Perl 5.0

Un lenguaje multiuso

$Autor = 'José Miguel Prellezo Gutiérrez';

$Mail = 'jprellezo@eresmas.com';

$Rev = 1.6;

$Fecha = "Octubre, 2002";

Tabla de Contenidos

I. INTRODUCCIÓN

Historia

Perl (Practical Extraction y Report Language) es un lenguaje de programación que se creó originalmente para extraer informes de ficheros de texto y utilizar dicha información para preparar informes. Actualmente ha evolucionado de forma que es posible realizar labores de administración en cualquier sistema operativo. Debe gran parte de su popularidad a tratarse de un intérprete que se distribuye de forma gratuita. Un script genérico de Perl puede ejecutarse en cualquier plataforma en la que tengamos un intérprete disponible.

Con el crecimiento del WWW se vio que era necesario realizar programas CGI y Perl se convirtió en la elección natural para los que ya estaban familiarizados con este lenguaje. El aumento de sitios Web ha transformado el papel de Perl de un lenguaje de Script oscuro y desconocido a la herramienta principal de programación CGI.

Instalación

Dependiendo del sistema operativo que se utilice, habrá que utilizar una distribución de Perl u otra. La principal referencia figura en http://www.perl.com. No obstante, en http://www.cpan.org podemos encontrar más distribuciones, disponiendo de al menos una para cada plataforma.

Este documento se centra en la utilización de Perl desde los sistemas operativos de la serie Microsoft Windows igual o superior a la 95. Una excelente fuente de recursos para Perl sobre Windows la podemos encontrar en http://www.activestate.com

Es conveniente utilizar como directorio base de la instalación C:\Perl, y añadir al PATH la ruta C:\PERL\BIN. El típico programa "Hola, mundo" en Perl se realiza poniendo en un fichero (supongamos "hola.pl") las siguientes instrucciones:

```
#!c:/perl/perl
print "Hola mundo\n";
```

Para ejecutar basta con escribir, desde una ventana de MS-DOS:

```
perl hola.pl
```

Peculiariadades

- Perl es un lenguaje case-sensitive.
- Para editar el código fuente necesitamos simplemente un editor de texto. El Notepad o cualquier otro con el que estemos familiarizados puede valer.

- Se ejecuta desde la línea de comandos de una ventana del sistema operativo.
- Los comentarios comienzan con el carácter **#**
- Las instrucciones terminan en punto y coma.
- La función **print** sirve para mostrar información por pantalla, y admite formatos muy diversos aunque sencillos de comprender. En Perl hay mucha flexibilidad para escribir los argumentos:

```
print("Un texto", "Otro texto");       # con paréntesis
print "Un texto", "Otro texto";        # sin parentesis
```

- Perl ofrece una ayuda en línea desde la consola de comandos. Por ejemplo, para obtener ayuda sobre la función print, escribiremos en una ventana MSDOS:

```
perldoc -f print
```

II. TIPOS DE DATOS

Por defecto, no es necesario declarar las variables previamente a su uso. Las variables se pueden empezar a usar directamente en las expresiones. Existen tres tipos básicos de variables, que son:

Escalar ($)

Las variables escalares empiezan por el carácter **$**. Ejemplos:

```
$a = 5;
$b = "xxx";
$c = $a++;   # $a++ es como en C, o sea, $a + 1
```

Un escalar puede almacenar la siguiente información:

- Números
- Strings
- Referencias a otras variables
- Descriptores de ficheros

Array (@)

Las variables array empiezan por el carácter **@**, y sirven para agrupar un conjunto de variables de tipo escalar.

```
@a = (95, 7, 'fff' );
print $a[2];      # imprime el tercer elemento: fff
print @a;         # imprime: 957fff (todo junto)
```

Sobre las matrices debes advertir que:

