Índice general

Índice de cuadros

Capítulo 1

¿Qué es perl?

Perl (*Practical Extracting and Reporting Language) es un lenguaje de programación interpretado. Esto quiere decir que un programa escrito en perl no necesita de ninguna clase de compilación previa a su ejecución.*

El nombre proviene de su origen, ya que inicialmente fue diseñado para extraer y formatear informes partiendo de ficheros de información, de ahí que sus expresiones regulares y capacidades para trabajar con ficheros y cadenas sean de las más potentes disponibles en la actualidad. A posteriori se amplió su uso para acabar convirtiéndose en un lenguaje de scripting de uso general, sustituyendo en ocasiones a scripts programados en shell y, sobre todo, convirtiéndose en el principal lenguaje a la hora de programar CGIs que recojan la información enviada en formularios HTML.

Capítulo 2

Obtención e instalación de Perl

2.1. Obtención de perl

Podemos encontrar el Perl que necesitamos en http://www.cpan.org/ports, y tan sólo deberemos elegir el adecuado a nuestro sistema operativo (linux, solaris, Windows). Puesto que este curso es para usuarios de linux, nos centraremos en él para los ejemplos que seguiremos a lo largo del curso.

2.2. Instalación de Perl

Para instalar Perl tendremos que actuar de diferente manera según la distribución sobre la que estemos trabajando:

2.2.1. Debian

```
apt-get install perl5
```

2.2.2. Red Hat, Suse

```
rpm -i perl5-00503.i386.rpm
```

Capítulo 3

Mi primer programa en perl

Antes de empezar cualquier programa en Perl quizás convenga que conozcamos algunas normas básicas de la programación en este lenguaje:

- *Los comentarios: se considerarán comentarios, y por tanto serán ignoradas por el intérprete, todas aquellas líneas que comiencen por el carácter almohadilla (#).*
- *Todas las órdenes deben terminar en un punto y coma, o bien en una apertura de llave ({) de tratarse de una estructura de control (if, for, while,...).*

Para irnos familiarizando con la estructura de un programa escrito en Perl vamos a hacer uno muy sencillo:

3.1. Escribir el programa

El código fuente de un programa Perl es texto plano, por lo que cualquier editor de texto servirá a nuestros propósitos, por ejemplo vim; por tanto:

```
rcortega@cursos> vim programa.pl
```

Nótese la extensión ".pl" del programa. Aunque ya sabemos que en linux no es necesario seguir estrictamente las convenciones acerca de las extensiones de archivo, hacerlo nos ayudará a reconocer a primer golpe de vista el lenguaje en el que hemos escrito un programa, sin necesidad siquiera de leer el contenido del fichero. Ahora teclearemos el texto del progama:

```
#!/usr/bin/perl

print "Hallo Welt\n";
```

En la primera línea lo primero que llama la atención es que comienza por almohadilla, por lo que podríamos pensar que sería ignorada, pero ésta es la única excepción a la norma, ya que es la línea que indica a nuestro ordenador dónde debe buscar el intérprete que deberá ejecutar nuestro programa. Otra cosa que puede llamarnos la atención es que no termina por el carácter ";" ni "{", una excepción más, pero no nos acostumbraremos, o tendremos que habituarnos también a estos mensajes tantas veces repetidos de "Syntax error". En la tercera línea aparece lo que es el programa en sí, la orden que indica al intérprete de Perl que queremos que muestre por pantalla (print) la frase "Hallo Welt". Como observaremos, esta línea sí que termina con el carácter ";". Otro detalle a destacar, es que si no queremos que el prompt nos reaparezca inmediatamente después de nuestra frase, sino que nuestra intención es que lo haga en la línea siguiente, debemos indicárselo explícitamente mediante el carácter de escape para "nueva línea" (\n).

3.2. Ejecutar el programa

Lo más seguro es que nuestro sistema tenga definidos para nuestro fichero los permisos de escritura y lectura, pero no de ejecución:

```
rcortega@cursos> ls -la
-rw-r--r--    1 rcortega rcortega       39 nov 10 21:50 programa.pl
```

Para añadirle los permisos necesarios nos bastará con teclear:

```
rcortega@cursos> chmod u+x programa.pl
```

Con lo que al comprobar de nuevo los permisos de nuestro programa deberíamos ver algo así:

```
rcortega@cursos> ls -la
-rwxr--r--    1 rcortega rcortega       39 nov 10 21:52 programa.pl
```

Ahora, para ejecutar nuestro programa no tenemos más que, desde una shell del sistema, teclear:

```
rcortega@cursos> ./programa.pl
Hallo Welt
rcortega@cursos>
```

Nótese lo que comentábamos antes del '\n', ya que si no lo hubiéramos puesto la salida habría sido algo así:

```
rcortega@cursos> ./programa.pl
Hallo Weltrcortega@cursos>
```

3.3. Otra manera de hacerlo

Si no queremos no tenemos por qué indicar en la primera línea el path al intérprete que debe ejecutar nuestro programa, dejando esa tarea a nuestra variable de entorno"`PATH`*". Haciéndolo de este modo nuestro programa quedaría reducido a una sola línea:*

```
print "Hallo Welt\n";
```

Y para ejecutarlo deberíamos escribir:

```
rcortega@cursos> perl programa.pl
```

Pero en este caso deberemos asegurarnos de que el path de nuestro intérprete de perl se encuentra en la variable de entorno PATH; para estar seguros no tenemos mas que teclear:

```
rcortega@cursos> echo $PATH
```

y comprobar que el directorio

```
/usr/bin
```

aparece.

3.4. Programas con parámetros

En este primer programa no hemos usado parámetros de la línea de comandos, pero puede que en algún momento necesitemos hacer un programa con esa capacidad, por lo que conviene saber dónde podemos encontrar esas palabras que han introducido a continuación del nombre de nuestro programa.
Igual que en un programa en java o en otros lenguajes, los parámetros quedan guardados en un array, almacenando en cada una de sus posiciones una palabra (el concepto de array y su funcionamiento se estudia más adelante). En el caso de Perl este array es el llamado @ARGV, por lo que si quisiéramos imprimir los parámetros recibidos, el programa podría ser algo así:

```
for($i = 0; $i <= $#ARGV; $i++){
    print "$ARGV[$i]\n";
}
```

Esta sección se ha incluido aquí a efectos de facilidad de consulta, así que no debe preocuparnos entender con exactitud lo que hace el código de arriba, ya lo veremos más adelante.

Capítulo 4

Variables en Perl

4.1. Tipado fuerte vs. Tipado débil

Lo primero que debemos saber a la hora de trabajar con variables en Perl es que es un lenguaje con tipado débil. Esto significa que el intérprete de Perl no distingue entre tipos de variables, y si le decimos que sume una cadena a un entero lo hará, claro que el resultado seguramente no sea el esperado. Sin embargo, otros lenguajes fuertemente tipados, como java, sencillamente nos enviarían un error en tiempo de compilación indicándonos que estamos intentando realizar una operación no permitida.

