your public key has been saved in /home/user/.ssh/id_rsa.pub.
The key fingerprint is:
xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx user@machine
```

Las claves se almacenan por defecto en `~/.ssh/`, quedando el directorio así:

```
$ ls -l
total 12
-rw-------  1 user user  883 2005-08-13 14:16 id_rsa
-rw-r--r--  1 user user  223 2005-08-13 14:16 id_rsa.pub
-rw-r--r--  1 user user 1344 2005-08-04 02:14 known_hosts
```

Los ficheros `id_rsa` e `id_rsa.pub` contienen respectivamente las claves privada y pública. El fichero `known_hosts` contiene la lista de las claves públicas de las máquinas reconocidas.

Transferencia de la Clave Pública al Servidor

Ahora se debe copiar la clave pública al servidor, al fichero `~/.ssh/authorized_keys`. Para ello se utiliza el comando `ssh-copy-id`:

```
$ssh-copy-id -i ~/.ssh/id_rsa.pub user@machine1
$ssh-copy-id -i ~/.ssh/id_rsa.pub user@machine2
```

`ssh-copy-id` es un script que se conecta a la máquina y copia el archivo (indicado por la opción `-i`) en `~/.ssh/authorized_keys`, y ajusta los permisos a los valores adecuados.

Si no se dispone del programa `ssh-copy-id` se puede realizar una copia manual a la máquina remota del fichero conteniendo la clave pública (por ejemplo usando `scp` o `sftp`) y añadir su contenido al fichero `~/.ssh/authorized_keys`.

Ahora la conexión debería funcionar sin necesidad de introducir la clave. Si no es así es posible que sea un problema de permisos en los ficheros. Los permisos correctos deben ser similares a estos:

```
    $ chmod go-w $HOME $HOME/.ssh
    $ chmod 600 $HOME/.ssh/authorized_keys
```

Controlando el Login de Usuario

Es posible indicarle a `ssh` nuestra identidad en máquinas remotas usando el fichero de configuración `~/.ssh/config`:

```
hp@nereida:~/Lperl/src/perl_networking/ch2$ cat -n ~/.ssh/config
     1  # man  ssh_config
     2  Host machine1.domain
     3  user casiano
     4
     5  Host machine2.domain
     6  user pp2
```

Para saber más sobre `ssh` lea la sección Conexiones con ssh en los apuntes de Programación en Paralelo II.

Ejecutando las Pruebas en un Conjunto de Máquinas

El siguiente programa muestra un guión sencillo que permite ejecutar las pruebas en un conjunto de máquinas remotas con las que se ha establecido un sistema de autentificación automática. Al ejecutar el programa se transfiere la distribución, se desempaqueta y se pasan las pruebas sobre cada máquina:

```
lhp@nereida:~/Lperl/src/perl_testing_adn_examples/chapter_03/Algorithm-Knap01DP-0.25$ make remotetest
remotetest.pl Algorithm-Knap01DP-0.25.tar.gz orion beowulf
************orion************
Checking if your kit is complete...
Looks good
Writing Makefile for Algorithm::Knap01DP
cp lib/Algorithm/Knap01DP.pm blib/lib/Algorithm/Knap01DP.pm
Manifying blib/man3/Algorithm::Knap01DP.3pm
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" \
       "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/01alltests....ok
t/02bench.......ok
        1/2 skipped: Algorithm::Knapsack not installed
All tests successful, 1 subtest skipped.
Files=2, Tests=16,  0 wallclock secs ( 0.10 cusr +  0.01 csys =  0.11 CPU)
************beowulf************
Checking if your kit is complete...
Looks good
Writing Makefile for Algorithm::Knap01DP
cp lib/Algorithm/Knap01DP.pm blib/lib/Algorithm/Knap01DP.pm
Manifying blib/man3/Algorithm::Knap01DP.3pm
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" \
       "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/01alltests....ok
t/02bench.......ok
        1/2 skipped: various reasons
All tests successful, 1 subtest skipped.
Files=2, Tests=16,  0 wallclock secs ( 0.11 cusr +  0.01 csys =  0.12 CPU)
lhp@nereida:~/Lperl/src/perl_testing_adn_examples/chapter_03/Algorithm-Knap01DP-0.25$
```

Añadiendo Objetivos al `Makefile` Construido

Es posible entonces establecer la ejecución del guión anterior extendiendo `Makefile.PL` con una función `MY::postamble`. La cadena retornada por esta función es añadida al `Makefile` construido:

```
lhp@nereida:$ cat -n Makefile.PL
 1  use ExtUtils::MakeMaker;
 2  # See lib/ExtUtils/MakeMaker.pm for details of how to influence
 3  # the contents of the Makefile that is written.
 4  WriteMakefile(
 5      NAME              => 'Algorithm::Knap01DP',
 6      VERSION_FROM      => 'lib/Algorithm/Knap01DP.pm', # finds $VERSION
 7      PREREQ_PM         => {}, # e.g., Module::Name => 1.1
 8      ($] >= 5.005 ?     ## Add these new keywords supported since 5.005
 9        (ABSTRACT_FROM  => 'lib/Algorithm/Knap01DP.pm', # retrieve abstract from module
10         AUTHOR         => 'Lenguajes y Herramientas de Programacion <lhp@>') : ()),
11  );
12
13  sub MY::postamble {
14    my @machines = qw( orion beowulf);
```

```

return <<"EOT";
remotetest:
        remotetest.pl \${DISTVNAME}.tar.gz @machines
EOT
}
```

Elaborando una Ejecución Remota

El siguiente programa `remotetest.pl` copia para cada máquina la distribucion y ejecuta las pruebas en la máquina remota. Forma parte de la distribución GRID::Machine disponible en CPAN.

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

eval 'exec /usr/bin/perl -w -S $0 ${1+"$@"}'
    if 0; # not running under some shell
use strict;
use GRID::Machine;

....................................................................

my $dist = shift or die "Usage:\n$0 distribution.tar.gz machine1 machine2 ... \n";

die "No distribution $dist found\n" unless -r $dist;

  die "Distribution does not follow standard name convention\n"
unless $dist =~ m{([\w.-]+)\.tar\.gz$};
my $dir = $1;

die "Usage:\n$0 distribution.tar.gz machine1 machine2 ... \n" unless @ARGV;
for my $host (@ARGV) {

  my $m = eval {
    GRID::Machine->new(host => $host)
  };

  warn "Cant' create GRID::Machine connection with $host\n", next unless UNIVERSAL::isa($m, 'GRID::Ma

  my $r = $m->eval(q{
      our $tmpdir = File::Temp::tempdir;
      chdir($tmpdir) or die "Can't change to dir <$tmpdir>\n";
    }
  );

  warn($r),next unless $r->ok;

  my $preamble = $host.".preamble.pl";
  if (-r $preamble) {
    local $/ = undef;

    my $code = slurp_file($preamble);
    $r = $m->eval($code);
    warn("Error in $host preamble: $r"),next unless $r->ok;
  }
```

```

 $m->put([$dist]) or die "Can’t copy distribution in $host\n";

 $r = $m->eval(q{
   my $dist = shift;

   eval(’use Archive::Tar’);
   if (Archive::Tar->can(’new’)) {
    # Archive::Tar is installed, use it
    my $tar = Archive::Tar->new;
    $tar->read($dist,1) or die "Archive::Tar error: Can’t read distribution $dist\n";
    $tar->extract() or die "Archive::Tar error: Can’t extract distribution $dist\n";
   }
   else {
    system(’gunzip’, $dist) or die "Can’t gunzip $dist\n";
    my $tar = $dist =˜ s/\.gz$//;
    system(’tar’, ’-xf’, $tar) or die "Can’t untar $tar\n";
   }
  },
  $dist # arg for eval
 );

 warn($r), next unless $r->ok;

 $m->chdir($dir)->ok or do {
  warn "$host: Can’t change to directory $dir\n";
  next;
 };

 print "************$host************\n";
 next unless $m->run(’perl Makefile.PL’);
 next unless $m->run(’make’);
 next unless $m->run(’make test’);

 # Clean files
 $m->eval( q{
  our $tmpdir;
  chdir "$tmpdir/..";
  system (’rm -fR $dir’);
 });
}
```

5.21.13. Práctica: Pruebas

Aunque la descripción de los objetivos a realizar en esta práctica se refieren a la distribución que resuelve el problema de la mochila puede elegir cualquier otro algoritmo. Lo importante es que la distribución cubra cada uno de los tópicos expuestos en la lista:

- Lea los siguientes documentos
    - `Test::Tutorial`: `Test::Tutorial - A tutorial about writing really basic tests` (En CPAN Test::Tutorial) por Michael Schwern.
    - Test Now, test Forever ” del libro de Scott [14].
    - Testing Tools por Michael Schilli

    - Perl Testing Reference Card por Ian Langworth.
    - Chapter 4: Distributing Your Tests (and Code) del libro Perl Testing: A Developer's Notebook
    - Lea 431702

- Lea 536312

- Añada pruebas que comprueben el funcionamiento de `ReadKnap` cuando los ficheros no existen o no contienen los datos deseados (no tienen números, contienen números negativos, etc.).

- 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 llamar a `GenKnap` dos veces y comprobar que los problemas generados son distintos.

- Añada a la distribución de `Algorithm::Knap01DP` un ejecutable 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.

- Compruebe la documentación usando el módulo Test::Pod de Andy Lester. Instálelo si es necesario.

- 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.

- Utilice el módulo Test::Warn para comprobar que los mensajes de warning (uso de `warn` and `carp`) se muestran correctamente.

- Comprueb la "Kalidad" de su distribución

- Haga un análisis de cubrimiento

- Haga un análisis de rendimiento

5.21.14. El módulo `Test::LectroTest`

Generación de Pruebas a Partir de la Especificación de Propiedades

El módulo Test::LectroTest por Tom Moertel permite la generación de tests mediante la especificación de propiedades. Así en vez de escribir algo como:

```
use Test::More tests => 4;
is( sqrt(0), 0, "sqrt(0) == 0" );
is( sqrt(1), 1, "sqrt(1) == 1" );
is( sqrt(4), 2, "sqrt(4) == 2" );
is( sqrt(9), 3, "sqrt(9) == 3" );
```

podemos hacer:

```
nereida:~/projects/coro/xpp/check> cat -n electrotest.pl
 1  #!/usr/bin/perl -w
 2  use Test::LectroTest;
 3  Property {
 4     ##[ x <- Float(range=>[0,1_000]) ]##
 5     sqrt( $x * $x ) == $x;
 6  }, name => "sqrt satisfies defn of square root";
 7
 8  Property {
 9     ##[ x <- Int ]##
10     $x + $x  >= $x;
11  }, name => "2*x is greater than x";
nereida:~/projects/coro/xpp/check> ./electrotest.pl
1..2
ok 1 - ’sqrt satisfies defn of square root’ (1000 attempts)
not ok 2 - ’2*x is greater than x’ falsified in 3 attempts
# Counterexample:
# $x = -3;
```

Uso de `Test::LectroTest` Dentro de `Test::More`

Para el uso del módulo con `Test::More` utilice el módulo `Test::LectroTest::Compat`. Este módulo permite mezclar la comprobación mediante propiedades con la familia de los módulos `Test::`. Es posible llamar a las funciones `is()` y `ok()` dentro del código de especificación de una propiedad. Véase el uso de `cmp_ok` en la línea 10 del siguiente ejemplo:

```
lhp@nereida:~/Lperl/src/LectroTest/Titi$ cat -n t/01lectro.t
 1  use Titi;  # contains code we want to test
 2  use Test::More tests => 2;
 3  use Test::LectroTest::Compat;
 4
 5  # property specs can now use Test::Builder-based
 6  # tests such as Test::More’s cmp_ok()
 7
 8  my $prop_nonnegative = Property {
 9    ##[ x <- Int, y <- Int ]##
10    cmp_ok(Titi::my_function( $x, $y ), ’>=’, 0);
11  }, name => "my_function output is non-negative" ;
12
13  # and we can now check whether properties hold
14  # as Test::Builder-style tests that integrate
15  # with other T::B tests
16
17  holds( $prop_nonnegative ); # test whether prop holds
18  cmp_ok( 0, ’<’, 1, "trivial 0<1 test" ); # a "normal" test
```

Como se ve en la línea 17 se provee también una función `holds` la cual somete a pruebas la propiedad que recibe como argumento.

La Función `holds`

La función `holds`:

```
holds(property, opts...)

holds( $prop_nonnegative );  # check prop_nonnegative
```

```
        holds( $prop_nonnegative, trials => 100 );

        holds(
           Property {
              ##[ x <- Int ]##
              my_function2($x) < 0;
           }, name => "my_function2 is non-positive"
        );
```

Comprueba el cumplimiento de la propiedad. Cuando se la llama crea un objeto Test::LectroTest::TestRunner al que solicita la comprobación de la propiedad, informando del resultado a Test::Builder el cual a su vez informa a Test::Simple o Test::More (función `plan`). Las opciones que se provean a `holds` serán pasadas al objeto `TestRunner` de manera que se puede cambiar el número de pruebas, etc. (véase la documentación de Test::LectroTest::TestRunner para la lista completa de opciones).

El Módulo a Comprobar

La prueba `t/01lectro.t` la usamos para comprobar el siguiente módulo `Titi.pm`. Esta es la jerarquía de directorios:

```
lhp@nereida:~/Lperl/src/LectroTest$ tree Titi/
Titi/
|-- Changes
|-- MANIFEST
|-- Makefile.PL
|-- README
|-- lib
|   `-- Titi.pm
`-- t
    |-- 01lectro.t
        `-- Titi.t

        2 directories, 7 files
```

En concreto la prueba `t/01lectro.t` vista antes comprueba que la función `my_function` abajo devuelve valores positivos:

```
lhp@nereida:~/Lperl/src/LectroTest/Titi$ cat -n lib/Titi.pm
 1  package Titi;
 2
 3  use 5.008007;
 4  use strict;
 5  use warnings;
 6
 7  require Exporter;
 8
 9  our @ISA = qw(Exporter);
10  our @EXPORT = qw( my_function );
11  our $VERSION = '0.01';
12
13  # Preloaded methods go here.
14  sub my_function {
15    my ($x, $y) = @_;
16    $x*$y*$y
```

```
17  }
18
19  1;
```

Ejecución

Cuando ejecutamos las pruebas obtenemos un contraejemplo a nuestra aserción de que `my_function` devuelve valores positivos:

```
lhp@nereida:~/Lperl/src/LectroTest/Titi$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM"
         "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/01lectro....
t/01lectro....NOK 1#     Failed test (t/01lectro.t at line 17)
#     '-9'
#         >=
#     '0'
#     Counterexample:
#     $x = -1;
#     $y = 3;
# Looks like you failed 1 test of 2.
t/01lectro....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/2 tests, 50.00% okay
t/Titi........ok
Failed Test  Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
t/01lectro.t    1   256     2    1  50.00%  1
Failed 1/2 test scripts, 50.00% okay. 1/3 subtests failed, 66.67% okay.
make: *** [test_dynamic] Error 255
```

Un Algebra de Generadores

El módulo Test::LectroTest::Generator provee generadores para los tipos de datos mas comunes. Provee además un algebra de combinadores de generadores que permite la construcción de generadores complejos.

```
$ perl -wde 0
  DB<1> use Test::LectroTest::Generator qw(:common Gen)
  DB<2> p 1<<31
2147483648
  DB<3> $tg = Int(range=>[0, 2_147_483_647], sized=>0)
  DB<4> x $tg
0  Test::LectroTest::Generator=HASH(0x84faa78)
   'generator' => CODE(0x84fa9a0)
      -> &Test::LectroTest::Generator::\
      __ANON__[/usr/local/share/perl/5.8.7/Test/LectroTest/Generator.pm:212]\
      in /usr/local/share/perl/5.8.7/Test/LectroTest/Generator.pm:210-212
```

Una expresión como `Int(range=>[0, 2_147_483_647], sized=>0)` construye un generador aleatorio de enteros.