- Cada uno de los elementos del array son variables de tipo escalar.
- Los subíndices de la matriz empiezan por 0 (como en el lenguaje C).
- Tenemos dos formas de conocer el número de elementos del array:

```
@a = (7,8,9,10);
$a = @a;              # $a vale 4
$a = scalar(@a);      # $a vale 4
```

- La variable **$a** no tiene que ver nada con **$a[0]**.
- Es posible inicializar un array con una sintaxis especial:

```
$a = (2..7);      #  $a queda con (2,3,4,5,6,7);
$a = ('a'..'e'); #  $a queda con ('a','b','c','d','e')
```

- Los subíndices positivos acceden a los elementos por porden creciente y los negativos por orden inverso

```
@a = ('a'..'e');      # $a[0]  es 'a'   $a[4]  es 'e'
@a = ('a'..'e');      # $a[-1] es 'e'   $a[-5] es 'a'
```

- Se puede obtener una parte de un array (subarray):

```
@a = ('a'..'e');
@b = @a[3, 0];    # @b = ('d', 'a');
@c = @a[2-5];     # es lo mismo que @a[2,3,4,5] ó
                  # @a[2..5]
```

- Es posible colocar una lista de variables escalares a la izquierda del igual

```
($a, $b) = @x     # $a queda con el valor de $x[0]
                  # $b queda con el valor de $x[1]
```

- Para recorrer un array utilizamos el bucle ***foreach***

```
foreach $valor(@elArray) {
    # Este bucle recorre el array, y en cada iteración
    # deja en $valor el contenido de la celda del array
    # que se está visitando.
}
```

Hash (%)

Las variables tipo *hash* o array asociativo empiezan por el carácter **%**. Se trata de un tipo característico de Perl, y consiste básicamente en un array en el cual se accede a sus distintos elementos a través de una clave en lugar de por un índice.

Para crear un elemento de un hash se requiere una lista de dos valores, siendo el primer elemento es la clave y el segundo es el valor asociado a dicha clave

```
%almacen = ( 'Peras', 5, 'Manzanas', 3);
print $almacen{'Peras'};              # Imprime: 5
print $almacen{'Manzanas'};           # imprime: 3
```

Si la clave es un string sencillo (no compuesto de palabras separadas con espacios en blanco) se pueden omitir las comillas. Por tanto, son equivalentes las instrucciones:

```
print $almacen{'Manzanas'};
print $almacen{Manzanas};
```

Existe otra sintaxis a la hora de inicializar un array asociativo que suele utilizarse habitualmente por ser más descriptiva que la anterior:

```
%almacen = (Peras=>3, Manzanas=>5);
```

En cualquier momento se puede agregar un elemento a un hash. Si no existe, se crea y se almacena el valor asignado:

```
$almacen{Naranjas} = 9;
```

En un hash los elementos se accesan por claves y no se permiten claves duplicadas.

FUNCIONES PERL PARA EL MANEJO DE ARRAYS ASOCIATIVOS

- La función **delete** sirve para borrar un elemento

```
delete $almacen{Manzanas};
```

- La función **keys** crea un array con las claves de un hash

```
@b = keys %almacen       # @b queda con ('Peras',
                                        'Manzanas',
                                      'Naranjas');
```

- La función **values** devuelve un array con los valores del hash

```
@v = values %almacen      #  @v queda con  ( 5, 3, 9 );
```

- La función **exists** prueba si existe la clave en el hash

```
$b = exists $almacen{Peras};     #  $b queda con 1
$c = exists $almacen{Tomates};   #  $c queda con ""
```

- Para recorrer un hash, utilizaremos **foreach**. Ejemplo:

```
foreach $k( keys %almacen )
{
    print "key=$k val=$almacen{$k} \n";
}
```

III. OPERADORES Y CONTROL DEL FLUJO

Los valores son los datos que se almacenan en las variables.