4.2. Usando variables en perl

En perl, cuando queremos guardar un dato individual lo almacenamos en un escalar. Un escalar puede ser un número entero, uno real, o un carácter o cadena de carácteres. La única diferencia práctica que vamos a encontrar es el resultado obtenido al hacer ciertas operaciones con ellos.

A la hora de trabajar con las variables de perl hay que saber que por defecto todas son de ámbito global, es decir, que si declaras una variable en una función, y otra del mismo nombre en una función diferente, para el intérprete va a ser la misma variable, con lo que cualquier modificación que hagamos sobre una afectará a la otra. Esto tiene especial importancia cuando se llama a una de esas funciones desde la otra, de modo que el resultado final puede verse afectado.

Para ver sobre el terreno esto que acabamos de decir vamos a ver diferntes tipos de operaciones con las variables de perl.

Operación asignación

La primera operación que vamos a ver es la asignación, que es la que utilizamos cuando asociamos un valor a una variable. Hasta ese momento, la variable no esta definida, tan sólo está declarada. Veamos esto con ejemplos; si ponemos

```
$var;
```

tan sólo estamos declarando la variable, es como decir

"Voy a usar una variable que se llame var"

sólo podremos decir que tenemos nuestra variable definida cuando hagamos

```
$var = 2;
```

Resulta importante no confundir el operador asignación (=) con el operador comparación (==), que veremos más adelante.

Operación	*Operador*
suma	+
resta	-
multiplicación	*
división	/

Cuadro 4.1: **Operadores aritméticos**

Operaciones aritméticas
Las operaciones aritméticas son las operaciones matemáticas habituales (suma, resta, multiplicación y división). Los operadores a emplear son:

*Debemos tener en cuenta que las operaciones aritméticas **deben realizarse con números**, ya sean enteros o reales, si no lo hacemos así, perl no se quejará (recordemos que es un lenguaje de tipado débil), pero el resultado posiblemente no sea el deseado. Veamos esto con ejemplos; si hacemos un programa que sea:*

```
$var1 = 4;
$var2 = 3;
$var3 = $var1 + $var2;

print "$var3\n";
```

la salida obtenida será algo así:

```
rcortega@cursos> ./suma.pl
7
```

Esto es lo esperado, al fin y al cabo estábamos sumando dos números, ¿no? Pero... ¿qué sucedería si sumásemos dos cadenas? Veámoslo con otro ejemplo:

```
$var1 = "cadena1";
$var2 = "cadena2";
$var3 = $var1 + $var2;

print "$var3\n";
```

pues el resultado sería el siguiente:

```
rcortega@cursos> ./suma.pl
0
```

es decir, el resultado de sumar aritméticamente dos cadenas es cero. Lo que sucede, es que el valor aritmético de una cadena de carácteres lo toma como cero, de este modo, si sumamos un número con una cadena, de esta forma:

```
$var1 = 23;
$var2 = "cadena";
$var3 = $var1 + $var2;

print "$var3\n";
```

el resultado será:

```
rcortega@cursos> ./suma.pl
23
```

Lo más sorprendente puede ser si sumamos un número a una cadena que contiene dígitos en su parte inicial:

```
$var1 = 23;
$var2 = "23cadena";
$var3 = $var1 + $var2;

print "$var3\n";
```

puesto que obtendríamos:

```
rcortega@cursos> ./suma.pl
46
```

Nótese que esto no sucede si los dígitos aparecen al final de la cadena, es decir, el resultado de:

```
$var1 = 23;
$var2 = "cadena23";
$var3 = $var1 + $var2;

print "$var3\n";
```

es:

```
rcortega@cursos> ./suma.pl
23
```

Esto que hemos visto para la suma es aplicable al resto de las operaciones aritméticas.

Hay ciertas operaciones, que por la frecuencia con que se utilizan, se ha implementado una serie de operadores "abreviados" para realizarlas:
Al igual que los anteriores, los operadores abreviados han de utilizarse con números.

Operación	*Ejemplo*	*Equivalencia*
incremento	`$num++`	`$num = $num + 1`
decremento	`$num-`	`$num = $num - 1`
autosuma	`$num += 3`	`$num = $num + 3`

Cuadro 4.2: **Operadores aritméticos abreviados**

Operaciones de comparación

Las operaciones de comparación no nos devuelven ningún número, ni ninguna expresión, tan sólo nos dicen si lo que estamos "preguntando" es cierto o falso; por ejemplo, si ponemos:

```
$var1 = 4;
$var2 = 3;

if($var1 > $var2){
    print "var1 es mayor que var2\n"
}
```

lo que estamos preguntando es

"¿$var1 es mayor que $var2?"

y en este caso la respuesta es "sí", por lo que por la pantalla nos saldrá la frase

```
rcortega@cursos> ./comparacion.pl
var1 es mayor que var2
```

Resumiendo, cuando empleamos un operador de comparación la respuesta va a ser "verdadero" (valor booleano true) o "falso" (valor booleano false). Sin embargo, en perl no existen las variables booleanas `true` *y* `false`*, por lo que estas operaciones de comparación sólo tienen sentido dentro de sentencias condicionales, que veremos más adelante. Antes de continuar echemos un vistazo a la lista de operadores de comparación:*
Una vez más nos encontramos con que los operadores son diferentes si las variables son de tipo numérico o son cadenas de carácteres, y no debemos olvidar que perl nunca nos avisará de estos errores, se limitará a ejecutar lo que nosotros le hayamos dicho, y esto puede llevar a error; por ejemplo:

```
$var1 = 100;
$var2 = 9;
```

Comparación	*Números*	*Cadenas*
Igual	`==`	`eq` (*EQual*)
Distinto	`!=`	`ne` (*Not Equal*)
Mayor	>	`gt` (*Greater Than*)
Menor	<	`lt` (*Less Than*)
Mayor o igual	>=	`ge` (*Greater or Equal*)
Menor o igual	<=	`le` (*Less or Equal*)

Cuadro 4.3: **Operadores de comparación**

```
if($var1 > $var2){
    print "$var1 es mayor que $var2\n";
}

if($var1 < $var2){
    print "$var1 es menor que $var2\n";
}
```

tendría el siguiente resultado:

```
rcortega@cursos> ./comparacion.pl
100 es mayor que 9
```

pero si modificamos nuestro programa de este modo:

```
$var1 = 100;
$var2 = 9;

if($var1 gt $var2){
    print "$var1 es mayor que $var2\n";
}

if($var1 lt $var2){
    print "$var1 es menor que $var2\n";
}
```

el resultado sería:

```
rcortega@cursos> ./comparacion.pl
100 es menor que 9
```

y esto es debido a que cuando usamos los operadores para cadenas, lo que hace es tomar los carácteres uno a uno y comparar sus valores en la tabla ASCII, y como el valor del carácter "1" es menor que el del carácter "9" concluye, sin continuar comparando, que `9` *es mayor que* `100`*.*