Si el argumento `sized` fuera cierto (que es el valor por defecto) los enteros generados lo serían de acuerdo con una cierta 'política de tamaños'. El generador es un objeto que contiene un método `generate`. Aqui el argumento `sized` puesto a falso indica que los valores generados sólo están restringidos por el rango especificado.

```
  DB<3> x $tg->generate
0  1760077938
  DB<4> x scalar localtime $tg->generate
0  'Wed Jan 14 00:07:07 1998'
```

La subrutina `Gen` permite la construcción de nuevos generadores:

```
  DB<7> $ctg = Gen { scalar localtime $tg->generate( @_ ) }
  DB<8> print $ctg->generate()."\n" for 1..3
Thu Aug 15 15:34:05 2019
Sat Sep  7 04:11:15 1974
Sun Jun 19 14:30:14 1977
```

de esta forma hemos construido un generador aleatorio de fechas.

Existen múltiples generadores básicos y un conjunto de combinadores de generadores. Sigue un ejemplo que muestra la influencia del parámetro *sizing guidance*:

```
  DB<1> use Test::LectroTest::Generator qw(:common :combinators)
  DB<2> $int_gen = Int
  DB<3> print $int_gen->generate($_)." " for 1..100 # El arg es sizing guidance
1 1 0 0 -2 3 -3 8 -6 6 6 6 -9 -10 2 1 -9 -11 4 12 19 -11 16 -1 -22 12 -13 -20 -12 9 \
-21 -9 21 -27 5 -19 -20 28 -18 -31 -14 2 -40 -10 33 32 1 33 6 21 3 5 45 31 -28 20 -17 \
-9 58 48 -58 2 -40 27 6 -20 -41 30 59 40 -49 -60 -38 44 -44 -63 12 -76 45 41 65 \
63 -20 59 -5 62 0 65 63 34 71 32 59 -61 -7 -14 30 -71 -13 -58
```

Veamos algunos constructores de generadores básicos:

```
  DB<4> $flt_gen = Float( range=>[0,1] )
  DB<5> x  $flt_gen->generate
0  0.793049252432869
  DB<6> x  $flt_gen->generate
0  0.543474544861482
  DB<7> $bln_gen = Bool
  DB<8> x $bln_gen->generate
0  0
  DB<9> x $bln_gen->generate
0  1
  DB<10> x $bln_gen->generate
0  1
  DB<11> $chr_gen = Char( charset=>"a-z" )
  DB<12> x $chr_gen->generate
0  's'
  DB<13> x $chr_gen->generate
0  'u'
  DB<14> x $chr_gen->generate
0  'i'
  DB<15> $elm_gen = Elements("e1", "e2", "e3", "e4")
  DB<16> x $elm_gen->generate
0  'e2'
  DB<17> x $elm_gen->generate
0  'e4'
  DB<18> x $elm_gen->generate
0  'e2'
```

El combinador `Frequency` permite construir un generador a partir de una lista de generadores que produce los elementos con probabilidades proporcionales a las frecuencias asignadas:

```
  DB<19> $english_dist_vowel_gen = Frequency([8.167,Unit("a")], [12.702,Unit("e")], \
                        [6.996,Unit("i")], [ 7.507,Unit("o")],[2.758,Unit("u")] )
  DB<20> x $english_dist_vowel_gen->generate
0  'i'
  DB<21> x $english_dist_vowel_gen->generate
0  'e'
  DB<22> x $english_dist_vowel_gen->generate
0  'i'
0  'o'
```

Otro combinador es `Paste`:

```
  DB<23> $digit_gen  = Elements( 0..9 )
  DB<24> $ssn_gen = Paste(Paste(($digit_gen)x3),Paste(($digit_gen)x2),Paste(($digit_gen)x4),glue => "-")
  DB<25> x $ssn_gen->generate
0  '168-19-1333'
  DB<26> x $ssn_gen->generate
0  '348-35-8320'
```

Veamos el generador `String`:

```
  DB<27>  $gen = String( length=>[3,5], charset=>"A-Z", size => 100 )
  DB<28> x $gen->generate
0  'KBZNB'
  DB<29> x $gen->generate
0  'AAK'
  DB<30> x $gen->generate
0  'AZL'
```

Un ejemplo con `List`:

```
  DB<31> $ary_gen = List( Int(sized=>0), length => 5 )
  DB<32> x $ary_gen->generate
0  ARRAY(0x8563850)
   0  19089
   1  '-13489'
   2  10390
   3  5382
   4  981
  DB<33> x $ary_gen->generate
0  ARRAY(0x853f030)
   0  17062
   1  18558
   2  29329
   3  31931
   4  2464
```

También podemos construir generadores de hashes:

```
  DB<34> $gen = Hash( String( charset=>"A-Z", length=>3 ), Float( range=>[0.0, 100.0] ), length => 4)
  DB<35> x $gen->generate
0  HASH(0x8576a30)
   'FSW' => 0.998256374071719
   'KLR' => 0.0577333231717212
   'PEV' => 0.834037952293134
   'TNK' => 0.0146371360307889
```

El combinador `OneOf` aplica uno de varios generadores:

```
  DB<36> $gen = OneOf( Unit(0), List(Int,length=>3) )
  DB<37> x $gen->generate
0  ARRAY(0x856c454)
   0  '-1'
   1  0
   2  0
  DB<38> x $gen->generate
0  0
```

El combinador `Each` permite generar listas formados de aplicar cada uno de los generadores:

```
  DB<39> $gen = Each( Char(charset => "aeiou"), Int( range=>[0,10], sized => 0 ) )
  DB<40> x $gen->generate
0  ARRAY(0x857b770)
   0  'o'
   1  3
  DB<41> x $gen->generate
0  ARRAY(0x85771bc)
   0  'e'
   1  5
  DB<42> x $gen->generate
0  ARRAY(0x857b080)
   0  'i'
   1  3
```

El combinador `Apply` aplica el código dado al resultado producido opr los generadores especificados:

```
  DB<42> $gen = Apply( sub { $_[0] x $_[1] }, Char( charset=>'a-z'), Unit(4) )
  DB<43> x $gen->generate
0  'xxxx'
  DB<44> x $gen->generate
0  'hhhh'
  DB<45> x $gen->generate
0  'jjjj'
```

El siguiente ejemplo produce matrices 3x3:

```
  DB<46> $loloi_gen = List( List( Int(sized=>0), length => 3 ), length => 3)
  DB<47> x $loloi_gen->generate
0  ARRAY(0x857bd1c)
   0  ARRAY(0x8576a90)
      0  9648
      1  2796
      2  9589
   1  ARRAY(0x8576dcc)
      0  '-29523'
      1  '-21714'
      2  31931
   2  ARRAY(0x857658c)
      0  '-9477'
      1  '-2434'
      2  '-3794'
```

Ejercicio 5.21.1. *Construya un generador que produzca problemas de la mochila 0-1 representados mediante una estructura de datos de la forma*

```
[$Capacity, [$w_0, ..., $w_n], [$p_0, ..., $p_n]]
```

Usando `Test::LectroTest::Generator` en `Algorithm::Knap01DP`

El módulo `Test::LectroTest::Generator` puede ser usado para generar entradas cuya solución sea conocida y de esta forma comprobar el buen funcionamiento del algoritmo.

En el ejemplo que sigue (fichero `t/03lectro.t`) se generan problemas de 10 objetos con los beneficios (`profits`) iguales a los pesos. Después se genera un conjunto solución (líneas 15-19). Al elegir la capacidad de la mochila igual a la suma de los pesos del subconjunto generado podemos estar seguros que dicho subconjunto es una solución y que el valor óptimo es igual a la capacidad. De hecho este subproblema de la mochila es conocido como *problema del subconjunto suma* (*Subset Sum Problem o SSP*).

```
lhp@nereida:~/Lperl/src/perl_testing_adn_examples/chapter_03/Algorithm-Knap01DP-0.25$ cat -n t/03lectro.t
 1  use strict;
 2  use Test::More;
 3  use Test::LectroTest::Generator qw(:common);
 4  use List::Util qw(sum);
 5
 6  use Algorithm::Knap01DP;
 7
 8  my $t = shift || 100;
 9  plan tests => $t;
10  my ($C, @sol);
11  for (1..$t) {
12    my $weights = List(Int(range => [5,100], sized=>0), length => 10);
13    my @w = @{$weights->generate};
14    my @p = @w;
15    my $set = List(Bool, length => 10);
16    do {
17      @sol = @{$set->generate};
18      $C = sum( @w[ grep { $sol[$_] } 0..9 ] );
19    } while ($C == 0);
20    my $knap = Algorithm::Knap01DP->new( capacity => $C, weights => \@w, profits => \@p);
21    $knap->Knap01DP();
22    is($C, $knap->{tableval}[-1][-1], "Random subset-sum problem");
23  }
```

Observe como el uso de `plan` en la línea 9 nos permite ajustar dinámicamente el número de pruebas a ejecutar. Podemos hacer una ejecución vía `make`:

```
lhp@nereida:~/Lperl/src/perl_testing_adn_examples/chapter_03/Algorithm-Knap01DP-0.25$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib', 'blib/arch')"
t/01alltests....ok
t/02bench.......ok
t/03lectro......ok
All tests successful.
Files=3, Tests=116,  4 wallclock secs ( 4.08 cusr +  0.02 csys =  4.10 CPU)
```

o una ejecución ”manual”:

```
lhp@nereida:~/Lperl/src/perl_testing_adn_examples/chapter_03/Algorithm-Knap01DP-0.25/t$ perl 03lectro.t 4
1..4
ok 1 - Random subset-sum problem
ok 2 - Random subset-sum problem
ok 3 - Random subset-sum problem
ok 4 - Random subset-sum problem
```

Referencias

Para saber mas sobre `Test::LectroTest` lea las traparencias de Tim Moertel en http://community.moertel.com/ thor/talks/

5.21.15. Práctica: Generación de Pruebas con `Test::LectroTest`

Añada pruebas de verificación usando `Test::LectroTest` al módulo construido en la práctica 5.21.13.

Si no realizó la práctica anterior puede partir del módulo `Algorithm::Knap01DP` que puede encontrar en http://search.cpan.org/ casiano/Algorithm-Knap01DP/ .

5.21.16. 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

o en otras palabras ¿Quien prueba las pruebas? ¿Quien controla la calidad de los controladores de calidad? ¿Quién verifica el buen funcionamiento de los verificadores de buen funcionamiento?

5.22. Software de Rastreo de Errores

Todo software contiene errores. Si descubres un error en un código lo mejor es informar al autor. Un Email puede ser suficiente, pero es mejor - si está disponible - usar un *software de rastreo de errores*.

Un software de rastreo de errores es un tipo particular de *software de rastreo de problemas*. También se les denomina *trouble ticket system* y *incident ticket system*. Un software de este tipo suele ser habitualmente una aplicación web de una organización/empresa/comunidad que gestiona y mantiene el estado de progreso de las quejas y problemas de los usuarios: creación de incidencias, actualización, estado de la incidencia: resuelta, rechazada, etc.

Un *ticket* es un fichero dentro del sistema de rastreo de incidencias que contiene información sobre las intervenciones de las partes implicadas en la resolución de la incidencia. Lo hbitual es que los tickets se crean via web o bien via un servicio de asistencia telefónica.

5.22.1. Request Tracker

CPAN utiliza el software de rastreo de errores RT (*Request Tracker*) para el mantenimiento de incidencias en los repositorios CPAN. (Véase http://rt.cpan.org/).

La siguiente cita del libro *RT Essentials* [16] de Dave Rolsky, Darren Chamberlain, Richard Foley, Jesse Vincent y Robert Spier ayuda a entender el concepto:

If your organization is anything like any of ours, there's always a lot of stuff to do. Vendors need to get paid. Customers need to get invoiced. Staff need to do work for customers. Sales inquiries need to be answered. Bugs in hard- or software need to be fixed, and everyone needs to know that they have been fixed. Somebody needs to take out the garbage. And at the end of the day, you've got to know who wanted what, who did it, when it got done, and most importantly what remains undone.

That's where a ticketing system comes in.

The convention is to call each request or piece of work a Ticket. When a new thing comes into the system, we give it a virtual slip of paper with a number, much like the ticket for checking your coat at the coat room or leaving your car in valet parking. This is the ticket we track. Before we get into typical applications, let's dissect a generic ticketing system into its component parts, by building a list of the things a typical ticketing system will do. You can use this list to decide whether or not you need a ticketing system.

These are the bones of a true ticketing system:

- *Register an event or a ticket*
- *Assign an owner, or person responsible, to the ticket*
- *Assign additional interested parties to the ticket*
- *Track changes to the ticket*
- *Inform interested parties of these changes*

Ejercicio 5.22.1. *Visite la página de bugs del módulo CPAN.*

5.23. Patches o Parches

Si se descubre errores en un módulo/distribución y se ha programado una solución al problema lo adecuado es enviar un *parche* o *patch*.

El programa patch creado por Larry Wall[7] en 1985 - permite la actualización de un conjunto de ficheros de texto de acuerdo con las instrucciones de un fichero de texto - denominado *fichero patch* o *parche* - aparte.

5.23.1. Creación de un Parche/Patch

Salvar

Primero copiamos recursivamente el directorio original:

```
pp2@nereida:/tmp/$ cp -Rp Algorithm-Knap01DP-0.01 Algorithm-Knap01DP-0.01.new/
```

Introducir Cambios

Luego hacemos los cambios y mejoras que consideremos necesarios:

```
pp2@nereida:/tmp/Algorithm-Knap01DP-0.01.new/lib/Algorithm$ vi Knap01DP.pm
..........................................................
```

Comprobarlos Cambios

Por supuesto, comprobamos que todo funciona correctamente y que las modificaciones corrigen los errores que hemos detectado:

```
pp2@nereida:/tmp/Algorithm-Knap01DP-0.01.new/lib/Algorithm$ perl -c Knap01DP.pm
Knap01DP.pm syntax OK
pp2@nereida:/tmp/Algorithm-Knap01DP-0.01.new/lib/Algorithm$ cd ../..
pp2@nereida:/tmp/Algorithm-Knap01DP-0.01.new$ make test TEST_VERBOSE=1
Skip blib/lib/Algorithm/newknap.patch (unchanged)
cp lib/Algorithm/Knap01DP.pm blib/lib/Algorithm/Knap01DP.pm
Skip blib/lib/Algorithm/Knap01DP.pm.original (unchanged)
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(1, 'blib/lib', 'blib/arch')"
t/01MartelloAndTothBook....1..12
ok 1 - use Algorithm::Knap01DP;
ok 2 - ReadKnap knap21.dat
ok 3 - Knap01DP knap21.dat
.......................
All tests successful.
Files=1, Tests=12,  0 wallclock secs ( 0.07 cusr +  0.01 csys =  0.08 CPU)
```

---

[7] Si, ese mismo

Crear el Patch

Entonces procedemos a crear el patch usando el programa diff

```
pp2@nereida:/tmp$ diff -ur Algorithm-Knap01DP-0.01 Algorithm-Knap01DP-0.01.new/ > newknap.patch
```

La opción `-u` indica que usaremos formato `unificado` para la representación de las diferencias. Este es el formato mas usado en el intercambio de código entre desarrolladores.