Números

- Independientemente del tipo numérico (entero o real), Perl trata a ambos de la misma forma, es decir como si los números enteros fuesen de tipo punto flotante, por lo que en la mayor parte de las máquinas tendremos una precisión de 16 dígitos en aritmética entera.
- Se puede expresar un número en base octal, precediéndolo del carácter "0".
- Se puede expresar un número en base hezadecimal, precediéndolo de los caracteres "0x".

```
$a = 010;              # $a tiene el valor 8 decimal
$a = 0x10;             # $a tiene el valor 16 decimal
```

Strings

Los strings pueden escribirse con comillas dobles ("), simples (') o invertidas (`).

- Cuando se usa comilla simple, la variable escalar toma el valor del string indicado directamente, sin hacer ninguna operación adicional.

- Cuando se usa doble comilla se pueden **interpolar** variables escalares y arrays; interpolar significa intercalar: en el resultado final la variable se sustituye por su valor. Por ejemplo:

```
$a = 'pueblo'
print "hola $a";      # imprime: hola pueblo
print 'hola $a';      # imprime: hola $a
```

Cuando se requiere interpolar una variable entre letras, hay que utilizar las llaves {}, p.e. como se coloca $a antes de una letra como "s"

```
"abc$as";    # Interpola la variable $as (no existe)
"abc${a}s"   # Interpola correctamente $a
```

Entre las comillas dobles se pueden tambien poner caracteres especiales como salto de línea (\n), tabulador (\t), backspace (\b), etc. Ejemplos:

```
print  "aaa\n";       # Salta linea después de imprimir
print  'aaa\n';       # No salta, imprime: aaa\n
```

Entre las comillas dobles, cuando se quiere utilizar algún carácter que tienen significado especial en Perl, (como $, ", \, etc.), hay que precederle por el carácter de escape (\).

```
$a = 1234;
print  "valor=$a\$";    # imprime: valor=1234$
```

Cuando se interpola un array, Perl separa los elementos del array por un espacio en blanco

```
@a = (95, 7, "fff" );
print "val=@a" ; # imprime: val=95 7 fff
```

El separador es realmente el valor de la variable escalar definida por Perl ($") y puede ser reasignada, por ejemplo:

```
$" = ',';
print "val=@a";   # imprime: val=95,7,fff
```

- Cuando se asigna un string encerrado con comillas invertidas, significa que dicho String es en realidad un comando del sistema operativo que debe ejecutarse, y cuya salida se almacenará en la variable escalar.

```
$a = `dir *.exe`; # $a queda con la salida del comando
                  # "dir *.exe"
```

En comillas invertidas se pueden interpolar variables.

Verdadero y falso

Como en C, cualquier expresión tiene un significado lógico; p.e. las asignaciones tienen el valor de lo asignado.

Sin embargo, no existe un tipo de datos booleano como en otros lenguajes. En su lugar, será considerado como verdadero todo aquello que no es falso, y los valores que se consideran como falsos son:

1. Los strings "", "0" y el número 0.
2. El valor "no definido" de una variable, esto es, cuando existe la variable pero no tiene un valor asignado. La función **defined** se usa para averiguar si una variable esta definida:

```
$a = 5;
print "a definida" if( defined $a );
print "b no definida"  if( !defined $b );
```

NOTA: La función **undef** no es lo contrario de **defined**. Lo que hace realmente es devolver el argumento "no definido", elimina el **bind** entre la variable y el valor, el cual desaparece si su contador de usuarios queda en cero.

Expresiones lógicas

Una expresión lógica es una expresión cuyo valor es verdadero o falso. Usualmente las expresiones lógicas se usan en condiciones.

```
if( $a and $b ) {
    print  "A y B son verdaderos";
}
if( $a  or  $b ) {
    print  "A o B son verdaderos";
}
```