Otras acciones con variables

Por último nos queda hablar de ciertas acciones (porque quizás no merezcan el nombre de operaciones propiamente dichas) que pueden realizarse con las variables:

- ***Interpolación****: es cuando "insertamos" una variable dentro de otra, por ejemplo:*

  ```
  $var1 = 3;
  $var2 = "El valor de var1 es $var1\n";

  print $var2;
  ```

  tiene como salida:

  ```
  rcortega@cursos> ./interpolacion.pl
  El valor de var1 es 3
  ```

es decir, hemos interpolado la variable $var1 en $var2. Nótese que para poder interpolar variables debemos emplear las comillas dobles (“”) y no las simples (‘’), ya que estas últimas no permiten la interpolación de variables, por ejemplo:

```
$var1 = 3;
$var2 = ‘El valor de var1 es $var1\n’;

print $var2;
```

daría como salida:

```
rcortega@cursos> ./interpolacion.pl
El valor de var1 es $var1\nrcortega@cursos>
```

porque no ha interpretado la variable para obtener su valor ni ha interpretado la secuencia de escape \n.

- ***Concatenación**: se dice que concatenamos dos variables cuando las “pegamos” una detrás de la otra. El operador concatenación en perl es el punto (“.”):*

  ```
  $var1 = "cadena1";
  $var2 = "cadena2";
  $eol = "\n";

  print $var1.$var2.$var3;
  ```

  da la salida:

  ```
  rcortega@cursos> ./concatenacion.pl
  cadena1cadena2
  ```

  Nótese que es el mismo resultado que si interpoláramos las dos variables dentro de una cadena más grande definida entre comillas dobles:

  ```
  $var1 = "cadena1";
  $var2 = "cadena2";
  $eol = "\n";

  print "$var1$var2$var3";
  ```

Capítulo 5

Variables que contienen variables: arrays y arrays asociativos

En perl encontramos dos tipos de variables cuya utilidad es contener otras variables dentro de sí. Esto puede ser de gran utilidad, como veremos más adelante, para emplearlas dentro de bucles así como para almacenar escalares de una forma ordenada.

5.1. Arrays

Un array es una variable que es capaz de almacenar variables en su interior, algo así como un cajón con compartimentos donde podemos guardar nuestras variables (escalares u otro array). Este almacenamiento tiene lugar de una forma ordenada, de modo que cada variable guardada tiene asignado un número, que nos permitirá extraerla para emplearla en nuestro programa. Pero vayamos por pasos, lo primero es saber que igual que un escalar va siempre precedido del carácter "$", un array irá siempre precedido del carácter "@", por ejemplo `@cajon` *sería un array llamado "cajon". Veamos ahora que podemos hacer con estos arrays:*

5.1.1. Inicialización

Un array puede inicializarse de dos maneras:

1. *Todo de una vez:*

```
@objetos = ("mesa", "silla", "armario", "percha");
```

2. *Elemento a elemento:*

```
$objetos[0] = "mesa";
$objetos[1] = "silla";
$objetos[2] = "armario";
$objetos[3] = "percha";
```

Ambas son totalmente equivalentes, y dan lugar al mismo array, pero hay que destacar que en el segundo caso, cuando nos referimos a elementos concretos dentro del array y no al array en su conjunto, el nombre del mismo va precedido de "$" y no "@".

5.1.2. Número de elementos del array

Lo primero que hay que destacar de los elementos de un array en el aspecto numérico, es que las posiciones empiezan a contar en 0 y acaban en n-1, donde n es el número total de elementos contenidos en el array. Para saber el número de elemntos que tiene un array se emplea la variable $#<nombre del array>, por ejemplo:

```
@objetos = ("mesa", "silla", "armario", "percha");
print "$#objetos\n";
```

daría como salida:

```
rcortega@cursos> ./array.pl
3
```

Nótese que lo que devuelve no es el número de elementos, sino la última posición ocupada en el array, es decir, en un array de n elementos nos devuelve n-1.

5.1.3. Accediendo a los elementos del array

Ya tenemos nuestro array creado y con unos cuantos elementos dentro, ahora queremos acceder a ellos, y esto también se puede hacer de varias maneras:

1. *Uno a uno:*

   ```
   $elemento = $objetos[2];
   print "$elemento\n";
   ```

   daría como salida:

   ```
   rcortega@cursos> ./array.pl
   armario
   ```

2. *Varios a la vez:*

   a) *Consecutivos:*

      ```
      @varios = @objetos[1..3];
      print "@varios\n";
      ```

      sería:

      ```
      rcortega@cursos> ./array.pl
      silla armario percha
      ```

   b) *Salteados:*

      ```
      @varios = @objetos[1,3];
      print "@varios\n";
      ```

      nos daría:

      ```
      rcortega@cursos> ./array.pl
      silla percha
      ```

Nótese que la variable $#objetos puede usarse también a la hora de extraerlos, por ejemplo, si quisiéramos los elementos del segundo al final, podríamos poner:

```
@varios = @objetos[1..$#objetos];
print "@varios\n";
```

y obtendríamos:

```
rcortega@cursos> ./array.pl
silla armario percha
```

5.1.4. Recorriendo el array

Ahora lo que queremos es recorrer el array entero, y como de costumbre, hay varias opciones a nuestra disposición:

1. *Con un bucle **for**, con dos sintaxis diferentes:*

   a)
   ```
   for($i = 0; $i <= $#objetos; $i++){
       print "$objetos[$i]\n";
   }
   ```

   b)
   ```
   for $i (0..$#objetos){
       print "$objetos[$i]\n";
   }
   ```

2. *Con la estructura **foreach**:*

```
foreach $objeto (@objetos){
    print "$objetos[$i]\n";
}
```

Todas estas formas dan lugar a la misma salida:

```
rcortega@cursos> ./array.pl
mesa
silla
armario
percha
```

5.2. Arrays asociativos

Los arrays asociativos, también conocidos como tablas hash consisten otra forma de almacenar varias variables en bajo un solo nombre, con la diferencia de que esta vez lo que se asocia a cada variable que introducimos no es necesariamente un número, sino que es otra variable que nosotros podemos elegir. En cuanto a notación, lo que distingue a un array asociativo es que el carácter que se antepone al nombre es "%" y no "$" ni "@". Pero vayamos por partes:

5.2.1. Inicialización

Hay varias formas de inicializar una tabla hash, todas ellas equivalentes:

1. *Todos a la vez, lo que a su vez se puede hacer de varias maneras:*

   a) *Poniendo* `%hash = (clave1,valor1,clave2,valor2,...)`*:*

   ```
   %familiares = ("madre","laura","padre","juan","hermano","jose","tio","pepe");
   ```

   b) *Poniendo* `($hashclave1,$hashclave2,...) = (valor1,valor2,...)`*:*

   ```
   ($familiares{"madre"},$familiares{"padre"},$familiares{"hermano"},
   $familiares{"tio"}) = ("laura","juan","jose","pepe");
   ```

   c) *Poniendo* `%hash = (clave1 =>valor1, clave2 =>valor2,...)`*:*

   ```
   %familiares = ("madre" => "laura", "padre" => "juan", "hermano"
   => "jose", "tio" => "pepe");
   ```

2. *De uno en uno:*

   ```
   $familiares{"madre"} = "laura";
   $familiares{"padre"} = "juan";
   $familiares{"hermano"} = "jose";
   $familiares{"tio"} = "pepe";
   ```

Nótese que como en el caso de los arrays, para referirnos a uno solo de sus elementos, el nombre de la tabla hash va precedido de "$" y no del "%" habitual.