La opción `-r` indica que queremos que se analizen toda la jerarquía de ficehros y directorios y se establezcan las diferencias entre ficheros, si estas existen. El fichero `newknap.patch` contiene una descripción de las diferencias entre la vieja y la nueva versión:

```
pp2@nereida:/tmp$ cat -n newknap.patch
 1  Sólo en Algorithm-Knap01DP-0.01.new/: blib
 2  diff -ur Algorithm-Knap01DP-0.01/lib/Algorithm/Knap01DP.pm Algorithm-Knap01DP-0.01.new/lib/Algorithm/K
 3  --- Algorithm-Knap01DP-0.01/lib/Algorithm/Knap01DP.pm   2005-05-20 07:40:28.000000000 +0100
 4  +++ Algorithm-Knap01DP-0.01.new/lib/Algorithm/Knap01DP.pm       2008-05-16 08:04:12.000000000 +0100
 5  @@ -4,6 +4,7 @@
 6   use warnings;
 7   use Carp;
 8   use IO::File;
 9  +use List::Util qw{first};
10
11   use Exporter;
12   our @ISA = qw(Exporter);
13  @@ -20,6 +21,7 @@
14     my @f;
15
16     croak "Profits and Weights don't have the same size" unless scalar(@w) == scalar(@p);
17  +  croak "Invalid weight/profit $_" if defined($_ = first { $_ <= 0 } (@w, @p));
18
19     for my $c (0..$M) {
20       $f[0][$c] = ($w[0] <= $c)? $p[0] : 0;
21  Sólo en Algorithm-Knap01DP-0.01.new/lib/Algorithm: Knap01DP.pm.original
22  Sólo en Algorithm-Knap01DP-0.01.new/lib/Algorithm: newknap.patch
23  Sólo en Algorithm-Knap01DP-0.01.new/: Makefile
24  Sólo en Algorithm-Knap01DP-0.01.new/: pm_to_blib
25  diff -ur Algorithm-Knap01DP-0.01/t/01MartelloAndTothBook.t Algorithm-Knap01DP-0.01.new/t/01MartelloAnd
26  --- Algorithm-Knap01DP-0.01/t/01MartelloAndTothBook.t   2005-05-20 07:40:04.000000000 +0100
27  +++ Algorithm-Knap01DP-0.01.new/t/01MartelloAndTothBook.t       2008-05-16 07:55:17.000000000 +0100
28  @@ -3,7 +3,7 @@
29
30   #########################
31   use strict;
32  -use Test::More  tests => 11;
33  +use Test::More  tests => 12;
34
35   BEGIN { use_ok('Algorithm::Knap01DP', qw/Knap01DP ReadKnap/); }
36
37  @@ -39,6 +39,11 @@
38   eval { Knap01DP($M, $w, $p) };
39   like $@, qr/Profits and Weights don't have the same size/;
40
41  +# test to check when weights and profits have negative value
42  +$M = 100; @$w = @$p = (1,7,-2,3..8);
43  +eval { Knap01DP($M, $w, $p) };
```

```
+like $@, qr/Invalid weight/;
+
 TODO: { # I plan to provide a function to find the vector solution ...
   local $TODO = "Randomly generated problem";
   can_ok('Algorithm::Knap01DP', 'GenKnap');
```

La cabecera

```
diff -ur Algorithm-Knap01DP-0.01/lib/Algorithm/Knap01DP.pm Algorithm-Knap01DP-0.01.new/lib/Algorithm/K
--- Algorithm-Knap01DP-0.01/lib/Algorithm/Knap01DP.pm   2005-05-20 07:40:28.000000000 +0100
+++ Algorithm-Knap01DP-0.01.new/lib/Algorithm/Knap01DP.pm       2008-05-16 08:04:12.000000000 +0100
```

indica que las diferencias que siguen se refieren a los ficheros `Knap01DP.pm`.

Un rango como `@@ -4,6 +4,7 @@` establece un trozo que ha cambiado. Nos indica que las líneas de la 4 a la 4+6 del original han sido cambiadas por las líneas de la 4 a la 4+7 en el nuevo. La presencia del prefijo `+` indica que se ha insertado una nueva línea:

```
+use List::Util qw{first};
```

Las restantes líneas son 'de contexto'.

Otro trozo que ha cambiado es `@@ -20,6 +21,7 @@`

```
+  croak "Invalid weight/profit $_" if defined($_ = first { $_ <= 0 } (@w, @p));
```

Otro fichero que ha cambiado es `01MartelloAndTothBook.t`:

```
diff -ur Algorithm-Knap01DP-0.01/t/01MartelloAndTothBook.t Algorithm-Knap01DP-0.01.new/t/01MartelloAnd
--- Algorithm-Knap01DP-0.01/t/01MartelloAndTothBook.t   2005-05-20 07:40:04.000000000 +0100
+++ Algorithm-Knap01DP-0.01.new/t/01MartelloAndTothBook.t       2008-05-16 07:55:17.000000000 +0100
```

El trozo:

```
@@ -3,7 +3,7 @@

 #########################
 use strict;
-use Test::More  tests => 11;
+use Test::More  tests => 12;

 BEGIN { use_ok('Algorithm::Knap01DP', qw/Knap01DP ReadKnap/); }
```

indica que se cambió la línea `use Test::More  tests => 11;` por `use Test::More  tests => 12;` Parece que se ha añadido una prueba. En efecto, vemos que se han insertado 5 nuevas líneas que comprueban el buen funcionamiento de la función `Knap01DP` cuando se le pasan pesos/beneficios negativos:

```
@@ -39,6 +39,11 @@
 eval { Knap01DP($M, $w, $p) };
 like $@, qr/Profits and Weights don't have the same size/;

+# test to check when weights and profits have negative value
+$M = 100; @$w = @$p = (1,7,-2,3..8);
+eval { Knap01DP($M, $w, $p) };
+like $@, qr/Invalid weight/;
+
 TODO: { # I plan to provide a function to find the vector solution ...
   local $TODO = "Randomly generated problem";
   can_ok('Algorithm::Knap01DP', 'GenKnap');
```

Enviar los Cambios
El último paso es enviar los cambios al autor del módulo. Via email o http://rt.cpan.org/.

5.23.2. Aplicar el Patch

Una vez el desarrollador de la librería recibe el patch (habitualmente via email tal vez por medio de un software de rastreo de errores) deberá situarse en el directorio de desarrollo y aplicar el parche:

```
$ cd /tmp/Algorithm-Knap01DP-0.01
pp2@nereida:/tmp/Algorithm-Knap01DP-0.01$ patch -p1 < ../newknap.patch
patching file lib/Algorithm/Knap01DP.pm
patching file t/01MartelloAndTothBook.t
```

La opción `-p1` utilizada indica que hay que saltarse/quitar el primer directorio en los caminos que describen los nombres completos de los ficheros. Esto es, un camino como:

```
Algorithm-Knap01DP-0.01/t/01MartelloAndTothBook.t
```

es convertido en:

```
t/01MartelloAndTothBook.t
```

5.24. Escribir Módulos para CPAN

Para saber mas sobre como escribir módulos para CPAN lea el libro [17]. Puede descargarlo desde internet desde http://www.apress.com/resource/freeebook/9781590590188 Véase también la FAQ en CPAN http://www.cpan.org/misc/cpan-faq.html

Capítulo 6

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*: 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.

6.1. 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;
...
```

Paquetes y Clases

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.

El convenio del Subguión para Significar Privacidad

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 ([18] .

El Constructor

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`

```
 1 sub new {
 2   my $class = shift;
 3   my ($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 }
```

La Operación `bless`

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.

La LLamada a un Método de Clase

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`.

La Referencia no es el Objeto

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.

```
 1 sub new {
 2   my $class = shift;
 3   my ($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 }
```

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 6.1 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.

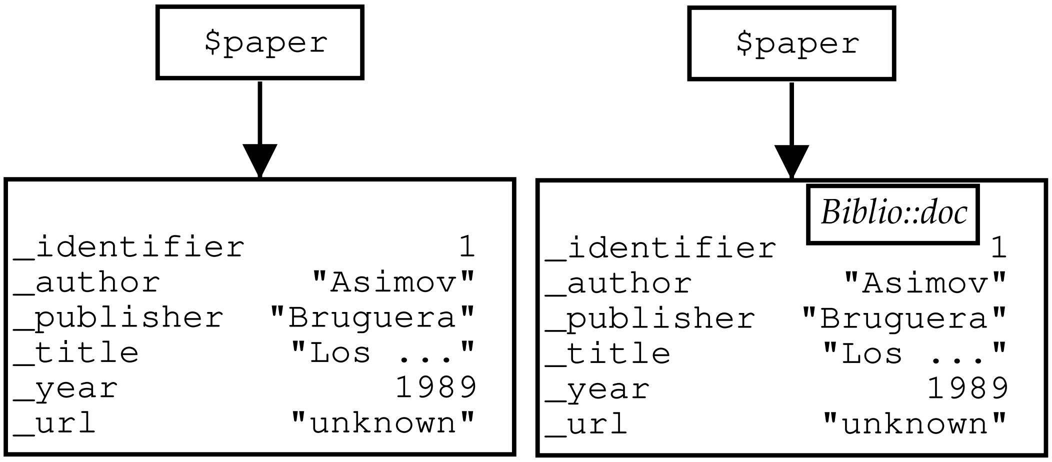

Figura 6.1: Al bendecir un objeto este es marcado con la clase a la que pertenece

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:

```
$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`

La función `reftype` en Scalar::Util permite obtener el tipo básico subyacente al objeto:

```
  DB<1> use Scalar::Util qw{reftype}
  DB<2> $x = bless {}, 'SomeClass'
  DB<3> p ref($x)
SomeClass
  DB<4> p reftype($x)
HASH
```

Métodos y la Sintáxis Flecha

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.

Métodos de Objeto y Métodos de Clase

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*.

Getters, Setters y Mutators

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*.

Comprobación de los Tipos de los Argumentos de un Método

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.

Sintáxis de Flecha y Nombres Completos

Es posible usar la sintáxis de flecha con una especificación de nombre completa.

```
nereida:˜> perl -wde 0
main::(-e:1):   0
  DB<1> package Tutu::Titi; sub new { bless {}, $_[0] }
  DB<2> p __PACKAGE__
main
  DB<3> x X->Tutu::Titi::new
0  X=HASH(0x8444024)
     empty hash
```

La Función `blessed`

La función `blessed` en Scalar::Util tiene la sintáxis:

```
blessed EXPR
```

Si `EXPR` es una referencia que fué bendecida retorna el nombre del corespondiente paquete. En otro caso retorna `undef`.

```
$scalar = "foo";
$class  = blessed $scalar;           # undef

$ref    = [];
$class  = blessed $ref;              # undef

$obj    = bless [], "Foo";
$class  = blessed $obj;              # "Foo"
```

6.1.1. Práctica: Un Módulo OOP Simple

Escriba un programa que haga uso del módulo `Biblio::Doc` presentado en la sección 6.1. Haga uso de `h2xs`.

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?

6.2. Generación Automática de Accesors/Mutators

Uso de `AUTOLOAD` para la Generación de Accesors/Mutators

La escritura de los métodos get/set es monótona y pesada. Si el objeto tiene 20 atributos habrá que escribir un buen número de métodos.

Podemos usar `AUTOLOAD` para que, cada vez que se llama a un método de acceso se instale en la tabla de símbolos. Para ello usaremos un ”typeglob” y una subrutina anónima.

Sigue el código:

```
sub AUTOLOAD {
  no strict "refs";

  my $self = shift;
  if ($AUTOLOAD =˜ /\w+::\w+::get(_.*)/) {
    my $n = $1;
    die "Error in $AUTOLOAD: Illegal attribute\n" unless exists $self->{$n};

    # Declarar el metodo get_*****
    *{$AUTOLOAD} = sub { return $_[0]->{$n}; };
```

```
11
12      return $self->{$n};
13    } elsif ($AUTOLOAD =~ /\w+::\w+::set(_.*)/) {
14      my $n = $1;
15      die "Error in $AUTOLOAD: Illegal attribute\n" unless exists $self->{$n};
16      $self->{$n} = shift;
17
18      # Declarar el metodo set_*****
19      *{$AUTOLOAD} = sub { $_[0]->{$n} = $_[1]; };
20    } else {
21      die "Error: Function $AUTOLOAD not found\n";
22    }
23  }
```

En la línea 5 obtenemos el nombre del atributo y lo guardamos (línea 6) en `$n`. En la línea 7 comprobamos que tal atributo existe. La subrutina anónima que crea `AUTOLOAD` en la línea 10 es una clausura con respecto a `$n`: el valor se recuerda, incluso después que se haya salido del ámbito.

Instalación en la Tabla de Símbolos

Obsérvese la asignación de la línea 10:

```
*{$AUTOLOAD} = sub { return $_[0]->{$n}; };
```

Esto hace que se instale una entrada con el nombre del método deseado en la tabla de símbolos.

Desactivación de `strict`

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.

El Método `DESTROY`

El método `DESTROY` es llamado cada vez que el contador de referencias del objeto alcanza cero. Como hemos escrito un `AUTOLOAD` la llamada automática a `DESTROY` que produce la salida de contexto del objeto provoca una llamada a `AUTOLOAD`. Dado que el `AUTOLOAD` que hemos escrito no contempla este caso deberemos evitar que la llamada `DESTROY` sea captada por `AUTOLOAD`. Para lograrlo basta con definir un método `DESTROY` vacío:

```
sub DESTROY {}
```

Programa Cliente: Ejemplos de LLamadas

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");
```

Ejercicio

Ejercicio 6.2.1. *¿Que pasaría en el ejemplo si la llamada tiene la forma:* `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?

Creando Accesors en Bucle

Una alternativa simple es disponer de una función que reciba el nombre de la clase y la lista de los nombres de atributo y cree los correspondientes métodos de acceso:

```
sub make_accessors { # Install getter-setters
  my $package = caller;

  no strict 'refs';
  for my $sub (@_) {
    *{$package."::$sub"} = sub {
      my $self = shift;

      $self->{$sub} = shift() if @_;
      return $self->{$sub};
    };
  }
}
```

No Re-invente la Rueda

Un buen número de módulos en CPAN proveen mecanismos similares a los explicados en esta sección para la construcción automática de métodos get/set y constructores. Estudie los siguientes módulos:

- Class::Accessor
- Class::Accessor::Assert
- Class::MethodMaker
- Class::Struct
- Class::Std
- Class::Std::Utils
- Class::MakeMethods::Emulator::MethodMaker

El Módulo `Class::Accessor`

El siguiente fragmento de un módulo ilustra el modo de uso de `Class::Accessor`. El módulo provee un servicio de llamadas a procedimientos remotos desde un cliente Perl a un servidor Perl situado en otra máquina. Los resultados de la llamada son objetos de la clase `GRID::Machine::Result`.

La línea 3 (`use base qw(Class::Accessor)`) indica que no sólo se usa el módulo `Class::Accessor` sino que se hereda del mismo. En la línea 5 definimos una lista con los atributos públicos del objeto. En la línea 6 se construyen getter/setters para esos atributos.