Existen también los operadores && y || como en C, que tienen más prioridad que and y or. Las expresiones logicas tiene otro uso muy interesante usándolas como una instruccion en sí misma:

```
$a  and  $b;
# si $a es falso toda la expresión anterior es falsa
# por lo tanto no se evalúa el elemento $b. Si $a es
# verdadero se tiene que evaluar $b para conocer el
# valor de la expresión. Y eso aunque el valor de
# la expresión total no se utiliza para nada.

$a and print "A es verdadero";
# el print solo se hace si $a es verdadero; es
# equivalente a: print "A es verdadero" if $a;

$a or print "A es falso";
# el print solo se hace si $a es falso; equivale a
# print "A es falso"  if(! $a);
# o también a:
# print "A es falso"  unless($a);
```

Operadores

OPERADORES LÓGICOS

- Operadores para comparar números (como en C)

```
$a == $b  and  print "A igual a B";
$a != $b  and  print "A distinto de B";
$a >= $b  and  print "A >= B";
```

- Operadores para comparar strings:

```
$a eq $b  and  print "A igual a B";
$a ne $b  and  print "A distinto de B";
$a ge $b  and  print "A >= B";
```

El siguiente ejemplo muestra por qué se necesita distinguir una comparación numérica de una comparación de strings:

```
$a=5;
$b=49;
$x = ($a  gt $b)      #  $x queda 1    ( verdadero )
$x = ($a  >  $b)      #  $x queda ""   ( falso )
```

- Comparación de números (*starship operator*)

```
$x = $a <=> $b
#  $x queda con -1  si $a <  $b
#  $x queda con  0  si $a == $b
#  $x queda con  1  si $a >  $b
```

- Comparación de strings

```
$x = $a cmp $b
# $x queda con  -1  si $a lt $b
# $x queda con   0  si $a eq $b
# $x queda con   1  si $a gt $b
```

OPERADOR TERNIARIO

Es una abreviatura de las 3 partes de la estructura if:

```
@a > 5 ?  print "a > 5":  print "a no es > 5";
# Es equivalente a:
#    if( a > 5 ) {
#        print "a > 5";
#    }
#    else {
#        print "a no es > 5";
#    }
```

Operadores de strings

- Repetir strings: con operador "x"

```
$a = "y";
$b = $a x 5;          # $b queda con "yyyyy";
```

- Concatenar strings: con operador "."

```
$a = "abc";
$b = $a . "def";      # $b queda con "abcdef"
```

Operador de rango

".." para generar listas

```
@a = "ay" .. "bb";
```

```
# @a queda con  ("ay", "az", "ba", "bb")
@a = 1..5;
# @a queda con  (1, 2, 3, 4, 5)
```

No deben mezclarse letras y números, ni mayúsculas y minúsculas

Control del flujo

Disponemos de las mismas estructuras del control del flujo que en C, en algunos casos con algún pequeño matiz.

BLOQUE IF

```
if( $condicion1 )  {
   # Entra aquí si la condición es cierta. Es obligatorio
   # poner las llaves aunque sólo exista una instrucción.
}
elsif($condicion2) {
   # Esta parte del if es opcional. Se puede repetir tantas
   # veces como sea necesario.
}
else {
   # Parte final de if por donde entrará a ejecutarse si
   # no lo hizo por ninguna de los bloques anteriores
}
```

BLOQUE IF ABREVIADO

Con ayuda del operador ? podemos escribir en una línea las tres partes de la estructura if:

```
  $maximo = ($x > $y) ?  $x : $y;
# Es equivalente a:
#       if( $x > $y ) {
#               $maximo = $x;
#       }
#       else {
#               $maximo = $y;
#       }
```

BLOQUE UNLESS

Es exactamente lo contrario del if. Por ejemplo

```
unless( $condicion ) {
   # Se puede leer como: "A menos que ..."
}
```

Y sería equivalente a

```
if( ! $condicion ) {
   # ...
}
```

Al final del bloque *unless* se pueden añadir las sentencias *elsif* y un *else* final como en el caso de la construcción *if*, sin embargo no es habitual su uso.