5.2.2. Accediendo a un elemento

En una tabla hash, cada elemento está unívocamente identificado por su clave, y para extraerlo podremos hacerlo de esta manera:

```
$padre = $familiares{"padre"};
print "$padre\n";
```

lo que nos daría la siguiente salida:

```
rcortega@cursos> ./hash.pl
juan
```

5.2.3. Recorriendo el array asociativo

A la hora de recorrer una tabla hash nos encontramos con que los índices no son numéricos, por lo que no nos vale un bucle for que recorra toda la tabla. Pero esto no quiere decir que no se pueda recorrer ni mucho menos, y como de costumbre, podremos hacerlo de varias maneras:

1. *Obteniendo las claves con la funcion keys y recorriendo con un foreach:*

```
foreach(keys %familiares){
    $familiar = $familiares{$_};
    print "$familiar\n";
}
```

   obtendríamos la salida:

```
laura
jose
pepe
juan
```

2. *Obteniendo los valores con la función values:*

```
foreach (values %familiares){
    print "$_\n";
}
```

   nos daría el mismo resultado que el método anterior.

3. *Obteniendo los pares clave-valor con la función each:*

```
while(($parentesco,$nombre) = each %familiares){
    print "Mi $parentesco se llama $nombre\n";
}
```

   daría:

```
Mi madre se llama laura
Mi hermano se llama jose
Mi tio se llama pepe
Mi padre se llama juan
```

   Nótese que el orden en el que salen no tiene por qué ser el mismo en el que se introdujeron.

5.2.4. Añadiendo y eliminando elementos

Puede que en algún momento nos interese añadir un elemento, o borrarlo, sin tocar el resto de la tabla hash, entonces deberemos hacerlo de esta manera:

- ***Añadir:*** *Se hace de igual manera que insertar los elementos uno a uno:*

```
%familiares{"abuela"} = "maria";
```

- ***Eliminar:*** *Se hace con la función delete:*

```
delete %familiares{"abuela"};
```

Capítulo 6

Estructuras de control

Entendemos como estructuras de control a aquellos elementos de la sintaxis de un lenguaje que nos permiten modelar el curso de la ejecución de nuestro programa según nuestra conveniencia. Podríamos dividirlas en dos tipos: sentencias condicionales y bucles de repetición. Veámoslos por partes:

6.1. Sentencias condicionales

Las sentencias condicionales son las que nos permiten decidir que una determinada porción de código se ejecute sólo si se cumple una condición (de ahí su nombre). Estas decisiones se toman en tiempo de ejecución, y se podrían interpretar como:

> *"Si pasa esto, hago esto otro"*

y en un ejemplo:

```
$var1 = 3;
$var2 = 4;
if($var1 < $var2){
    print "$var1 es menor que $var2\n";
}
```

que significa algo asi como:

> *"Si $var1 es menor que $var2 entonce imprimo el mensaje ..."*

En nuestro caso $var1 es menor que $var2, por lo que la condición se cumple, y veremos que por pantalla nos sale:

```
rcortega@cursos> ./condicional.pl
3 es menor que 4
```

Ahora la pregunta inmediata es:

> *"¿Y si no se cumple?"*

entonces podemos usar la estructura else:

```
$var1 = 4;
$var2 = 3;
if($var1 < $var2){
    print "$var1 es menor que $var2\n";
}else{
    print "$var1 no es menor que $var2\n";
}
```

Nótese que haber puesto

> *"$var1 es mayor que $var2\n"*

habría sido incorrecto, ya que lo único que sabemos es que no es menor, pero nada impide que sean iguales.

6.1.1. Sentencias condicionales concatenadas

Si quisiéramos, siguiendo con el ejemplo anterior, comprobar además que sean menores, mayore o iguales, podríamos hacerlo de la siguiente manera:

```
$var1 = 4;
$var2 = 3;
if($var1 < $var2){
    print "$var1 es menor que $var2\n";
}elsif($var1 > $var2){
    print "$var1 es mayor que $var2\n";
}else{
    print "$var1 es igual que $var2\n";
}
```

Hemos concatenado dos sentencias condicionales, es decir, si no se cumple la primera, comprobaremos la segunda antes de irnos al else.

6.1.2. Sentencias condicionales anidadas

Otra forma de hacer lo anterior, que resulta equivalente sería anidando las dos sentencias condicionales:

```
$var1 = 4;
$var2 = 3;
if($var1 < $var2){
    print "$var1 es menor que $var2\n";
}else{
    if($var1 > $var2){
        print "$var1 es mayor que $var2\n";
    }else{
        print "$var1 es igual que $var2\n";
    }
}
```

6.1.3. Sentencia *unless*

La sentencia unless es equivalente a la negación de una sentencia if, supongamos que lo que nos interesa es la situación en que no se cumple la condición, entonces podríamos usar:

```
var1 = 4;
$var2 = 3;
if($var1 < $var2){
}else{
    print "$var1 no es menor que $var2\n";
}
```

o bien:

```
var1 = 4;
$var2 = 3;
if(!($var1 < $var2)){
    print "$var1 no es menor que $var2\n";
}
```

o, por último:

```
var1 = 4;
$var2 = 3;
unless($var1 < $var2){
    print "$var1 no es menor que $var2\n";
}
```

Resultando las tres equivalentes.

6.1.4. Qué es cierto y qué es falso en perl

Cuando usamos sentencias condicionales, sabemos que lo que se encuentra entre llaves después del `if` *se ejecutará si la condición entre paréntesis es cierta, y hemos visto las operaciones de comparación, que nos dan un resultado indicando si son ciertas o no, pero sin embargo, podemos utilizar también variables a modo de condición. En ese sentido debemos saber que:*

1. *Una variable no definida es* falsa*, por ejemplo, en el código:*

```
$var;
if($var){
    print "Es cierto\n";
}else{
    print "Es falso\n";
}
```

   el resultado sería:

```
rcortega@cursos> ./prueba.pl
Es falso
```

   mientras que si fuera:

```
$var = 2;
if($var){
    print "Es cierto\n";
}else{
    print "Es falso\n";
}
```

   el resultado sería:

```
rcortega@cursos> ./prueba.pl
Es cierto
```

2. *Todos los números dan* verdadero *salvo el cero, que da* falso*.*

3. *Cualquier cadena de carácteres que esté definida es* verdadera *salvo si es* “” *o* “0”*.*