```
pp2@nereida:~/LGRID_Machine/examples$ sed -ne ’353,370p’ Machine.pm | cat -n
 1  package GRID::Machine::Result;
 2  use List::Util qw(first);
 3  use base qw(Class::Accessor);
 4
 5  my @legal = qw(type rstdout rstderr results);
 6  GRID::Machine::Result->mk_accessors(@legal);
 7  my %legal = map { $_ => 1 } @legal;
 8
 9  sub new {
10    my $class = shift || die "Error: Provide a class\n";
11    my %args = @_;
12
13    my $a = "";
14    die "Illegal arg  $a\n" if $a = first { !exists $legal{$_} } keys(%args);
15
16    bless \%args, $class;
17  }
```

El constructor del ejemplo comprueba que todos los nombres de los argumentos son legales en la línea 14.

El siguiente código muestra un ejemplo de llamada:

```
$r = GRID::Machine::Result->new(%result);
print $r->rstdout;
```

6.2.1. 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 5.21.

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.

6.3. 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.

Puntos a la Hora de Escribir un Constructor

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. Los constructores suelen tener muchos parámetros. Se debe aplicar la estrategia (sección 1.15.7) de llamada con nombres.
4. Es necesario comprobar que los nombres usados en la llamada son legales y que los tipos de los argumentos se ajustan a los tipos de los atributos.
5. Hay que proveer valores por defecto en la inicialización, de manera que campos no especificados por el usuario resulten inicializados de manera apropiada.
6. 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.

Ejemplo

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-- }
}

```

El Hash `_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’]
                  );
```

Se ha introducido un hash que contiene los valores por defecto y el tipo de acceso permitido (lectura/escritura) para cada clave.

El Método `_standard_keys`

```
  sub _standard_keys {
    return keys %_defaults;
  }
```

Queremos además disponer de un método que nos diga que claves forman el objeto. Esa es la función del método privado `_standard_keys`.

El Método `_default_for`

El método `_default_for` permite acceder al valor por defecto.

```
  sub _default_for {
    my ($self, $attr) = @_;

    return $_defaults{$attr}[0];
  }
```

El Método `_accesible`

```
  sub _accesible {
    my ($self, $attr, $access) = @_;
    return $_defaults{$attr}[1] =˜ /$access/;
  }
```

El método `_accesible` nos dice si una clave esta accesible para el tipo de acceso requerido.

El Método `_init`

El método privado `_init` inicializa el hash referenciado por `$self` según lo indicado en el hash `%args`.

```
  sub _init {
    my ($self, %args) = @_;
    my %inits;
    my ($i, $j);

    for $i ($self->_standard_keys) {
      $j = $i;
      $j =˜ s/_//;
```

```
47       $inits{$i} = $args{$j} || $self->_default_for($i);
48     }
49     %$self = %inits;
50   }
51 }
```

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.

Las Claves Correctas

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).

Habría además que comprobar que todas las claves en `%args` están en el conjunto de claves legales.

El Constructor

En la línea 57 se bendice el objeto aún sin inicializar.

```
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 }
```

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'`.

Formas de LLamada de un Constructor

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'`.

Desacomplando la Inicialización de la Construcción

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`).

El Destructor

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).

```
65 sub DESTROY {
66   my $self = shift;
67
68   $self->_decr_count();
69 }
```

Métodos de Acceso a los Atributos

También hemos extendido la subrutina `AUTOLOAD` para que compruebe (líneas 74 y 82) que el método de acceso requerido está permitido.

```
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     die "Error in $AUTOLOAD: Illegal attribute\n" unless exists $self->{$n};
77
78     # Declarar el metodo get_*****
79     *{$AUTOLOAD} = sub { return $_[0]->{$n}; };
80
81     return $self->{$n};
82   } elsif (($AUTOLOAD =~ /\w+::\w+::set(_.*)/) && ($self->_accesible($1,’write’))) {
83     my $n = $1;
84     die "Error in $AUTOLOAD: Illegal attribute\n" unless exists $self->{$n};
85     $self->{$n} = shift;
86
87     # Declarar el metodo set_*****
88     *{$AUTOLOAD} = sub { $_[0]->{$n} = $_[1]; };
89   } else {
90     die "Error: Function $AUTOLOAD not found\n";
91   }
```

6.4. 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 4.2). 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.

6.4.1. Práctica: Constructores-Copia

Utilizando las ideas esbozadas en el párrafo anterior, reescriba el constructor de la clase `Biblio::Doc->new()` descrita en la sección 6.3 (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();
```

6.5. 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`.

Una alternativa mas breve es usar el módulo base:

```
use base qw(B C);
```

Búsqueda de un Método

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 6.5) 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”.

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 5.6). Es posible añadir métodos o atributos a UNIVERSAL.

El método `isa`

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. Sin embargo la modificación de los arrays `@ISA` en la jerarquía borra las caches:

```
  DB<1> @A::ISA = qw{B}; @B::ISA = qw(C); @C::ISA = ()
  DB<2> $x = bless {}, 'A'
  DB<3> x $x->isa('C')
0  1
  DB<4> @B::ISA = ()
  DB<5> x $x->isa('C')
0  ''
```

El método `can`

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.

Un Ejemplo: Métodos Singleton Simon Cozens muestra en el módulo Class::SingletonMethod y en el libro [19] como diseñar métodos singleton en Perl.

Un *método singleton* es uno que se aplica a un objeto particular y no a toda la clase.

La Idea de los Métodos Singleton

El ejemplo que sigue ilustra la idea de usar *singletons*. Aunque los objetos `$a` y `$b` pertenecen a la misma clase, sólo el objeto `$a` dispone del método `dump`.

Para crear un método singleton usamos el método `singleton_method`. Puesto que queremos que todo objeto pueda crear sus métodos singleton debemos ubicar `singleton_method` en la clase `UNIVERSAL`.

```
hp@nereida:~/Lperl/src/advanced_perl_programming2$ cat -n singleton.pl
#!/usr/local/bin/perl -w
use strict;
use Class::SingletonMethod;

my $a = Some::Class->new(yuyu => 4, chuf => [ 1..5]);
my $b = Some::Class->new(yuyu => 8, chuf => [0..9]);

$a->singleton_method( dump => sub {
  my $self = shift;
  require Data::Dumper;
  no warnings;
  $Data::Dumper::Indent = 0;
  print STDERR Data::Dumper::Dumper($self)."\n"
});

$a->dump; # Prints a representation of the object.
$b->dump; # Can't locate method "dump"

package Some::Class;

sub new {
  my $class = shift;

  bless { @_ }, $class;
}
```

Ejecución de `singleton.pl`

```
lhp@nereida:~/Lperl/src/advanced_perl_programming2$ singleton.pl
$VAR1 = bless( {'chuf' => [1,2,3,4,5],'yuyu' => 4}, '_Singletons::135852824' );
Can't locate object method "dump" via package "Some::Class" at ./singleton.pl line 17.
```

La Implantación de Singletons

El método `singleton_method` crea una clase para el objeto que hace la invocación - si no fué creada anteriormente - y hace que esta nueva clase herede de la clase del objeto. El algoritmo garantiza que objetos distintos tendrán clases distintas.

```
lhp@nereida:~/Lperl/src/advanced_perl_programming2$ sed -ne '9,23p' `perldoc -l Class::SingletonMethod` |
package UNIVERSAL;

no warnings; no strict; # no guarantee

sub singleton_method {
    my ($object, $method, $subref) = @_;

    my $parent_class = ref $object;
    my $new_class = "_Singletons::".(0+$object);
    *{$new_class."::".$method} = $subref;
    if ($new_class ne $parent_class) {
        @{$new_class."::ISA"} = ($parent_class);
        bless $object, $new_class;
    }
}
```

Evaluación de una Referencia en un Contexto Numérico

```
nereida:~/doc/casiano/PLBOOK/PLBOOK> perl -wde 0
main::(-e:1):   0
  DB<1> $x = {}
  DB<2> p $x
HASH(0x8150634)
  DB<3> $y =$x+0
  DB<4> p $y
135595572
  DB<5> printf "%x",$y
8150634
```

6.5.1. 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.

6.5.2. 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 4.8 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 1.15.10. Compruebe el buen funcionamiento del método.

```
#!/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 6.1: Un programa con una herencia complicada

6.5.3. Ejercicio: Búsqueda de Métodos

Puesto que los constructores son métodos, son buscados siguiendo el mecanismo descrito en la sección 6.5. Considere el código en la tabla 6.1. Las clases en el ejemplo heredan según el esquema que se muestra en la figura 6.2. ¿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?

6.5.4. 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);
```

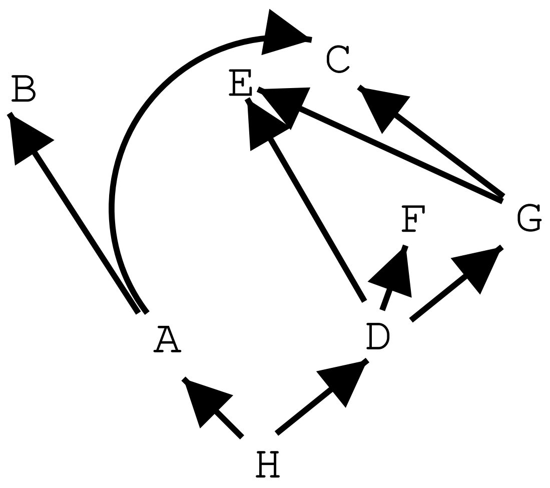

Figura 6.2: Formas de bendición: esquema de herencia del programa

```
  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 6.3 muestra el esquema de herencia y el de delegación (flechas punteadas) usado para el método `_init`.

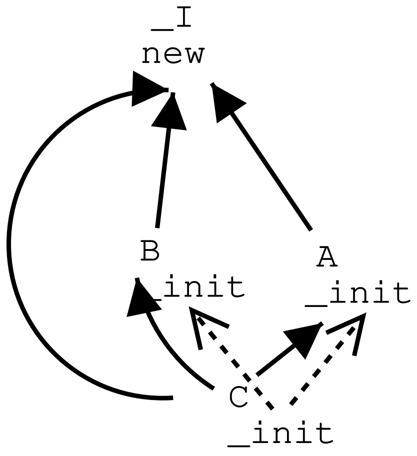

Figura 6.3: Esquema de herencia/delegación del programa

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.

6.5.5. 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 [18] 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);
```

```
  $self->{_c1} = $args{c1};
}
```

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.

6.5.6. 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 6.5.4. 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`.

El prefijo `SUPER::` se refiere a la clase que hace la llamada, como muestra el siguiente ejemplo:

```
main::(-e:1):   0
  DB<1> @A::ISA = qw{B}; @B::ISA = qw{C}; @C::ISA =()
  DB<2> sub C::tutu { 8 }
  DB<3> $x = bless {}, 'A'
  DB<4> x $x->SUPER::tutu
Can't locate object method "tutu" via package "main" at \
     (eval 8)[/usr/share/perl/5.8/perl5db.pl:628] line 2.
  DB<5> x $x->tutu
0  8
  DB<6> package A; $x = bless {}, 'A'; print $x->SUPER::tutu,"\n"
8
```

6.5.7. 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 6.2 cuyo esquema aparece en la figura 6.4, explique como se producirán las llamadas y cual será la salida.

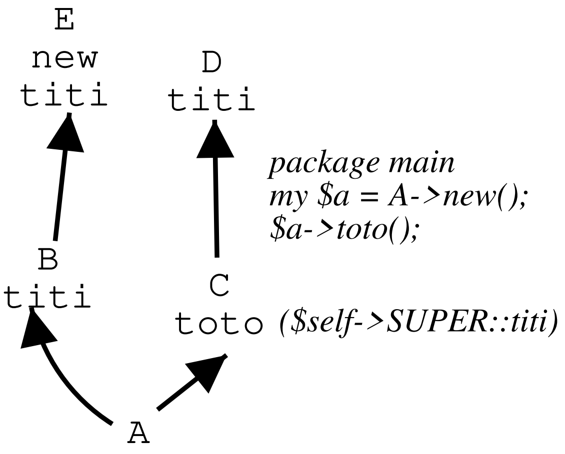

Figura 6.4: Esquema del programa

6.5.8. 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 una declaración 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:

```
#!/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";
}
```

```
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 6.2: Ejemplo de uso de `SUPER`

```
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 1.15.10 para determinar la localización exacta de la llamada.

6.5.9. Práctica: Herencia

Escriba una clase `Biblio::Doc::Article` que herede de la clase `Biblio::Doc` descrita en la sección 6.3. 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 6.3 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:

```
 1 sub _init {
 2   my ($self, %args) = @_;
 3   my %inits;
 4   my ($i, $j);
 5
 6   for $i ($self->_standard_keys) {
 7     $j = $i;
 8     $j =~ s/_//;
 9     $inits{$i} = $args{$j} || $self->_default_for($i);
10   }
11   %$self = %inits;
12 }
```

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.

6.6. 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";
}
$a = "objeto apuntado por \$b";
$b = \$a;
bless $b;

sub DESTROY {
  my $self = shift;
  printf("\n****\n$$self eliminado a las %s\n", scalar localtime);
}
```

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 interfaz 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 6.2:

```
Has llamado a Biblio::Doc::DESTROY() y no existe!
```

Una solución simple para esta situación consiste en definir un método `DESTROY` vacío.

6.7. Instalación Automática de Métodos con `Class::Struct`

El módulo `Class::Struct` forma parte de la distribución núcleo de Perl. El módulo exporta una única función denominada `struct`. La función `struct` recibe como argumentos una lista que describe la estructura de datos asociada con un objeto. El efecto es la generación de un método constructor (que tiene por nombre `new`) y métodos de acceso.

Creación de Métodos

La siguiente sesión con el depurador muestra un ejemplo de uso:

```
lhp@nereida:~/LCLASS_TRAIT/myexample$ perl -MClass::Struct -wde 0
main::(-e:1):   0
  DB<1> struct Foo => { a => '@', s => '$', h => '%', c => 'Tutu' }
  DB<2> struct Tutu => [ nombre => '$', apellidos => '@' ]
```

Estas dos líneas crean los métodos.

Objetos Hash y Objetos Lista

La diferencia entre usar la sintáxis de llaves y la de corchetes estriba en que el objeto construido es un hash en el caso de las llaves y un array en el caso de los corchetes:

```
  DB<32> x Tutu->new(nombre=>'Alberto', apellidos=>[qw(Gonzalez Martinez)])
0  Tutu=ARRAY(0x8704558)
   0  'Alberto'
   1  ARRAY(0x85a3fe0)
      0  'Gonzalez'
      1  'Martinez'
```

Estructura del Constructor

Activando `$Data::Dumper::Deparse` podemos ver la forma del constructor proporcionado:

```
DB<23> use Data::Dumper
DB<24> $Data::Dumper::Deparse = 1
DB<25> p Dumper(\&Foo::new)
$VAR1 = sub {
    package Foo;
    use strict 'refs';
    my($class, %init) = @_;

    $class = 'Foo' unless @_;

    my($r) = {};

    if (defined $init{'c'}) {
        if (ref $init{'c'} eq 'HASH') {
            $$r{'Foo::c'} = 'Tutu'->new(%{$init{'c'};});
        }
        elsif (UNIVERSAL::isa($init{'c'}, 'Tutu')) {
            $$r{'Foo::c'} = $init{'c'};
        }
        else {
            croak('Initializer for c must be hash or Tutu reference');
        }
    }
      croak('Initializer for h must be hash reference')
    if defined $init{'h'} and ref $init{'h'} ne 'HASH';

    $$r{'Foo::h'} = defined $init{'h'} ? $init{'h'} : {};

      croak('Initializer for a must be array reference')
    if defined $init{'a'} and ref $init{'a'} ne 'ARRAY';

    $$r{'Foo::a'} = defined $init{'a'} ? $init{'a'} : [];

    $$r{'Foo::s'} = defined $init{'s'} ? $init{'s'} : undef;

    bless $r, $class;
};
```

Construccion

Ahora podemos construir objetos de esas clases::