BLOQUE WHILE

Mientras la condición que acompaña al while sea cierta, se ejecuta el bloque de código asociado.

```
while( $condicion ) {
   # Ejecutará estas instrucciones mientras la condición sea cierta
}
```

BLOQUE UNTIL

Es exactamente lo contrario que el bucle *while*. Mientras la condición que acompaña al *until* sea falsa, se ejecuta el bloque de código asociado.

```
until( $condicion ) {
   # Ejecutará estas instrucciones mientras la condición sea falsa
}
```

Equivale a:

```
while( !$condicion ) {
   # ...
}
```

BLOQUE DO

Las dos construcciones anteriores no sirven cuando al menos queremos que se ejecute una vez el código del bloque y chequee la condición al final. Esto es precisamente lo que nos permite el bucle ***do***, que permite dos tipos de construcción:

```
do{
   # Ejecutará estas instrucciones al menos una vez.
   # Mientras la condición sea cierta seguirá iterando
} while( $condicion )
```

Y la complementaria:

```
do{
   # Ejecutará estas instrucciones al menos una vez.
   # Mientras la condición sea falsa seguirá iterando
} until( $condicion )
```

BLOQUE FOR

Sirve para construir bloques en los que en cada iteración se modifica una o varias variables. Se pueden realizar construcciones complejas dentro de un *for*, pero su forma más común es la siguiente:

```
for($i=0; $i<10; $i++) {
   # Ejecuta 10 veces las instrucciones que se pongan aquí.
   # En cada iteración la variable $i se incrementa
}
```

Si queremos un *for* descendente de 2 en 2, podemos hacer:

```
for($i=10; $i>0; $i-=2) {
   # ...
}
```

RUPTURA DE LOS BUCLES

A menudo resulta imprescindible modificar el comportamiento de un bucle bajo determinadas circunstancias, para lo cual tenemos dos palabras reservadas:

- **next**

  Continua ejecutando la siguiente iteración del bucle. Es equivalente al *continue* de C.

```
# Imprime los números pares del 1 al 20.
for( $i =1; $i<=20; $i++ )
{
   next if( $i % 2 );
   #Si el número es par, el if es falso y sigue por aquí.
   print $i, "\n";
}
```

- **last**

  Finaliza la ejecución del bucle, y sigue en la primera instrucción que hay a continuación del mismo. Ejemplo:

```
$i=0;
while(1)
{
   $i++;
   #Cuando llega a 50, se finaliza el bucle al hacer last.
   last if( $i == 50);
   print $i, "\n";
}
```

En numerosas ocasiones tenemos varios bucles anidados y cuando se cumple una determinada condición en el bloque más interno queremos salir de toda la estructura. Podemos actuar de dos formas, que son:

1. Usar la palabra reservada *goto*. Los puristas de la programación estructurada no estarán muy conformes, pero es una posibilidad que tenemos disponible. Por ejemplo:

```
for($i=0;$i<10;$i++) {
   # ...
   for($j=0;$j<10;$j++)   {
      # ...
      for($k=0;$k<10;$k++) {
         print "($i,$j,$k)\n";
         # Queremos salir si el producto de las variables es 12
         goto SALIR if($i*$j*$k==12);
      }
   }
}
   # Para las etiquetas se suelen usar letras mayúsculas.
SALIR:
print "final\n";
```

2. Usar los bloques etiquetados. Esto es, ponemos una etiqueta en el bloque que deseamos romper, y cuando se ejecute el last le indicaremos que bloque es el que se termina. Podemos conseguir el mismo efecto que en el ejemplo anterior de la siguiente forma:

```
SALIR: for($i=0;$i<10;$i++) {
     # ...
    for($j=0;$j<10;$j++)   {
       # ...
       for($k=0;$k<10;$k++) {
          print "($i,$j,$k)\n";
          # Queremos salir si el producto de las variables es 12
          last SALIR if($i*$j*$k==12);
       }
    }
}

print "final\n";
```

SALIDA DEL PROGRAMA

Para finalizar en un punto concreto la ejecución del programa y regresar al sistema operativo, utilizaremos la sentencia:

```
exit $numero_de_error;
```

Como vemos, es posible indicar un número de error devuelto al sistema operativo.