6.2. Bucles de repetición

Los bucles de repetición son unas estructuras que nos permiten repetir un fragmento de código un determinado número de veces. Hay dos tipos de bucles de repetición: los bucles for y los bucles while, cada uno con su utilidad:

6.2.1. Bucle *for*

Se utilizan cuando se sabe de antemano (antes de empezar a iterar con el bucle) el número de veces que se va a tener que repetir. Nótese que el número se puede obtener en tiempo de ejecución, por ejemplo leyendo una variable de teclado que será el número de veces que se tenga que hacer, pero esto se deberá hacer antes de meterse en el bucle. Una vez hecho, el bucle se ejecutará una vez tras otra, aplicando en cada una de ellas el incremento indicado antes de pasar a la siguiente iteración, y si al ir a empezar detecta que se ha excedido el límite indicado, sale del bucle y continúa la ejecución del programa. La estructura es:

```
for(<inicialización>;<límite>;<incremento/decremento>){ ... }
```

Por ejemplo, si queremos un bucle que nos imprima los 10 primeros números enteros:

```
for($i = 0; $i <= 10; $i++){
    print ‘‘$i\n’’;
}
```

Nótese que la inicialización, el límite y el incremento pueden ser cualquier número, tanto positivo como negativo, y que si dentro del cuerpo del bucle modificamos la variable de control (en el ejemplo $i) modificaremos el comportamiento del bucle.

6.2.2. Bucle *while*

Se utiliza cuando no sabemos el número de repeticiones que vamos a tener que hacer antes de meternos en el bucle, y será en una de las iteraciones cuando se dé la condición de fin del bucle. El cuerpo del bucle se repetirá siempre que se verifique la condición, deteniéndose cuando deje de hacerlo. La estructura es:

```
while(<condicion>){ ... }
```

Por ejemplo, si queremos leer de teclado hasta que el usuario teclee la palabra "FIN", deberíamos hacer esto:

```
$linea = ‘‘’’;
while($linea ne ‘‘FIN\n’’){
print ‘‘$linea’’;
$linea = <STDIN>;
}

$var = "";
while{$var ne "FIN"){
    $var = <STDIN>;
}
```

De momento no nos preocuparemos por la lectura de teclado, la veremos más adelante.
Como de costumbre, habría una manera de hacer un bucle for con un while; hagamos uno que haga lo mismo que el del ejemplo anterior:

```
$i = 0;
while($i <= 10){
    print ‘‘$i \n’’;
    $i++;
}
```

Capítulo 7

Lectura de teclado y salida por pantalla

En muchos scripts y otros pequeños programas que queramos hacer necesitaremos recibir información del usuario, así como mostrarle los resultados de las operaciones realizadas, es por ello que necesitaremos poder leer del teclado y mostrar datos por la pantalla, de ello se encargan la función print y el manejador de ficheros <> (recordemos que en linux casi todo se puede leer como si fuera un fichero, y el teclado no iba a ser una excepción).
Veámoslo un poco más en detalle:

7.1. La función *print*

La función print es la responsable de imprimir por pantalla las variables que nosotros le indiquemos, incluyendo secuencias de escape como \n, de hecho sólo imprime lo que nosotros le indiquemos, es decir, que si no especificamos que imprima el carácter de nueva línea no lo hará, y nuestro programa podría ser algo así:

```
print ‘‘Hallo Welt’’;
```

dándonos el siguiente resultado:

```
rcortega@cursos> ./print.pl
Hallo Weltrcortega@cursos>
```

Para evitarlo simplemente debemos poner la secuencia de escape \n cuando queramos cambiar de línea.

7.2. El manejador de ficheros <>

Para leer de teclado se usa un manejador de ficheros preparado para leer de la entrada estándar (stdin), de modo que sería algo así:

```
$var = <STDIN>;
```

Como puede observarse lo que hacemos es recoger lo que entra por el teclado en una variable. Este método es bloqueante, es decir, una vez invocado, detiene la ejecución del programa hasta que finaliza su tarea, y esto sucede cuando el usuario pulsa ENTER. Cabe destacar que en la variable queda guardado también el carácter de nueva línea enviado al pulsar enter, por lo que por ejemplo el siguiente código:

```
$vart = ‘‘’’;
while($var ne ‘‘FIN’’){
    $var = <STDIN>;
}
```

sería un bucle sin fin, ya que lo leído siempre tendrá un \n al final, y por tanto nunca será exactamente igual a `FIN`.
Para solucionar esto tenemos la función chomp(), que nos permite eliminar el carácter de nueva línea del final de una variable (si es que lo tiene), por lo que para que funcionara deberíamos hacer algo así:

```
$var = "";
while{$var ne "FIN"){
    $var = <STDIN>;
    chomp($var);
}
```

y ahora sí terminaría de hacer repeticiones cuando tecleasemos `FIN`*.*

Capítulo 8

Trabajando con ficheros

En el apartado anterior hemos visto cómo emplear el manejador de ficheros para leer de la entrada estándar, pero también es útil para trabajar con ficheros; la única diferencia es que a nuestros ficheros deberemos asignarle un manejador con un nombre distinto de STDIN. Para trabajar con un fichero debemos seguir tres fases:

8.1. Abriendo un fichero

A la hora de abrir un fichero, debemos decidir para qué vamos a abrirlo: si queremos leerlo, escribirlo, o anexar al final de un fichero existente, según qué sea lo que queremos hacer, será distinta la forma de abrirlo; veámoslo:

8.1.1. Abriendo un fichero para lectura

Para abrir un fichero en modo de sólo lectura basta con asociar al manejador el nombre del fichero a abrir, sin más, y eso se hace con la funcion open():

```
open(FILE,"fichero.txt");
```

Con esto lo que hemos conseguido es asociar al fichero llamado `fichero.txt` *un filehandle llamado* `FILE`*, de modo que a partir de este momento nos podremos referir al fichero a través del manejador.*

8.1.2. Abriendo un fichero para escritura

Cuando lo que queremos es abrir un fichero para escribirlo, el procedimiento es idéntico al de abrirlo para lectura con la salvedad de que deberemos anteponer el carácter ">" al nombre del fichero, de esta manera:

```
open(FILE,">fichero.txt");
```

Debemos recordar que si el fichero existe, abriéndolo de esta manera cuando escribamos en él lo que haremos será sobreescribirlo, de modo que perderemos los datos que tuviéramos en él.