```
  DB<4> x $a = Tutu->new(nombre=>'Juan', apellidos=>[qw(Gonzalez Hernandez)])
0  Tutu=ARRAY(0x84eb024)
   0  'Juan'
   1  ARRAY(0x8474e5c)
      0  'Gonzalez'
      1  'Hernandez'
  DB<5> x $b = Foo->new(a=>[1..3], s=>'hola', h=>{x=>1, y=>2}, c=>$a)
0  Foo=HASH(0x850524c)
   'Foo::a' => ARRAY(0x84eb4e0)
      0  1
```

```
      1  2
      2  3
   'Foo::c' => Tutu=ARRAY(0x84eb024)
      0  'Juan'
      1  ARRAY(0x8474e5c)
         0  'Gonzalez'
         1  'Hernandez'
   'Foo::h' => HASH(0x851bed0)
      'x' => 1
      'y' => 2
   'Foo::s' => 'hola'
```

Acceso a Atributos de Tipo Lista

Los siguientes comandos muestran como acceder a los elementos de un atributo de tipo lista:

```
  DB<6> x $b->a
0  ARRAY(0x84eb4e0)
   0  1
   1  2
   2  3
  DB<7> x $b->a(1)
0  2
  DB<8> x $b->a(1,7)
0  7
  DB<9> x $b->a(1)
0  7
  DB<30> $b->a([9..12])
  DB<31> x $b->a
0  ARRAY(0x85399f0)
   0  9
   1  10
   2  11
   3  12
```

El Formato de un Método de Acceso

Mediante `Dumper` podemos ver el codigo del método de acceso. He introducido comentarios que explican las diferentes partes del método::

```
  DB<26> p Dumper(\&Foo::a)
$VAR1 = sub {
    package Foo;
    use strict 'refs';
    my $r = shift @_;

    my $i;
    # Si no quedan args retornamos la ref al array
    @_ ? ($i = shift @_) : return($$r{'Foo::a'});

    # Si $i es ARR y no args modificamos array
    if (ref $i eq 'ARRAY' and not @_) {
        $$r{'Foo::a'} = $i;
        return $r;
    }
```

```
    croak('Too many args to a') if @_ > 1;

    # $i definido y no es ARR y no args. Modificamos elemento
    @_ ? ($$r{'Foo::a'}[$i] = shift @_) : $$r{'Foo::a'}[$i];
};
```

Acceso a Atributos de Tipo Escalar

Mas sencillo aún es acceder a un atributo escalar:

```
  DB<10> x $b->s
0  'hola'
  DB<11> x $b->s('mundo')
0  'mundo'
  DB<12> x $b->s
0  'mundo'
```

Acceso a Atributos de Tipo Hash

Los siguientes comandos muestran accesos al atributo de tipo hash:

```
  DB<13> x $b->h
0  HASH(0x851bed0)
   'x' => 1
   'y' => 2
  DB<14> x $b->h('x')
0  1
  DB<15> x $b->h(x => 3)
0  3
  DB<16> x $b->h('x')
0  3
```

Acceso a Atributos de Tipo Objeto

Los atributos de un objeto pueden ser de tipo objeto:

```
  DB<17> x $b->c
0  Tutu=ARRAY(0x84eb024)
   0  'Juan'
   1  ARRAY(0x8474e5c)
      0  'Gonzalez'
      1  'Hernandez'
  DB<18> x $b->c->nombre
0  'Juan'
  DB<19> x $b->c->nombre('Pedro')
0  'Pedro'
  DB<20>
```

6.8. Sobrecarga de Operadores

Que es la Sobrecarga

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;
```

Categoría	Operadores / Claves
Aritmética	`+ - * / % ** x . neg`
Bit	`<< >> & | ^ ~`
Asignación	`+= -= *= /= %= **= <<= >>= x= .= ++ --`
Comparación	`< <= > >= == != <=> lt le gt ge eq ne cmp`
Funciones	`atan cos sin exp abs log sqrt`
Conversiones	`q("") 0+ bool`
Seudo-operadores	`nomethod fallback =`

Cuadro 6.3: Operadores que pueden ser sobrecargados en Perl. `neg` es la negación unaria

```
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.

El Módulo `overload`

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.

Las Claves

La clave puede ser cualquiera de las reseñadas en la tabla 6.8.

Los Valores

La referencia a la subrutina puede ser

- Una referencia a una subrutina con nombre,

- Una referencia simbólica o
- Una referencia a una subrutina anónima.

¿Cuándo es LLamado el Manejador?

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.

Una Sesión con el Depurador

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
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.
```

Formas de LLamada

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, "")`

Métodos versus Subrutinas

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 6.5).

Los Tres Argumentos del Manejador

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.

Manejadores de Operaciones no Commutativas

Es por esto que, para operaciones no conmutativas como la resta o la divisón, la función de implementación suele comenzar asi:

```
sub bsub {
  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
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 6.5) muestra una forma de hacerlo.

Conflictos en la Sobrecarga

Si ambos operandos son objetos sobrecargados pertenecientes a clases distintas se aplica el método correspondiente al primero:

```
  DB<1> package A; use overload '+' => 'myadd'; sub myadd { 5 }
  DB<2> package B; use overload '+' => 'myadd'; sub myadd { 9 }
  DB<3> $x = bless {}, 'A'
  DB<4> $y = bless {}, 'B'
  DB<5> p $x+$y
5
  DB<6> p $y+$x
9
```

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`).

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.

6.8.1. Búsqueda de la Implementación de un Operador

Búsqueda del Manejador de un Operador Sobrecargado

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 la página 332 puede ser aplicado
3. En otro caso, se mira a ver si el pseudo operador `nomethod` fué definido. Si es así se llama a la subrutina asociada con `nomethod`.
4. En cualquier otro caso generar una excepción

Búsqueda del Manejador con `fallback` Cierto

Este orden puede alterarse si se define el pseudo 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 la página 332 puede ser aplicado
3. En otro caso, se mira a ver si el pseudo 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

Búsqueda del Manejador con `fallback` Falso

Si `fallback` esta definido pero es falso, el mecanismo de propagación de la sobrecarga (anterior 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 pseudo 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" };
```

El cuarto argumento de `nomethod` es el operador ”realmente” solicitado (`$_[3]`)

Argumentos de `nomethod`

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]`)

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.

6.8.2. 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.

Sobrecarga y Contexto de Cadena

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}`)

Sobrecarga y Contexto Numérico

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

A menos que el pseudooperador `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.

Sobrecarga de Operadores Lógicos

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 que usa Set::Scalar 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;
```

El comando `T` nos permite ver la pila de llamadas:

```
  DB<1> T
$ = 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)
```

6.8.3. Sobrecarga de las Constantes

El Problema

El paquete que desarrollaremos en la práctica 6.8.5 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 escribir nuestra práctica de manera que constantes de tipo cadena o numéricas conteniendo fracciones fueran convertidas en objetos de tipo fracción.

La Subrutina `overload::constant`

Para cambiar el modo en que Perl interpreta las constantes enteras, 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 5.7) que como sabemos es ejecutada cada vez que se usa (`use Math::BigFloat`) el módulo. Aquí la función de `import` no es exportar símbolos sino hacer que se ejecute la llamada `overload::constant %_constant_handlers` produciendo la consiguiente modificación de la interpretación de las constantes enteras y flotantes.

Argumentos de `overload::constant`
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/.../.../`).

Los Valores: Argumentos de un Manejador de Constantes
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

Invocación de los manejadores de constantes
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	(’desde’, ’desde’, ’tr’)
	q	(’hacia’, ’hacia’, ’tr’)
qr{qr comi}	qr	(’qr comi’, ’qr comi’, ’qq’)
s/s_pat/s_text/	qr	(’s_pat’, ’s_pat’, ’qq’)
	q	(’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 6.4: Invocación de los manejadores de constantes

Ejemplo: el Módulo `Number::Fraction`

El módulo `Number::Fraction` permite la sobrecarga de constantes. Véase un ejemplo:

```
lhp@nereida:~/Lperl/src$ cat -n fractions.pl
 1  #!/usr/bin/perl
 2  use warnings;
 3  use strict;
 4  use Number::Fraction ':constants';
 5
 6  no warnings 'numeric';
 7  my $trescuartos = '1/2'+'1/4';
 8  print "$trescuartos\n";
 9
10  my $x = '1hola'+2;
11  print "$x\n";
12
13  no Number::Fraction;
14  $trescuartos = '1/2'+'1/4';
15  print "$trescuartos\n";
```

La salida de este programa es:

```
$ ./fractions.pl
3/4
3
2
```

La transformación de las constantes ocurre en tiempo de compilación como muestran las siguientes ejecuciones con B::Terse[1]

```
$ perl -MO=Terse,-exec \
       -MNumber::Fraction=':constants' \
       -e '$x = "1/2"'
OP (0x81ce758) enter
COP (0x824c708) nextstate
SVOP (0x8290460) const [2] RV (0x8298d88) \HASH
PADOP (0x824c308) gvsv  GV (0x814f648) *x
BINOP (0x8235818) sassign
LISTOP (0x8235048) leave [1]
-e syntax OK
```

```
$ perl -MO=Terse,-exec -e '$x = "1/2"'
OP (0x81570b8) enter
COP (0x816c5a0) nextstate
SVOP (0x81571a8) const [2] PV (0x814f5dc) "1/2"
PADOP (0x816c740) gvsv  GV (0x814f660) *x
BINOP (0x816c880) sassign
LISTOP (0x816c708) leave [1]
-e syntax OK
```

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. El constructor `new` será especificado en la sección 6.8.5: Cuando recibe la cadena produce el objeto `Number::Fraction`.

```
 1 package Number::Fraction;
 2
 3 use 5.006;
 4 use strict;
```

---

[1] `B::Terse` provee mecanismos para visualizar el árbol sintáctico y el código máquina generado por el compilador de Perl

```
 5 use warnings;
 6 use Carp;
 7
 8 our $VERSION = sprintf "%d.%02d", ’$Revision: 1.34 $ ’ =~ /(\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 }
```

El `import` proveído hace que, en el caso de que el constructor no devuelva un objeto válido, se devuelva 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.

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.

Seguimos:

```
26 sub unimport {
27   overload::remove_constant(q => undef);
28 }
29
30 sub new {
31  ...
32 }
```

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.

6.8.4. La Sobrecarga y el 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 cualquier mutante. Asi pues, el código:

```
$v1++;
```

Se traduce por:

```
$v1 = $v1->copy(undef,"");
$v1->incr(undef,"");
```

Ejercicio 6.8.1. *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 6.8.3 pagina 336.*

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)) { # Ambos heredan de ref $l
      return (ref $l)->new( # Puede que 'new' hay sido sobreescrito
                          $l->{num} * $r->{den} + $r->{num} * $l->{den},
   $r->{den} * $l->{den}
                         );
    } else {
      croak "Can't add a ", ref $l, " to a ", ref $r;
    }
  } else {
    if ($r =~ /^[-+]?\d+$/) { # $r es un número entero
      return $l + (ref $l)->new($r, 1);
    } else { # r es un double: convertimos
      return $l->to_num + $r;
    }
  }
}
```

6.8.5. Práctica: Números Fraccionarios

Construya un módulo que permita el uso de números fraccionarios como $\frac{4}{5}, \frac{2}{9}, \ldots$, etc. y las operaciones habituales entre los mismos.

El Constructor: Requerimientos de Polimorfismo

Se desea que el contructor sea lo suficientemente polimorfo para admitir las siguientes formas de llamada:

```
lhp@nereida:~/Lperl/src$ perl -MNumber::Fraction -de 0
  DB<1> x Number::Fraction->new(2,4)
0  Number::Fraction=HASH(0x84666a0)
   'den' => 2
   'num' => 1
  DB<2> x Number::Fraction->new(2) # Un número. El denominador es 1
0  Number::Fraction=HASH(0x8527660)
   'den' => 1
   'num' => 2
  DB<3> x Number::Fraction->new(2,0) # Error. El denominador es cero.
Can't make a Number::Fraction with demominator 0
  DB<4> $b = Number::Fraction->new # Sin número: es 0/1
  DB<5> x $b->new # LLamado por un objeto. Se copia el objeto
0  Number::Fraction=HASH(0x85281f4)
   'den' => 1
   'num' => 0
  DB<6> $b = Number::Fraction->new(2,5) # Dos números: numerador y denominador
  DB<7> x Number::Fraction->new($b) # Un argumento objeto: se copia
0  Number::Fraction=HASH(0x8528410)
   'den' => 5
   'num' => 2
```

El Constructor: Solución

Para simplificar la práctica aquí esta un constructor que cumple los requerimientos:

```
lhp@nereida:~/Lperl/src$ sed -ne '31,74p' Number/Fraction.pm | cat -n
 1  {
 2    my $isint = qr{^\s*-?\d+\s*$};
 3    my $isfrac =  qr{^\s*(-?\d+)(/(-?\d+))?$};
 4
 5    sub new {
 6      my $class = shift;
 7
 8      my $self;
 9      if (@_ >= 2) {
10        return unless $_[0] =~ $isint and $_[1] =~ $isint;
11
12        $self->{num} = $_[0];
13        unless ($_[1]) {
14          carp "Can't make a $class with demominator 0";
15          return;
16        }
17        $self->{den} = $_[1];
18      }
19      elsif (@_ == 1) {
20        if (ref $_[0]) { # Number::Fraction->new($x) y $x objeto. Lo copiamos
21          return $class->new($_[0]->{num}, $_[0]->{den}) if (UNIVERSAL::isa($_[0], __PACKAGE__));
22          croak "Can't make a $class from a ", ref $_[0];
23        } else { # Es una cadena 'num/num' o 'num'
24          return unless $_[0] =~ $isfrac;
25
26          $self->{num} = $1;
27          $self->{den} = $3 || 1;
28        }
29      }
30      elsif (ref($class)) { # LLamado $x->new. Copiamos el objeto
31        %$self = %$class;
32      }
33      else {
34        $self->{num} = 0;
35        $self->{den} = 1;
36      }
37
38      bless $self, ref($class) || $class;
39
40      $self->normalise;
41
42      return $self;
43    }
44  }
```

La Función de Normalización de una Fracción

A continuación sigue el código de `normalise`. La función "normaliza" un quebrado, esto es lo reduce a una fracción irreducible.

```
sub normalise {
  my $self = shift;

  # Calculamos el máximo común divisor
```

```
  my $hcf = _hcf($self->{num}, $self->{den});

  # ... y dividmos ambos por el hcf
  $self->{num} /= $hcf;
  $self->{den} /= $hcf;

  # Denominador positivo en forma normalizada
  if ($self->{den} < 0) {
    $self->{num} *= -1;
    $self->{den} *= -1;
  }
}
```

Cálculo del Máximo Común Divisor

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;
}
```

La subrutina `_hcf` es ”puro cálculo” por lo que la implementación Perl es obviamente inferior en rendimiento a la correspondiente versión C:

```
$ cat -n hcf.c
 1  int hcf(int x, int y) {
 2    int t;
 3
 4    if (y > x) {
 5      t = x; x = y; y = t;
 6    }
 7    if (x == y) return x;
 8    while (y) {
 9      t = x;
10      x = y;
11      y = t % y;
12    }
13    return x;
14  }
```

El Módulo `P5NCI`

Para poder llamar a la versión C de la subrutina desde Perl usando el módulo P5NCI lo único que necesitamos es crear una librería dinámica:

```
$ cc  -shared  hcf.o -o libhcf.so
$ ls -ltr | tail -1
-rwxr-xr-x  1 lhp lhp 5446 2007-05-29 16:52 libhcf.so
$ nm libhcf.so                  # nm nos lista los símbolos en la librería
00001650 A __bss_start
```

```
000003c0 t call_gmon_start
00001650 b completed.4463
00001558 d __CTOR_END__
00001554 d __CTOR_LIST__
         w __cxa_finalize@@GLIBC_2.1.3
000004f0 t __do_global_ctors_aux
000003f0 t __do_global_dtors_aux
00001648 d __dso_handle
00001560 d __DTOR_END__
0000155c d __DTOR_LIST__
00001568 a _DYNAMIC
00001650 A _edata
00001654 A _end
00000534 T _fini
00000450 t frame_dummy
00000550 r __FRAME_END__
00001634 a _GLOBAL_OFFSET_TABLE_
         w __gmon_start__
0000048c T hcf                    # Nuestra función máximo común divisor
00000485 t __i686.get_pc_thunk.bx
0000036c T _init
00001564 d __JCR_END__
00001564 d __JCR_LIST__
         w _Jv_RegisterClasses
0000164c d p.4462
```

LLamada desde Perl

El módulo `P5NCI` permite cargar la librería desde Perl y llamar a las funciones:

```
$ cat -n usehcf.pl
 1  #!/usr/local/bin/perl -w
 2  use strict;
 3  use P5NCI::Library;
 4
 5  my $lib = P5NCI::Library->new( library => './libhcf.so' );
 6  $lib->install_function( 'hcf', 'iii' );
 7
 8  print hcf( 20, 10 ), "\n";
 9  print hcf( 12, 9 ), "\n";
10  print hcf( 18, 12 ), "\n";
```

La llamada a `P5NCI::Library->new` carga la librería. La subsiguiente llamada al método `install` busca la función en la librería y crea una interfaz Perl para la especificación dada por la firma `'iii'`. Esta firma indica que la función recibe dos enteros y devuelve un entero. Observe que mediante `P5NCI` es posible usar funciones cuyo fuente no esta disponible. Tampoco importa en que lenguaje esté escrito. Importa que dispongamos de la librería y que conozcamos el nombre y la interfaz de la función.

El constructor admite la opción `path` que permite especificar el path de búsqueda para la librería. En este caso podemos usar el "nombre" oficial de la librería (`hcf`) y no la especifciación completa. También dispone de una opción `package` que permite especificar el paquete en el que se aloja la función.