8.1.3. Abriendo un fichero para anexar datos al final

Si no queremos perder los datos que teníamos en el fichero, sino que queremos escribir a continuación de lo que había, el procedimiento es el mismo con la salvedad de que esta vez deberemos anteponer dos ">":

```
open(FILE,">>fichero.txt");
```

8.2. Leyendo un fichero

A la hora de leer el fichero podemos querer hacerlo línea a línea, lo que se hace de la misma manera que cuando leíamos de teclado; supongamos que hemos abierto un fichero asignándole el filehandle `FILE`*, pues con la línea:*

```
$linea = <FILE>;
```

estaríamos recogiendo una línea del fichero, de tal modo que cada vez que lo hagamos cogeremos la siguiente, de forma secuencial, lo cual nos permitiría usar el siguiente código:

```
while{<FILE>){
    print "La línea $. es $_";
}
```

para leer el fichero entero.
La variable `$_` *contiene la línea que acabamos de leer, y la variable* `$.` *el número de dicha linea.*
Cuando lleguemos al final del fichero la llamada a `FILE` *nos devolverá NULL, con lo que saldremos del bucle.*

Otra forma de leer el fichero es guardarlo entero en un array, de modo que cada posición del array sea una línea del fichero; esto se haría de la siguiente manera:

```
@datos = <FILE>;
```

Aunque esta es la forma más rápida, hay que tener en cuenta que si el fichero es grande esta operación nos consumirá mucha memoria.
Como en el caso de la lectura de teclado, cada línea contiene un carácter de final de línea, que también es trasladado a la variable con la que estamos leyendo el fichero.

8.3. Escribiendo un fichero

Si queremos escribir en un fichero debemos usar la función print() especificando en qué manejador de ficheros queremos escribir (por defecto usará STDOUT); por ejemplo, la línea:

```
print FILE ‘‘probando...\n’’;
```

escribirá la línea `‘‘probando...’’` *en el fichero representado por el filehandle* `FILE`*, incluyendo un carácter de fin de línea.*

8.4. Cerrando un fichero

Aunque al finalizar la ejecución de nuestro programa, el intérprete de perl cerrará de forma automática todos los manejadores de ficheros que queden abiertos, suele ser recomendable cerrarlos manualmente en el momento en que dejemos de necesitarlos, en favor de una programación más ordenada y cuidadosa.
Para cerrarlo tan sólo deberemos incluir la función close() en una línea como ésta:

```
close(FILE);
```

8.5. Trabajando con directorios

Para trabajar con ficheros a menudo deberemos cambiar de directorio, o listar los ficheros que contiene. En perl esto puede hacerse con funciones:

8.5.1. La función *chdir*

La función chdir equivale al cd de toda la vida, cambiando el directorio de trabajo actual al indicado como parámetro. Por supuesto también puede fallar, por lo que quizás la mejor manera de emplearlo sea así:

```
$dir = ‘‘/home/rcortega/’’;
chdir $dir or die $!;
```

8.5.2. Leyendo directorios

Para leer un directorio, al igual que pasa con los ficheros, hay que seguir tres pasos:

Abrir un directorio

La operación de abrir un directorio equivale a abrir un fichero para lectura, y se hace con la función opendir:

```
opendir(DIRECTORIO, $dir);
```

Leer un directorio

Si quisiéramos, por ejemplo, mostrar por pantalla los ficheros que contiene un directorio, deberíamos usar la función readdir, y podríamos hacerlo de la siguiente manera (suponiendo que ya lo hemos abierto como hemos visto en el apartado anterior):

```
while($file = readdir(DIRECTORIO)){
    print ‘‘$file\n’’;
}
```

Cerrar un directorio

Por último, y al igual que pasa con los ficheros, conviene cerrarlo, aunque sepamos que al terminar la ejecución el intérprete lo cerrará. Pensemos que si nuestro programa es bastante largo, o se producen muchas lecturas de directorios/ficheros, podemos tener problemas de memoria o demasiados ficheros abiertos.
Para cerrar un directorio se hace con la función closedir:

```
closedir(DIRECTORIO)
```

8.6. Obteniendo información de un fichero

Para conseguir información sobre un fichero, tenemos los test, que nos dan datos como si está vacío, qué tipo de fichero es, etc.
Podemos ver la lista completa en la siguiente tabla:

Test	*Descripción*
`-r`	Comprueba si tenemos permiso de lectura (UID/GID efectivo).
`-w`	Comprueba si tenemos permiso de escritura (UID/GID efectivo).
`-x`	Comprueba si tenemos permiso de ejecución (UID/GID efectivo).
`-R`	Comprueba si tenemos permiso de lectura (UID/GID real).
`-W`	Comprueba si tenemos permiso de escritura (UID/GID real).
`-X`	Comprueba si tenemos permiso de ejecución (UID/GID real).
`-o`	Comprueba si el UID efectivo es el propietario del fichero.
`-O`	Comprueba si el UID real es el propietario del fichero.
`-e`	Comprueba si el fichero existe.
`-z`	Comprueba si el fichero está vacío.
`-s`	Devuelve el tamaño del fichero.
`-f`	Comprueba si el fichero es un fichero plano.
`-d`	Comprueba si es un directorio.
`-l`	Comprueba si es un enlace simbólico.
`-S`	Comprueba si es un Socket.
`-p`	Comprueba si es un pipe (FIFO).
`-b`	Comprueba si es un dispositivo de bloques (HD).
`-c`	Comprueba si es un dispositivo de carácter (consola).
`-u`	Comprueba si tiene el SetUID activo.
`-g`	Comprueba si tiene el SetGID activo.
`-k`	Comprueba si tiene el *Sticky bit* activo.
`-t`	Comprueba si la entrada estándar (*STDIN*) es interactiva.
`-T`	Comprueba si el fichero es de texto.
`-B`	Comprueba si el fichero es binario.
`-M`	Devuelve el número de días desde la última modificación.
`-A`	Devuelve el número de días desde el último acceso.
`-C`	Devuelve el número de días desde la última modificación del inodo.

Cuadro 8.1: **Test para ficheros/directorios**

Capítulo 9

Excepciones

Hay ocasiones a lo largo de la ejecución de un programa en las que una llamada a una determinada función puede fallar por una determinada razón, entonces decimos que se ha producido una excepción. Algunas excepciones apenas afectan al curso de nuestro programa, pero otras veces será más que aconsejable notificar tal circunstancia al usuario, eso por no hablar de su utilidad a la hora de realizar pruebas o debug. Si queremos saber si una determinada función se ha ejecutado con éxito podemos usar una estructura como ésta:

```
<llamada a funcion> or <mensaje de error>
```

Por ejemplo:

```
open(FILE, ‘‘fichero.txt’’)or print ‘‘No se pudo abrir el fichero\n’’;
```

nos permite notificar el fallo al intentar abrir el fichero. En ciertas ocasiones nos interesará interrumpir en ese momento la ejecución de nuestro programa, entonces podemos usar la función die, que nos permite imprimir un mensaje en pantalla y abortar la ejecución:

```
open(FILE, ‘‘fichero.txt’’)or die ‘‘No se pudo abrir el fichero\n’’;
```

Aunque podemos ser un poco más exigentes y mostrar por pantalla el mensaje de error generado automáticamente por el intérprete y que queda guardado en la variable $!:

```
open(FILE, ‘‘fichero.txt’’)or die $!;
```

Capítulo 10

Expresiones regulares

Las expresiones regulares son, posiblemente, la capacidad estrella de perl, aquello que realmente le distingue de otros lenguajes de programación de su tipo. Se trata de patrones de búsqueda expresados como secuencias de caracteres, que nos permiten determinar si una cierta expresión está dentro de una variable que estamos analizando, o bien extraer fragmentos de esa variable que respondan al patrón especificado.
Su uso sigue la siguiente sintaxis:

```
<variable> =~ /<expresion regular>/;
```

Por ejemplo:

```
$var =~ /porcion/;
```

busca si la palabra `porcion` *se encuentra dentro de la variable* `$var`*. Esto tal cual no es muy útil, pero puede serlo dentro de una sentencia condicional, ya que si la variable en cuestión cumple el patrón definido por la expresión regular, esta operación devolverá verdadero y si no devolverá falso.*
Para hacer un ejemplo, supongamos que hoy es el santo de todas las que se llamen Maria, y queremos felicitarlas, pero queremos que se felicite a todas aquellas mujeres cuyo nombre contenga la palabra "Maria", aunque su nombre sea compuesto, entonces podríamos escribir el siguiente programa:

```
print ‘‘Introduzca su nombre:’’;
$nombre = <STDIN>;

if($nombre =~ /Maria/){
    print ‘‘Felicidades!!! Hoy es tu santo!!\n’’;
}
```

Pero puede que lo que queramos sea que esté al principio del nombre, como por ejemplo en "Maria Jesus", entonces debemos poner:

```
print ‘‘Introduzca su nombre:’’;
$nombre = <STDIN>;

if($nombre =~ /^Maria/){
    print ‘‘Felicidades!!! Hoy es tu santo!!\n’’;
}
```

El "^índica que la variable debe contener la palabra `Maria` *al principio del todo.*
También podría ser que buscásemos un nombre en el que Maria sea la última parte; en ese caso deberíamos poner:

```
print "Introduzca su nombre:";
$nombre = <STDIN>;
chomp($nombre);

if($nombre =~ /Maria$/){
    print "Felicidades!!! Hoy es tu santo!!\n";
}
```

El “$” al final de la expresión regular indica que queremos que esa sea la parte final de la variable, o si no consideraremos que no cumple el patrón. Cabe destacar también que en esta ocasión hemos usado la funcion chomp() para eliminar el carácter de nueva línea que queda al leer de teclado, porque si no nunca se cumpliría la expresión regular.

Nótese que si pusiéramos:

```
print "Introduzca su nombre:";
$nombre = <STDIN>;
chomp($nombre);

if($nombre =~ /^Maria$/){
    print "Felicidades!!! Hoy es tu santo!!\n";
}
```

el resultado sería equivalente a:

```
print "Introduzca su nombre:";
$nombre = <STDIN>;
chomp($nombre);

if($nombre eq "Maria"){
    print "Felicidades!!! Hoy es tu santo!!\n";
}
```

Pero en ocasiones puede que queramos buscar patrones en los que desconozcamos ciertas partes, y para eso debemos emplear ciertas secuencias de escape que nos permitan sustituir a esas partes indeterminadas. Podemo ver esas secuencias en la siguiente tabla:

Secuencia	*significado*
\d	un dígito
\D	cualquier cosa que no sea dígito
\w	cualquier carácter alfanumérico y -
\W	cualquier carácter no alfanumérico
\s	cualquier carácter blanco (espacio, tabulador, etc)
\S	cualquier carácter no blanco
[a|b|c]	cualquier línea que contenga “a” “b” o “c”
.	cualquier carácter salvo el de nueva línea

Cuadro 10.1: **Secuencias de escape de sustitución**

También puede ser que deseemos que una cierta secuencia se repita un determinado número de veces, eso se puede hacer usando ciertos complementos:

Complemento	*significado*
?	debe suceder 0 o 1 vez
*	debe suceder 0 o más veces
+	debe suceder 1 o más veces
{n}	debe ocurrir *n* veces
{n,}	debe ocurrir *n* o más veces
{n,m}	debe ocurrir entre *n* y *m* veces

Cuadro 10.2: **Complementos a las secuencias de escape**

Con esto podemos definir expresiones regulares más complejas, por ejemplo, si queremos comprobar si una cierta variable es una cadena compuesta por tres palabras separadas por dos espacios unas de otras, podríamos hacer:

```
if($var =~ /^\w+  \w+  \w+$/){
    print "Cumple la condición\n";
}
```

Pero supongamos que no sólo pueden ser espacios, sino que podrían ser tabuladores, pero también han de ser dos:

```
if($var =~ /^\w+\s{2}\w+\s{2}\w+$/){
    print "Cumple la condición\n";
}
```

Ahora supongamos que quisiéramos saber cuál es la palabra que hay en segunda posición; para almacenarla en una variable lo que debemos hacer es rodear la secuencia de escape que la representa entre paréntesis, y de este modo, si se cumple la expresión regular, tendremos dicha palabra guardada en una variable que perl genera automáticamente llamada $1:

```
if($var =~ /^\w+\s{2}(\w+)\s{2}\w+$/){
    print "La segunda palabra es $1\n";
}
```

Pero puede también que quisiéramos almacenar las tres, entonces debemos seguir la misma operación, ponerlas entre paréntesis, y perl les irá asignando a variables análogas a la anterior, y que serán numeradas por orden de aparición:

```
if($var =~ /^(\w+)\s{2}(\w+)\s{2}(\w+)$/){
    print "La primera palabra es $1\n";
    print "La segunda palabra es $2\n";
    print "La tercera palabra es $3\n";
}
```

Los más observadores se habrán percatado de que hay ciertos caracteres que tienen significados especiales dentro de las expresiones regulares, entonces surge la pregunta:

"¿Qué pasa entonces si quiero buscar el carácter '$'(por ejemplo) como parte del patrón?"

La respuesta es bien sencilla: habría que escaparlo con la barra invertida: \$. Y al igual que con el carácter dólar habría que hacerlo con los siguientes:

```
$ @ . + ? * ( ) { } [ ] | \
```

10.1. Sustituyendo en cadenas de caracteres

Hay algunas funciones que nos permiten hacer sustituciones en cadenas de caracteres, cambiando porciones de la cadena por otra cadena que le especifiquemos, o bien carácter a carácter. Estas funciones son s/// y tr///:

10.1.1. La función *s///*

La función s nos permite sustituir subcadenas dentro de una cadena, siguiendo la sintaxis siguiente:
`<variable>= s/<patron>/<sustitución>/`
Por ejemplo, el siguiente código:

```
$cadena = "Esta es mi cadena\n";
$cadena =~ s/es/era/;
print $cadena;
```

daría el siguiente resultado:

```
rcortega@cursos> ./sustitute.pl
Esta era mi cadena
```

Suponiendo que la subcadena apareciera varias veces, la función s sustituye sólo la primera, y puede que esto no sea lo que deseamos, o bien puede que no sepamos con exactitud si lo que buscamos aparecerá en mayúsculas o minúsculas, entonces es cuando empleamos dos modificadores que tiene la función:

Modificador	*Efecto*
g	sustituye tantas ocurrencias como encuentre
i	al buscar el patrón ignora las diferencias mayúsculas/minúsculas

Cuadro 10.3: **Modificadores de la función** *s///*

10.1.2. La función *tr///*

La función tr sustituye caracteres sueltos uno a uno, pero lo hace todo en una sola "pasada", y esto, aunque parezca trivial, no lo es tanto, supongamos por ejemplo que deseamos aplicar el complemento a uno a una secuencia binaria, cambiando los 1's por 0's y los 0's por 1's, podríamos pensar que se puede hacer de la siguiente manera:

```
$cadena = "01100011011110";
$cadena =~ s/1/0/g;
$cadena =~ s/0/1/g;
print $cadena;
```

Pero se nos escapaba que tras la primera sustitución, toda nuestra secuencia son 0's, por lo que al hacer la segunda sustitución nos quedará una secuencia sólo de 1's. Es en estas situaciones donde la función tr/// resulta útil:

```
$cadena = "01100011011110";
$cadena =~ tr/01/10/;
print $cadena;
```

Lo que hace la función tr es sustituir cada ocurrencia del primer carácter del patrón por el primer carácter de la sustitución, y así con todos.
La función tr también acepta ciertas abreviaturas para facilitar sustituciones largas; supongamos por ejemplo que queremos pasar una cadena de minúsculas a mayúsculas:

```
$cadena = "esta es mi cadena\n";
$cadena =~ tr/a-z/A-Z/;
print $cadena;
```

El resultado sería:

```
rcortega@cursos> ./translate.pl
ESTA ES MI CADENA
```

Al igual que la función s///, la función tr/// tiene modificadores:

- ***s*** *si al traducir hay varios caracteres seguidos iguales borra todos menos uno.*
- ***c*** *sustituye todos los caracteres que no se encuentran en el patrón.*
- ***d*** *si el patrón es más largo que la sustitución, los caracteres del patrón que no tienen correspondencia en la sustitución son borrados.*

Capítulo 11

Llamando al sistema

Desde perl es posible hacer llamadas a comandos del sistema y de ser necesario recoger su salida. Como de costumbre hay varias maneras de hacerlo:

11.1. La función *system()*

La función system nos permite invocar a un comando del sistema, y nos deja en la variable $? el resultado, que dividido entre 256 nos dirá si el comando tuvo éxito o no:

Resultado (`$?/256`)	*Significado*
1	El comando fracasó en su ejecución
0	El comando se ejecutó con éxito

Cuadro 11.1: **Resultado de** *system*

Un ejemplo de uso sería:

```
system{"chown root:root fichero.txt");
```

Debemos tener en cuenta que si el comando tiene alguna salida por pantalla, ésta nos será mostrada al ejecutar nuestro programa (por ejemplo si hacemos un ls).

11.2. La función *qx()*

La función qx tiene la característica de que nos devuelve la salida del comando, permitiéndonos guardarla en una variable:

```
$hora = qx{"date");
print "$hora\n";
```

Nótese que ahora no saldrá la salida del comando por pantalla hasta que nosotros hagamos el print (de no ser así, en este caso saldría dos veces).

11.3. Comunicación con procesos

En ocasiones necesitaremos recoger la salida de un comando que sea demasiado lento, o que genere una salida demasiado extensa, o bien querremos pasarle algo a un comando a través de su entrada estándar, entonces deberemos usar unas estructuras similares a los filehandles que empleábamos con los ficheros, añadiendoles una tubería. Supongamos que tenemos un comando llamado comando que produce una salida de varias líneas, entonces podríamos hacer así:

```
open(COMANDO ,"comando |");
while(<COMANDO>){
    print $_;
}
close COMANDO;
```

El pipe al final del comando indica que deseamos obtener su salida estándar. Sin embargo, supongamos que lo que queremos es pasar algo al comando que vamos ainvocar a través de su entrada estándar, por ejemplo, si pretendemos usar el comando sendmail para enviar un mail cuyo cuerpo sea el contenido de un fichero llamado `fichero.txt`*:*

```
open(COMANDO ,"| /usr/sbin/sendmaili rcortega@di.uc3m.es");
open(FILE, "fichero.txt");
while(<FILE>){
    print COMANDO $_;
}
close FILE;
close COMANDO;
```

Capítulo 12

Estructuración de código: funciones

Hasta ahora hemos usado varias funciones definidas en las librerías de perl, pero nuestro programa siempre ha sido muy pequeño y ha bastado con declararlo de forma secuencial y sin necesidad de programarnos métodos. Sin embargo, puede que queramos hacer un programa algo más complejo, y para que resulte más legible y se pueda trabajar con el necesitaríamos estructurarlo de modo que esté formado por varias funciones, a las que llamaremos cuando sea necesario. Pero vayamos por partes:

12.1. Declarando la función

La declaración de la función debe comenzar por la palabra reservada `sub`*, seguida de un espacio y el nombre de la función. Después pueden (no es obligatorio) ir tantos caracteres* `$` *como parámetros deba recibir la función encerrados entre paréntesis y después las llaves que encerrarán el código de nuestra función, algo así:*

```
sub hacerFuncion ($$){
    ....
}
```

Acabamos de declarar una función llamada `hacerFuncion` *que recibe dos parámetros.*
Nótese que como perl no hace distinción de tipos de variables no se declara el tipo de dato que devuelve la función.

12.2. Recogiendo los parámetros

Los parámetros se nos dan dentro de la variable `@_` *que, como a estas alturas debería ser ya inmediato deducir, es un array. Para separarlos con mayor facilidad podemos hacerlo de esta manera:*

```
($param1,$param2) = @_;
```

12.3. Devolviendo el resultado

La devolución del resultado de la función se hace como en todos los demás lenguajes de programación, con la palabra reservada `return`*:*

```
return $resultado;
```

Veámoslo ahora todo junto en un método:

```
sub sumar($$){
    ($sumando1,$sumando2) = @_;
    $resultado = $sumando1 + $sumando2;
    return $resultado;
}
```

12.4. Invocando funciones

Cuando una función está declarada dentro del mismo fichero donde se la invoca, se debe hacer mediante el carácter `&` *seguido del nombre de la función y los parámetros separados por comas, y encerrados entre paréntesis, con un punto y coma al final:*

```
$suma = &sumar(2,2);
```

Capítulo 13

Bibliografía y referencias

13.1. “Curso de Perl” *Rafael Calzada Pradas*

13.2. “Teach Yourself Perl 5 in 21 days” *David Till*

Un curso de lo más completo, disponible en http://docs.rinet.ru/P7/

13.3. “Comprehensive Perl Archive Network”

La colección más completa de módulos e información sobre Perl, disponible en http://www.cpan.org

13.4. Otros recursos web

13.4.1. http://www.perl.com

Página de O'Really dedicada al mundo de Perl.

13.4.2. http://www.perl.org

Página bastante completa con información sobre perl, tutoriales, etc.