```
$ cat -n usehcf2.pl
 1  #!/usr/local/bin/perl -w
 2  use strict;
```

```
use P5NCI::Library;

my $lib = P5NCI::Library->new( library => 'hcf', path => '.' );
print "Path de búsqueda:\n@DynaLoader::dl_library_path\n";
$lib->install_function( 'hcf', 'iii' );

print "hcf( 20, 10 ) = ",hcf( 20, 10 ), "\n";
print "hcf( 12,  9 ) = ",hcf( 12, 9 ), "\n";
print "hcf( 18, 12 ) = ",hcf( 18, 12 ), "\n";
```

La variable `DynaLoader::dl_library_path` contiene el camino de búsqueda de las librerías dinámicas.

Ejecución

Ahora al ejecutar el programa obtenemos el máximo común divisor para cada una de las tres parejas:

```
$ usehcf2.pl
Path de búsqueda:
. /usr/local/lib /lib /usr/lib
hcf( 20, 10 ) = 10
hcf( 12,  9 ) = 3
hcf( 18, 12 ) = 6
```

La portabilidad de `P5NCI`

La portabilidad de `P5NCI` es todavía insuficiente. El porcentaje de plataformas en las que funciona es bajo aún. Existen otros mecanismos para empotrar otros lenguajes en Perl: XS, Inline, etc. que son mas robustos.

El Módulo `Inline`

El módulo Inline proporciona los medios para realizar las pasarelas entre el lenguaje Perl y otro lenguaje de programación. En particular Inline::C facilita la integración entre código C y código Perl. Veamos un ejemplo:

```
$ cat -n hcfinline.pl
#!/usr/local/bin/perl -w
use strict;
use Inline 'C';

print hcf( 20, 10 ), "\n";
print hcf( 12, 9 ), "\n";
print hcf( 18, 12 ), "\n";

__END__
__C__

int hcf(int x, int y) {
  int t;

  if (y > x) {
    t = x; x = y; y = t;
  }
  if (x == y) return x;
  while (y) {
```

```
20      t = x;
21      x = y;
22      y = t % y;
23    }
24    return x;
25  }
```

El código C puede almacenarse dentro del fichero `DATA` (esto es, a partir de la aparición del marcador `__END__` en una sección determinada por el marcador `__C__`. También puede guardarse en una cadena ordinaria que es pasada a Inline::C:

```
$ cat -n shcfinline.pl
 1  #!/usr/local/bin/perl -w
 2  use strict;
 3  use Inline 'C' => q{
 4    int hcf(int x, int y) {
 5      int t;
 6
 7      if (y > x) {
 8        t = x; x = y; y = t;
 9      }
10      if (x == y) return x;
11      while (y) {
12        t = x;
13        x = y;
14        y = t % y;
15      }
16      return x;
17    }
18  };
19
20  print hcf( 20, 10 ), "\n";
21  print hcf( 12, 9 ), "\n";
22  print hcf( 18, 12 ), "\n";
```

Esta segunda forma es preferible a la hora de realizar la práctica.

Ejecutémos dos veces el ejemplo y cronometremos los tiempos de ejecución:

```
$ time shcfinline.pl
10
3
6

real    0m1.797s
user    0m1.520s
sys     0m0.250s
$ time shcfinline.pl
10
3
6

real    0m0.065s
user    0m0.060s
sys     0m0.010s
```

Vemos que la segunda vez tarda menos que la primera. Esto es así porque la primera vez `Inline` genera una librería dinámica que - por defecto - se guarda en el subdirectorio `__Inline`:

```
$ ls -ltr | tail -1
drwxr-xr-x  4 lhp lhp 4096 2007-05-31 07:42 _Inline
$ tree _Inline/
_Inline/
|-- build
|-- config
`-- lib
    `-- auto
        `-- shcfinline_pl_677d
            |-- shcfinline_pl_677d.bs
            |-- shcfinline_pl_677d.inl
            `-- shcfinline_pl_677d.so

4 directories, 4 files
```

Además genera el código de traducción de las llamadas desde Perl a las funciones C.

Durante las ejecuciones posteriores `Inline` comprueba que el código fuente no ha cambiado. Si es así carga la librería dinámica generada.

Cabeceras Reconocidas por `Inline::C`

La gramática de `Inline` para `C` reconoce ciertas definiciones de funciones del código `C`. Esto es, `Inline` generará el código necesario para enlazar la llamada a la función como si fuera una subrutina Perl. Si no puede reconocer la definición, esta será ignorada sin que hayan mensajes de error o advertencia. No quedará disponible en el ámbito de Perl, pero podrá aún seguir siendo usada en el ámbito de C. `Inline` busca por definiciones de funciones de estilo:

```
tipo_de_retorno nombre_de_funcion ( pares_tipo_identificador ) { cuerpo }
```

en las parejas `tipo identificador, ...` puede usarse cualquier tipo que esté definido en el fichero `typemap`.

Las siguientes definiciones no serán reconocidas:

```
Foo(int i) # no hay tipo de retorno
int foo(float f) { # no existe definición en typemap para float
int Foo(num) double num; { # la vieja sintáxis C no se soporta
void Foo(void) { # void se permite solo para el retorno
```

`Inline` y Otros Lenguajes

Inline no está limitado a `C` y puede ser usado con otros lenguajes. El siguiente ejemplo muestra como usar Inline::Python para usar código `Python` desde `C`:

```
lhp@nereida:~/Lperl/src/testing$ cat -n python.pl
 1  #!/usr/bin/perl -w
 2  use strict;
 3
 4  use Inline Python => <<'END_OF_PYTHON_CODE';
 5  def add(x,y):
 6    return x + y
 7
 8  def subtract(x,y):
 9    return x - y
10
```

```
END_OF_PYTHON_CODE

print "9 + 16 = ", add(9, 16), "\n";
print "9 - 16 = ", subtract(9, 16), "\n";
```

Al ejecutar este programa obtenemos la siguiente salida:

```
lhp@nereida:~/Lperl/src/testing$ python.pl
9 + 16 = 25
9 - 16 = -7
```

Reescriba en C las Funciones de Cálculo

Cuando escriba esta práctica haga uso de `P5NCI` o de `Inline` para la construcción de las funciones numéricas con prototipos simples que aparezcan durante la construcción del módulo. Basta con que considere la función del cálculo del máximo común divisor. Si se siente realmente fuerte y capacitado puede leer la sección como retornar múltiples valores del Inline::C Cookbok para ver como implantar la suma y el producto de fracciones.

Este es un ejemplo de como hacerlo: Se deben empujar - usando `Inline_Stack_Push` - en la pila del intérprete Perl los valores de retorno:

```
void add_c(int n1, int d1, int n2, int d2) {
  n1 = n1*d2 + n2*d1;
  d1 = d1*d2;
  Inline_Stack_Vars;
  Inline_Stack_Reset;
  Inline_Stack_Push(sv_2mortal(newSViv(n1)));
  Inline_Stack_Push(sv_2mortal(newSViv(d1)));
}
```

La función `Inline_Stack_Vars` define un conjunto de variables internas que son necesarias para poder acceder a la pila del intérprete Perl.

En general, cuando la función sea de tipo `void` como es el caso del ejemplo, se deben usar las macros con nombre `Inline_Stack_`. También si la función tiene un número variable de argumentos (uso de la elípsis `...`).

Antes de comenzar a empujar items en la pila se debe llamar a la función `Inline_Stack_Reset`. Inicializa el puntero de pila interno al comienzo de la pila.

A partir del númerador `n1 = n1*d2 + n2*d1` debemos construir el *escalar* Perl que guarda el número. La función `newSViv` es la que lo permite. La notación `newSViv` significa *new Scalar Value from an Integer Value*. El valor retornado por esta función es la estructura de datos que representa un valor escalar Perl, conocido en la jerga como `SV` (véase `perlguts`). Existe una familia de constructores de valores escalares `SV` a partir de diferentes tipos de fuentes: `newSVuv` para enteros sin signo, `newSVnv` para `doubles`, etc. (véase `perlapi`).

Como sabemos el recolector de basura de Perl usa un contador de referencias para saber que valores están en desuso y liberar la memoria correspondiente. La función `sv_2mortal` marca el `SV` recién creado como mortal para que su memoria pueda ser liberada por el sistema de gestión de memoria cuando sea seguro hacerlo.

Pruebas: Use `Test::LectroTest`

Use Test::LectroTest en la generación de las pruebas. Construya un generador de fracciones usando Test::LectroTest::Generator.

6.9. ¿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;
```

6.9.1. Volcado automático de una variable

Como ejemplo, supongamos que queremos depurar una variable de paquete. Podemos modificar la interfaz 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".

6.9.2. 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 5.7 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.
```

6.9.3. Práctica: Tie Escalar

Escriba una clase que permita conocer el número de asignaciones realizadas a una variable dada.

Capítulo 7

SQLite

Este capítulo está basado en el tutorial de Mike Chirico [20].

7.1. Introducción

Para crear una base de datos ejecute el comando `sqlite3`:

```
$ sqlite3 test.db
SQLite version 3.3.8
Enter ".help" for instructions
sqlite> create table t1 (t1key INTEGER PRIMARY KEY,data TEXT,num double,timeEnter DATE);
sqlite> insert into t1 (data,num) values (’This is sample data’,3);
sqlite> insert into t1 (data,num) values (’More sample data’,6);
sqlite> insert into t1 (data,num) values (’And a little more’,9);
sqlite> .q
lhp@nereida:~/Lperl/src/SQLITE/examples$ ls -l
total 4
-rw-r--r--  1 lhp lhp 2048 2007-08-17 11:02 test.db
```

No es necesario entrar en modo interactivo:

```
lhp@nereida:~/Lperl/src/SQLITE/examples$ sqlite3 test.db  "insert into t1 (data,num) values (’Another item
```

Un `select` retorna los valores en la tabla:

```
lhp@nereida:~/Lperl/src/SQLITE/examples$ sqlite3 test.db
SQLite version 3.3.8
Enter ".help" for instructions
sqlite> select * from t1 limit 2;
1|This is sample data|3.0|
2|More sample data|6.0|
sqlite> select * from t1 order by t1key limit 2 offset 2;
3|And a little more|9.0|
4|Another item|12.0|
```

El comando `.table` permite conocer los nombres de las tablas en la base de datos. Se puede obtener una información mas completa haciendo consultas a la tabla predefinida `sql_master`:

```
sqlite> .table
t1
sqlite> select * from sqlite_master;
table|t1|t1|2|CREATE TABLE t1 (t1key INTEGER PRIMARY KEY,data TEXT,num double,timeEnter DATE)
```

El comando `.dump` permite volcar información sobre los datos insertados en la base de datos:

```
sqlite> .dump
BEGIN TRANSACTION;
CREATE TABLE t1 (t1key INTEGER PRIMARY KEY,data TEXT,num double,timeEnter DATE);
INSERT INTO "t1" VALUES(1, 'This is sample data', 3.0, NULL);
INSERT INTO "t1" VALUES(2, 'More sample data', 6.0, NULL);
INSERT INTO "t1" VALUES(3, 'And a little more', 9.0, NULL);
INSERT INTO "t1" VALUES(4, 'Another item', 12.0, NULL);
COMMIT;
```

La salida del comando `.dump` puede ser filtrada:

```
lhp@nereida:~/Lperl/src/SQLITE/examples$ sqlite3 test.db ".dump" | sed -e 's/\<t1\(key\)\?\>/tabla\1/g'
BEGIN TRANSACTION;
CREATE TABLE tabla (tablakey INTEGER PRIMARY KEY,data TEXT,num double,timeEnter DATE);
INSERT INTO "tabla" VALUES(1, 'This is sample data', 3.0, NULL);
INSERT INTO "tabla" VALUES(2, 'More sample data', 6.0, NULL);
INSERT INTO "tabla" VALUES(3, 'And a little more', 9.0, NULL);
INSERT INTO "tabla" VALUES(4, 'Another item', 12.0, NULL);
COMMIT;
```

El siguiente comando muestra como usar un filtro para obtener una modificación de la base de datos inicial:

```
$ sqlite3 test.db ".dump" | sed -e 's/\<t1\(key\)\?\>/tabla\1/g' | sqlite3 test2.db
lhp@nereida:~/Lperl/src/SQLITE/examples$ ls -ltr
total 8
-rw-r--r--  1 lhp lhp 2048 2007-08-17 11:05 test.db
-rw-r--r--  1 lhp lhp 2048 2007-08-17 11:38 test2.db
```

7.2. Triggers

En el ejemplo anterior la clave `timeEnter` no tiene valores por defecto. Para que se incialice automáticamente es necesario installar un `trigger`. El ejemplo muestra como escribir un trigger:

```
lhp@nereida:~/Lperl/src/SQLITE/examples$ cat trigger1
CREATE TRIGGER insert_t1_timeEnter AFTER  INSERT ON t1
BEGIN
UPDATE t1 SET timeEnter = DATETIME('NOW')  WHERE rowid = new.rowid;
END;
```

La llamada `DATETIME('NOW')` retorna el tiempo UTC mientras que `datetime('now','localtime')` devuelve la hora actual:

```
sqlite> select datetime('now');
2007-08-17 11:27:21
sqlite> select datetime('now','localtime');
2007-08-17 12:27:33
```

Para ejecutarlo hacemos:

```
hp@nereida:~/Lperl/src/SQLITE/examples$ sqlite3 test.db < trigger1
```

Si ahora introducimos un nuevo registro vemos que tiene inicializado su campo `timeEnter`:

```
lhp@nereida:~/Lperl/src/SQLITE/examples$ sqlite3 test.db
sqlite> insert into t1 (data,num) values ('First entry with timeEnter',19);
sqlite> select * from t1;
```

```
1|This is sample data|3.0|
2|More sample data|6.0|
3|And a little more|9.0|
4|Another item|12.0|
5|First entry with timeEnter|19.0|2007-08-17 10:53:09
```

7.3. Logging

```
lhp@nereida:~/Lperl/src/SQLITE/examples$ cat examScript
-- *****************************************************************
--  examScript: Script for creating exam table
--   Usage:
--       $ sqlite3 examdatabase < examScript
--
--   Note: The trigger insert_exam_timeEnter
--          updates timeEnter in exam
-- *****************************************************************
-- *****************************************************************
CREATE TABLE exam (ekey      INTEGER PRIMARY KEY,
                   fn        VARCHAR(15),
                   ln        VARCHAR(30),
                   exam      INTEGER,
                   score     DOUBLE,
                   timeEnter DATE);

CREATE TRIGGER insert_exam_timeEnter AFTER  INSERT ON exam
BEGIN

UPDATE exam SET timeEnter = DATETIME('NOW')
         WHERE rowid = new.rowid;
END;
-- *****************************************************************
-- *****************************************************************

$ sqlite3 examdatabase < examScript
$ sqlite3 examdatabase "insert into exam (ln,fn,exam,score) values ('Anderson','Bob',1,75)"
$ sqlite3 examdatabase "select * from exam"
1|Bob|Anderson|1|75.0|2007-08-17 11:37:59

-- *****************************************************************
--  examLog: Script for creating log table and related triggers
--   Usage:
--       $ sqlite3 examdatabase < examLOG
--
--
-- *****************************************************************
-- *****************************************************************
CREATE TABLE examlog (lkey INTEGER PRIMARY KEY,
                  ekey INTEGER,
                  ekeyOLD INTEGER,
                  fnNEW   VARCHAR(15),
                  fnOLD   VARCHAR(15),
                  lnNEW   VARCHAR(30),
                  lnOLD   VARCHAR(30),
```

```
                  examNEW INTEGER,
                  examOLD INTEGER,
                  scoreNEW DOUBLE,
                  scoreOLD DOUBLE,
                  sqlAction VARCHAR(15),
                  examtimeEnter    DATE,
                  examtimeUpdate   DATE,
                  timeEnter        DATE);

--  Create an update trigger
CREATE TRIGGER update_examlog AFTER UPDATE  ON exam
BEGIN

  INSERT INTO examlog  (ekey,ekeyOLD,fnOLD,fnNEW,lnOLD,
                        lnNEW,examOLD,examNEW,scoreOLD,
                        scoreNEW,sqlAction,examtimeEnter,
                        examtimeUpdate,timeEnter)

          values (new.ekey,old.ekey,old.fn,new.fn,old.ln,
                  new.ln,old.exam, new.exam,old.score,
                  new.score, 'UPDATE',old.timeEnter,
                  DATETIME('NOW'),DATETIME('NOW') );

END;
--
--  Also create an insert trigger
--    NOTE  AFTER keyword ------v
CREATE TRIGGER insert_examlog AFTER INSERT ON exam
BEGIN
INSERT INTO examlog  (ekey,fnNEW,lnNEW,examNEW,scoreNEW,
                      sqlAction,examtimeEnter,timeEnter)

          values (new.ekey,new.fn,new.ln,new.exam,new.score,
                  'INSERT',new.timeEnter,DATETIME('NOW') );

END;

--  Also create a DELETE trigger
CREATE TRIGGER delete_examlog DELETE ON exam
BEGIN

INSERT INTO examlog  (ekey,fnOLD,lnNEW,examOLD,scoreOLD,
                      sqlAction,timeEnter)

          values (old.ekey,old.fn,old.ln,old.exam,old.score,
                  'DELETE',DATETIME('NOW') );

END;
-- *******************************************************************
-- *******************************************************************

$ sqlite3 examdatabase < examLOG
$
$ sqlite3 examdatabase "insert into exam (ln,fn,exam,score) values  ('Anderson','Bob',2,80)"
```

```
$ sqlite3 examdatabase "update exam set score=82 where ln='Anderson' and fn='Bob' and exam=2"
$ sqlite3 examdatabase "select * from examlog"
1|2||Bob||Anderson||2||80.0||INSERT|||2007-08-17 11:44:32
2|2|2|Bob|Bob|Anderson|Anderson|2|2|80.0|80.0|UPDATE||2007-08-17 11:44:32|2007-08-17 11:44:32
3|2|2|Bob|Bob|Anderson|Anderson|2|2|82.0|80.0|UPDATE|2007-08-17 11:44:32|2007-08-17 11:45:07|2007-08-17 11
```

Capítulo 8

DBI

Capítulo 9

Objetos y Bases de Datos

En este capítulo se estudian un conjunto de técnicas para el alamacenamiento y la recuperación de datos complejos.

9.1. Ficheros de Texto como Listas

Una aproximación usual al tratamiento de un fichero (posiblemente de texto) en Perl consiste en cargar todo el fichero en memoria en un array para a continuación manipular el array. Esta aproximación no es adecuada si el fichero es de gran tamaño.

El módulo Perl `Tie::File` permite atar un fichero a un array de manera que las operaciones de lectura/escritura sobre el array se reflejan en el fichero:

```
$ cat -n tiearray.pl
#!/usr/local/bin/perl -w
use strict;
use Tie::File;

my @array;

tie @array, 'Tie::File', 'introcomp.tex.old' or die "$!";

print "Línea 43: $array[42]\n"; # El array comienza en cero ...

my $n_recs = @array;

print "Número de líneas: $n_recs\n";

for (@array) {
 s/\\section/\\SECTION/g;
}
print "Línea 1: $array[0]\n";

push @array, "Penúltima\n", "Última\n";
print "Última línea: $array[-1]\n";

for (@array) {
 s/\\SECTION/\\section/g;
}
print "Línea 1: $array[0]\n";

untie @array;
```

El fichero completo no es cargado en memoria. La primera vez que se itera sobre el fichero se construye un índice conteniendo las desviaciones de cada uno de los registros del fichero. De esta manera los accesos subsiguientes pueden obtenerse de manera directa mediante un `seek()`. Además los registros accedidos se guardan en memoria. El tamaño de la cache puede ajustarse mediante el argumento `memory` del comando `tie`.

Al ejecutar este programa obtenemos una saldia similar a esta:

```
~/Lperl/src$ ./tiearray.pl
Línea 43: Looks good
Número de líneas: 2402
Línea 1: \SECTION{Las Bases}
Última línea: Última
Línea 1: \section{Las Bases}
```

Desafortunadamente, los cambios en el array (en particular inserciones y supresiones en medio del mismo) son costosos.

9.2. Hashes DBM

Cuando Perl se instala en un sistema proporciona una pasarela a las librerías C que dan soporte a DBM (y a la librería SDBM que se encuentra en la distribución de Perl).

Las DBM tienen una limitación: sólo se pueden almacenar pares clave-valor escalares. No es posible almacenar estructuras de datos complejas.

La función `dbmopen`

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

```
lhp@nereida:~/Lperl/src$ cat -n dbm.pl
#!/usr/bin/perl -w
use strict;
my %DATA;

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:

```
lhp@nereida:~/Lperl/src$ cat -n dbm2.pl
#!/usr/bin/perl -w
use strict;
my %G;
dbmopen(%G, "my_gramma", 0644) or die "can not create my_gramma $!";

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

print "TOKENS: ";
{
  local $" = ',';
  print "@tokens\n";
}

my $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
```

El Módulo `DB_File`

Una alternativa a `dbmopen` es usar el módulo `DB_File`. El siguiente ejemplo muestra como usarlo:

```
$ cat -n testdbfile.pl
#!/usr/local/bin/perl -w
use strict;
use DB_File;
my $file = shift || 'test';
my %h;
tie (%h, 'DB_File', $file) or die $!;

$h{abc} = "ABC"; $h{def} = "DEF"; $h{ghi} = "GHI";
untie (%h);

my %g;
tie (%g, 'DB_File', $file) or die $!;

print "$_ => $g{$_}\n" for keys %g;

untie(%g);
```

al ejecutar se obtiene:

```
$ testdbfile.pl
abc => ABC
ghi => GHI
def => DEF
```

9.3. DBMs Multinivel

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.

Filtros para DBM

Es posible añadir un filtro que procese los accesos a un DBM. Podemos definir una subrutina que es llamada cada vez que almacenamos un valor en el hash DBM y otra que sea llamada cada vez que se lee una entrada del hash DBM.

El formato general de uso es:

```
$db = tie %hash, 'DBM', ...

$old_filter = $db->filter_store_key  ( sub { ... } );
$old_filter = $db->filter_store_value( sub { ... } );
$old_filter = $db->filter_fetch_key  ( sub { ... } );
$old_filter = $db->filter_fetch_value( sub { ... } );
```

Usando estos filtros podemos serializar valores del hash que sean estructuras de datos complejas. En el siguiente código el hash DBM `%h` contiene como valores referencias a cadenas. El filtro vuelca con `Data::Dumper` el valor complejo y lo comprime usando la función `compress` del módulo `Compress::Zlib`. El proceso inverso consiste en descomprimir y evaluar:

```
$ cat -n  dbwithfilter.pl
 1  #!/usr/bin/perl
 2  use warnings;
 3  use Compress::Zlib;
 4  use DB_File;
 5  use Data::Dumper;
 6
 7  unlink 'mldbmtest.dat';
 8
 9  $h = tie my %db1, 'DB_File', 'mldbmtest.dat', O_CREAT | O_RDWR, 0666
10      or die "No se pudo inicializar el fichero MLDBM: $!\n";
11
12  $h->filter_store_value(sub { $_ = compress(Dumper($_)) });
13  $h->filter_fetch_value(sub { $_ = eval(uncompress($_)) });
14
15  %db1 = (
16      'alu2511' => [ 'a'x30 ],
17      'alu2233' => [ 'b'x30 ]
18  );
19
20  print Data::Dumper->Dump( [ \%db1 ] );
```

Este otro programa lee el hash DBM creado por el programa anterior:

```
$ cat -n dbwithfilterretrieve.pl
#!/usr/bin/perl
use warnings;
use Compress::Zlib;
use DB_File;
use Data::Dumper;

my $h = tie my %db2, 'DB_File', 'mldbmtest.dat', O_RDWR, 0666
    or die "No se pudo leer el fichero MLDBM: $!\n";
$h->filter_store_value(sub { $_ = compress(Dumper($_)) });
$h->filter_fetch_value(sub { $_ = eval(uncompress($_)) });

print Data::Dumper->Dump( [ \%db2 ] );
```

Cuando se ejecutan estos dos programas obtenemos la siguiente salida:

```
$ dbwithfilter.pl
$VAR1 = {
          'alu2233' => [ 'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb' ],
          'alu2511' => [ 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa' ]
        };
lhp@nereida:~/Lperl/src$ dbwithfilterretrieve.pl
$VAR1 = {
          'alu2233' => [ 'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb' ],
          'alu2511' => [ 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa' ]
        };
```

MLDBM

El módulo *Multi level DBM MLDBM* permite guardar estructuras de datos Perl en ficheros DBM (Véase 9.2). 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.

Es por esto que cuando se usa admite como parámetros una especificacción de las librerías de manejo de `DBM` y del serializador. Por ejemplo:

```
use MLDBM;                           # Manejadores por defecto: SDBM
#use MLDBM qw(DB_File FreezeThaw);    # Usaremos FreezeThaw para serialización
#use MLDBM qw(DB_File Storable);      # Usaremos Storable para serialización
```

Véase un ejemplo de uso del módulo `MLDBM`:

```
$ cat -n mldbmtest2.pl
#!/usr/bin/perl
use warnings;
use Data::Dumper;
use MLDBM qw( DB_File Storable );
use Fcntl; # Para importar constantes O_CREAT O_RDWR

unlink 'mldbmtest.dat';

```

```
 9  tie my %db1, 'MLDBM', 'mldbmtest.dat', O_CREAT | O_RDWR, 0666
10      or die "No se pudo inicializar el fichero MLDBM: $!\n";
11
12  %db1 = (
13      'alu2511' => {
14          nombre => 'Josefina Fernández Pérez',
15          tel => '922 00 00 00',
16          fecha => '22/07/84'
17      },
18      'alu2233' => {
19          nombre => 'Ana Feliú Forner',
20          tel => '922 00 11 22',
21          fecha => '14/06/85'
22      }
23  );
24
25  untie %db1;
26
27  tie my %db2, 'MLDBM', 'mldbmtest.dat', O_RDWR, 0666
28      or die "No se pudo leer el fichero MLDBM: $!\n";
29
30  print Data::Dumper->Dump( [ \%db2 ] );
31
32  untie %db2;
33
34  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'
  }
};
```

Limitaciones

Una limitación que proviene de que los MLDBM y los DBM son atados es que sólo la escrituras directas del tipo `$db1{one} = { a => 'e' }` produce actualización. Una escritura indirecta como `$db1{one}{a} = 'e'` no produce actualización. Véase el siguiente ejemplo:

```
$ cat -n dbwithfilter2.pl
 1  #!/usr/bin/perl
 2  use warnings;
 3  use DB_File;
 4  use Data::Dumper;
 5
```

```
 6  unlink 'mldbmtest.dat';
 7
 8  $h = tie my %db1, 'DB_File', 'mldbmtest.dat', O_CREAT | O_RDWR, 0666
 9      or die "No se pudo inicializar el fichero MLDBM: $!\n";
10
11  $h->filter_store_value(sub { $_ = Dumper($_) });
12  $h->filter_fetch_value(sub { $_ = eval($_) });
13
14  %db1 = (
15      'one' => { a => 'b' },
16      'two' => { c => 'd' }
17  );
18
19  $db1{one}{a} = 'e';
20
21  $Data::Dumper::Indent = 0;
22  print "Asignacion indirecta:\n",Dumper( \%db1 ),"\n";
23
24  $db1{one} = { a => 'e' };
25
26  print "Asignacion directa:\n",Dumper( \%db1  ),"\n";
```

Cuando se ejecuta produce la siguiente salida:

```
lhp@nereida:~/Lperl/src$ dbwithfilter2.pl
Asignacion indirecta:
$VAR1 = {'one' => {'a' => 'b'},'two' => {'c' => 'd'}};
Asignacion directa:
$VAR1 = {'one' => {'a' => 'e'},'two' => {'c' => 'd'}};
```

La explicación reside en que al hacer `$db1{one}{a} = 'e'` no se produce una llamada a `STORE`; de hecho se produce una llamada a `FETCH`. Como no se llama a `STORE` no se produce la modificación de la estructura de datos atada.

9.4. Class::DBI

9.4.1. Instalar una base de Datos

Debe existir una base de datos previa. Además el módulo `DBI` deberá estar instalado.

Sigue la descripción SQL de la base de datos que usaremos en las siguietnes secciones:

```
lhp@nereida:~/Lperl/src/CLASS_DBI$ cat -n example.sql
 1  CREATE TABLE artist (
 2    artistid INTEGER PRIMARY KEY,
 3    name TEXT NOT NULL
 4  );
 5
 6  CREATE TABLE cd (
 7    cdid INTEGER PRIMARY KEY,
 8    artist INTEGER NOT NULL REFERENCES artist(artistid),
 9    title TEXT NOT NULL
10  );
11
12  CREATE TABLE track (
```

```
13    trackid INTEGER PRIMARY KEY,
14    cd INTEGER NOT NULL REFERENCES cd(cdid),
15    title TEXT NOT NULL
16  );
```

Veamos una descripción mas detallada usando el comando `.dump`:

```
lhp@nereida:~/Lperl/src/CLASS_DBI$ sqlite3 example.db
SQLite version 3.3.8
Enter ".help" for instructions
sqlite> .dump
BEGIN TRANSACTION;
CREATE TABLE artist (
  artistid INTEGER PRIMARY KEY,
  name TEXT NOT NULL
);
INSERT INTO "artist" VALUES(1, 'Michael Jackson');
INSERT INTO "artist" VALUES(2, 'Eminem');
CREATE TABLE cd (
  cdid INTEGER PRIMARY KEY,
  artist INTEGER NOT NULL REFERENCES artist(artistid),
  title TEXT NOT NULL
);
INSERT INTO "cd" VALUES(1, 1, 'Thriller');
INSERT INTO "cd" VALUES(2, 1, 'Bad');
INSERT INTO "cd" VALUES(3, 2, 'The Marshall Mathers LP');
CREATE TABLE track (
  trackid INTEGER PRIMARY KEY,
  cd INTEGER NOT NULL REFERENCES cd(cdid),
  title TEXT NOT NULL
);
INSERT INTO "track" VALUES(1, 3, 'The Way I Am');
INSERT INTO "track" VALUES(2, 3, 'Stan');
INSERT INTO "track" VALUES(3, 1, 'Billie Jean');
INSERT INTO "track" VALUES(4, 2, 'Leave Me Alone');
INSERT INTO "track" VALUES(5, 2, 'Smooth Criminal');
INSERT INTO "track" VALUES(6, 1, 'Beat It');
INSERT INTO "track" VALUES(7, 2, 'Dirty Diana');
COMMIT;
sqlite>
```

9.4.2. Describir la Aplicación

`Class::DBI` trabaja con un modelo una clase/una tabla. Las tablas de la base de datos deben estar previamente definidas e iniciadas.

Para describir la base de datos de manera que los objetos puedan ser construidos por`Class::DBI` se requieren varios pasos.

Establecer una Clase Base

Es conveniente establecer una clase base de la cual hereden las clases asociadas con las diferentes tablas. La introducción de esta clase proporciona un lugar conveniente para las sobreescrituras y extensiones que queramos hacer a `Class::DBI`.

```
$ cat -n synopsis1.pl
```

```
package Music::DBI;
use strict;

use base 'Class::DBI';
Music::DBI->connection("dbi:SQLite:example.db");
```

Lo primero es conocer como acceder a la base de datos. Esto se hace por medio del método `connection`. Esto hace que todas las clases/tabla que heredan de la clase base compartan la misma conexión.

Establecer cada Clase y Declarar las Columnas

```
package Music::Artist;
use base 'Music::DBI';
Music::Artist->table('artist');
Music::Artist->columns(All => qw/artistid name/);
Music::Artist->has_many(cds => 'Music::CD');

package Music::CD;
use base 'Music::DBI';
Music::CD->table('cd');
Music::CD->columns(All => qw/cdid artist title/);
Music::CD->has_many(tracks => 'Music::Track');
Music::CD->has_a(artist => 'Music::Artist');

package Music::Track;
use base 'Music::DBI';
Music::Track->table('track');
Music::Track->columns(All => qw/trackid cd title/);

my $artist = Music::Artist->insert({ name => 'U2' });

my $cd = $artist->add_to_cds({
  title  => 'October',
});

$cd->update;
```

Ejecución:

```
lhp@nereida:~/Lperl/src/CLASS_DBI$ perl synopsis1.pl
lhp@nereida:~/Lperl/src/CLASS_DBI$ sqlite3 example.db
SQLite version 3.3.8
Enter ".help" for instructions
sqlite> .dump
BEGIN TRANSACTION;
CREATE TABLE artist (
  artistid INTEGER PRIMARY KEY,
  name TEXT NOT NULL
);
INSERT INTO "artist" VALUES(1, 'Michael Jackson');
INSERT INTO "artist" VALUES(2, 'Eminem');
INSERT INTO "artist" VALUES(3, 'U2');
CREATE TABLE cd (
  cdid INTEGER PRIMARY KEY,
```

```
  artist INTEGER NOT NULL REFERENCES artist(artistid),
  title TEXT NOT NULL
);
INSERT INTO "cd" VALUES(1, 1, 'Thriller');
INSERT INTO "cd" VALUES(2, 1, 'Bad');
INSERT INTO "cd" VALUES(3, 2, 'The Marshall Mathers LP');
INSERT INTO "cd" VALUES(4, 3, 'October');
CREATE TABLE track (
  trackid INTEGER PRIMARY KEY,
  cd INTEGER NOT NULL REFERENCES cd(cdid),
  title TEXT NOT NULL
);
INSERT INTO "track" VALUES(1, 3, 'The Way I Am');
INSERT INTO "track" VALUES(2, 3, 'Stan');
INSERT INTO "track" VALUES(3, 1, 'Billie Jean');
INSERT INTO "track" VALUES(4, 2, 'Leave Me Alone');
INSERT INTO "track" VALUES(5, 2, 'Smooth Criminal');
INSERT INTO "track" VALUES(6, 1, 'Beat It');
INSERT INTO "track" VALUES(7, 2, 'Dirty Diana');
COMMIT;
sqlite>
```

```
lhp@nereida:~/Lperl/src/CLASS_DBI$ cat -n synopsis2.pl
 1  #!/usr/local/bin/perl -w
..  ............................ # like in the former example
25
26  my $cd = Music::Artist->search_like(name => 'Eminem')->first;
27
28  print $cd->artistid,": ",$cd->name,"\n";
29
30  my $first = Music::Artist->retrieve(1);
31  print $first->name,"\n";
32
33  print "CDs\n";
34  for my $cd (Music::CD->retrieve_all) {
35    print "\t",$cd->title,"\n";
36  }
```

```
lhp@nereida:~/Lperl/src/CLASS_DBI$ perl synopsis2.pl
2: Eminem
Michael Jackson
CDs
        Thriller
        Bad
        The Marshall Mathers LP
```

9.5. DBIx::Class

```
  mkdir app
  cd app
  mkdir db
  cd db
```

Guarde el siguiente fichero `example.sql` en el directorio `db`:

```
$ cat example.sql
CREATE TABLE artist (
  artistid INTEGER PRIMARY KEY,
  name TEXT NOT NULL
);

CREATE TABLE cd (
  cdid INTEGER PRIMARY KEY,
  artist INTEGER NOT NULL REFERENCES artist(artistid),
  title TEXT NOT NULL
);

CREATE TABLE track (
  trackid INTEGER PRIMARY KEY,
  cd INTEGER NOT NULL REFERENCES cd(cdid),
  title TEXT NOT NULL
);
```

Capítulo 10

El Compilador de Perl

10.1. Los Paquetes `O` y `B`

Durante el proceso de compilación Perl traduce el código a una representación interna que puede ser ejecutada por el intérprete Perl. El intérprete Perl es un *procesador software* orientado a pila.

El producto de la traducción - que es la entrada para el intérprete - es un árbol. *Cada operación del intérprete es un nodo del árbol. Los argumentos de la operación son los hijos del nodo.*

Hay diferentes tipos de operadores. Por ejemplo el operador binario `add` toma dos valores de la pila y empuja la suma. El operador unario `readline` toma un manejador de ficheros de la pila y empuja el valor leído.

Algunos operadores actúan sobre listas. Por ejemplo `print` toma un número variable de elementos de la pila. Los operadores de listas se apoyan en un uso previo del operador `pushmark` para marcar el comienzo de la listade argumentos.

Los módulos en la familia `B::*` permiten acceder a dicho árbol (`B`, `B::Deparse`, `B::Bytecode`, `B::ByteLoader`, etc.). El módulo `B` representa cada operador como una subclase de la clase `B::OP`. Estas clases contienen métodos que nos permiten obtener información sobre o modificar el estado interno del operador. Por ejemplo obtener los hijos de un operador. Para tener una visión en profundidad del tema puede consultar los libros de Simon Cozens [**?**] y [8].

Recorrer el Árbol

El siguiente programa `Btree3.pl` implanta una subrutina `treetrav` que muestra el árbol generado por el compilador para una subrutina dada.

```
pp2@nereida:~/src/perl/B$ cat -n Btree3.pl
 1  #!/usr/local/bin/perl -w
 2  use strict;
 3  use B::Utils;
 4
 5  my $s = sub {
 6    my $a = shift;
 7    print $a+1;
 8  };
 9
10  sub treetrav {
..    ...........
22  }
23
24  my $b = B::svref_2object($s); # Objeto B::CV
25  print "*** Tree ***\n";
26  my $op = $b->ROOT;
27  treetrav($op, 0);
```

La llamada a `B::svref_2object` sobre la subrutina referenciada por `$s` nos produce un objeto `B::CV` que representa el árbol de codigo resultante.

Cuando `treetrav` es llamada con el objeto y un sangrado inicial igual a 0 nos produce la siguiente salida:

```
pp2@nereida:~/src/perl/B$ Btree3.pl
*** Tree ***
leavesub(               #subroutine exit
  lineseq(              #line sequence
    sassign(            #scalar assignment
      shift(            #shift
        rv2av(          #array dereference
          gv(           #glob value
          ) # end gv
        ) # end rv2av
      ) # end shift
      padsv(            #private variable
      ) # end padsv
    ) # end sassign
    nextstate(          #next statement
    ) # end nextstate
    print(              #print
      add(              #addition (+)
        padsv(          #private variable
        ) # end padsv
        const(          #constant item
        ) # end const
      ) # end add
    ) # end print
  ) # end lineseq
) # end leavesub
```

La función `treetrav` usa los métodos `name` y `desc` de la subclase `B::OP` para obtener el nombre y la descripción de la operación actual. El método `kids` en `B::Utils` devuelve la lista de hijos:

```
sub treetrav {
  my $op = shift;
  my $indent = shift;

  my $spaces = " "x$indent;
  print $spaces.$op->name."(\t\t#".$op->desc;
    print "\n";
    for ($op->kids()) {
      treetrav($_, $indent+2);
    }
    print "$spaces) # end ".$op->name;
  print "\n";
}
```

Hay varias subrutinas de recorrido del árbol de códigos. Una de las mas sencillas es `walkoptree_simple` (en `B::Utils`):

```
walkoptree_simple($op, \&callback, [$data])
```

Orden de Ejecución En el párrafo anterior recorríamos el árbol según su orden jerárquico. Sin embargo el árbol está a su vez hilvanado en orden de ejecución. Esto es: superpuesta a la estructura de árbol existe una lista que recorre el árbol según el flujo de control. A esta estructura de datos se la denomina *Grafo de Flujo de Control* o *Control flow Graph* (*CFG*).

El método `START` de un objeto `B::CV` nos da el punto de entrada en la ejecución. El método `next` de un objeto `B::OP` proporciona el siguiente nodo a ejecutar.

```
pp2@nereida:~/src/perl/B$ cat -n bex2.pl
 1  #!/usr/local/bin/perl -w
 2  use strict;
 3  use B;
 4
 5  my $s = sub {
 6    my $a = shift;
 7    print $a+1;
 8  };
 9
10  my $b = B::svref_2object($s);
11  my $op = $b->START;
12  my ($class, $name, $desc);
13  format STDOUT =
14  @<<<<<<<<<@|@<<<<<<<<<<<<@|@<<<<<<<<<<<<<<<<<<<<<<<<<
15  $class,   '|', $name,    '|', $desc
16  .
17  do {
18    ($class, $name, $desc) = (B::class($op), $op->name, $op->desc);
19    write;
20  } while $op = $op->next and !$op->isa("B::NULL");
```

Es necesario comprobar que `$op` no es un objeto `B::NULL` ya que el puntero `next` es un puntero C que `B` deja disponible sin envoltorio alguno.

La ejecución produce la secuencia de operaciones en orden de ejecución:

```
pp2@nereida:~/src/perl/B$ bex2.pl | cat -n
 1  COP        | nextstate    | next statement
 2  PADOP      | gv           | glob value
 3  UNOP       | rv2av        | array dereference
 4  UNOP       | shift        | shift
 5  OP         | padsv        | private variable
 6  BINOP      | sassign      | scalar assignment
 7  COP        | nextstate    | next statement
 8  OP         | pushmark     | pushmark
 9  OP         | padsv        | private variable
10  SVOP       | const        | constant item
11  BINOP      | add          | addition (+)
12  LISTOP     | print        | print
13  UNOP       | leavesub     | subroutine exit
```

```
my $s = sub {
  my $a = shift;
  print $a+1;
};
```

Apéndice

Código de `01MartelloAndTothBook.t`

```
lhp@nereida:/tmp/Algorithm-Knap01DP-0.01/t$ cat -n 01MartelloAndTothBook.t
 1  # Before ‘make install’ is performed this script should be runnable with
 2  # ‘make test’. After ‘make install’ it should work as ‘perl Algorithm-Knap01DP.t’
 3
 4  #########################
 5  use strict;
 6  use Test::More  tests => 11;
 7
 8  BEGIN { use_ok(’Algorithm::Knap01DP’, qw/Knap01DP ReadKnap/); }
 9
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];
23
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  }
36
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/;
41
42  TODO: { # I plan to provide a function to find the vector solution ...
```

```
  local $TODO = "Randomly generated problem";
  can_ok(’Algorithm::Knap01DP’, ’GenKnap’);
}
```

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 );
}

```

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 R. L. *Programming Perl.* O'Reilly, USA, 1996.

[4] Richard Foley and Andy Lester. *Pro Perl Debugging.* Apress, Berkely, CA, USA, 2005.

[5] Jefrrey E.F. Friedl. *Mastering Regular Expressions.* O'Reilly, USA, 1997. ISBN 1-56592-257-3.

[6] Lincoln D. Stein. *Network Programming with Perl.* Addison Wesley, USA, 2001. ISBN 0-201-61571-1.

[7] Christiansen T. and Schwartz L. *Perl Cookbook.* O'Reilly, USA, 1998. ISBN 1-56592-243-3.

[8] Tim Jenness and Simon Cozens. *Extending and embedding Perl.* Manning Publications Co., Greenwich, CT, USA, 2003.

[9] Joseph Hall and Randall L. Schwartz. *Effective Perl Programming. Writing Better Programs with Perl.* Addison Wesley, USA, 2001. ISBN 0-201-41975-0.

[10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 1986.

[11] Damian Conway. *Perl Best Practices.* O'Reilly Media, Inc., 2005.

[12] Michael Pilato. *Version Control With Subversion.* O'Reilly & Associates, Inc. `http://svnbook.red-bean.com/en/1.4/svn-book.html`, Sebastopol, CA, USA, 2004.

[13] Mark Jason Dominus. *Higher-Order Perl.* Morgan Kaufmann Publishers, 2005.

[14] Peter Scott. *Perl Medic: Maintaining Inherited Code.* Addison Wesley, USA, 2004. ISBN 0201795264.

[15] S. Martello and P. Toth. *Knapsack Problems.* John Wiley and Sons, Chichester, 1990.

[16] Jesse Vincent, Dave Rolsky, Darren Chamberlain, Richard Foley, and Robert Spier. *RT Essentials.* O'Reilly Media, Inc., 2005.

[17] Sam Tregar. *Writing Perl Modules for CPAN.* Apress. Descarguelo desde `http://www.apress.com/free/download_free.html`, Berkely, CA, USA, 2002.

[18] Conway D. *Object Oriented Perl.* Manning, Greenwich, USA, 2000.

[19] Simon Cozens. *Advanced Perl Programming, Second Edition.* O'Reilly Media, 2005.

[20] Mike Chirico. *SQLite Tutorial.* `http://souptonuts.sourceforge.net/readme_sqlite_tutorial.html`, 2004.