Si además de finalizar, deseamos mostrar un mensaje de error y la línea concreta en que se ha producido, utilizaremos:

```
die “Mensaje de error”;
```

EXCEPCIONES

Al igual que en otros lenguajes, es posible romper el flujo de ejecución mediante unas excepciones que serán tratadas de una forma adecuada. Esto se logra en Perl de una forma un tanto rudimentaria, pero eficaz. Por ejemplo, consideremos el siguiente código:

```
while($a=<STDIN>) {
     chomp $a;
     die "No sirve este valor: $a" unless $a;
     print 100/$a;
}
```

Está esperando a que usuario introduzca valores por el teclado y pulse *Enter*. Si no pone nada o escribe *0*, el programa finaliza con un mensaje de error (instrucción **die**).

Podemos utilizar una construcción diferente:

```
while($a=<STDIN>) {
     eval {
          chomp $a;
          die "No sirve este valor: $a" unless $a;
          print 100/$a;
     };
     if( $@ )
     {
          print "Se ha producido un error: $@";
     }
}
```

En este caso hemos introducido un bloque **eval**. Este bloque simplemente se limita a ejecutar una tras otra las instrucciones que contiene, pero si se ejecuta una sentencia **die**, el resultado no es la salida inmediata del programa, sino del bloque. Aquí entra en juego la variable especial de Perl **$@**, la cual contiene el valor del último error provocado en el bloque **eval**. Si no hubo error, la variable **$@** está indefinida y no entrará por el **if**. Pero si lo hubo, en este caso concreto la variable **$@** contendrá el texto que se pasó como parámetro a la instrucción **die**, y como resultado final se imprimirá el mensaje especificado en el bloque **if**.

Puesto que el **die** no provoca la finalización del programa, en este caso concreto después de imprimir el mensaje de error, se continúa dentro del bloque *while* a la espera del siguiente valor.

IV. FUNCIONES

Definición y uso

En Perl se puede definir una función es cualquier parte, aunque lo común es hacerlo al principio del fichero. La función solo se ejecuta cuando se llama expresamente. La función se define con la palabra **sub**. Ejemplo:

```
sub funcion_1 {
    $a = shift;
            # shift asume el array @_
            # @_ contiene los argumentos
            # que se dan al llamar la función
    $y = 2 * $a;
    return($y);
    # devuelve ese valor al que llamó la función
}
```

La función se llama simplemente escribiendo su nombre[1]:

```
$x = 5;
$z = funcion_1($x);   # pasa $x como único elemento de @_
                      # por tanto, $z queda con 10.
```

Una función que no tiene un ***return*** explícito retorna, no obstante, el valor de la última expresión que se ha ejecutado; por tanto, la función ***funcion_1*** anterior no necesita la sentencia ***return***.

Cuanto se llama a una función, no es obligatorio recoger el valor devuelto por ésta.

Los parámetros de una función se pasan siempre por referencia; por consiguiente, si se modifica $_[1] se está cambiando el segundo parámetro usado en la expresión que llama a la función. Puede ser peligroso si no se maneja con cautela.

```
sub funcion_2 {
    $_[0]=7;
    # Modifica el primer parámetro en el llamador
}
$a = 5;
funcion_2($a);
```

[1] Si la función está definida en un lugar del fichero posterior al sitio desde donde se la llama, es necesario anteponer el símbolo **&**

```
print  $a;        #  imprime:  7
```

Bloques

Un bloque consiste en un conjunto de expresiones dentro de llaves {}. Las funciones son bloques, pero también puede haber bloques sin la palabra ***sub***. Una de las razones para tener bloques es la de disponer de variables locales que desaparecen cuando el bloque termina.

```
$a=5;                  # variable global que nunca muere
{
     $b=7;             # variable global que nunca muere
     my($c)= 3;
              # "my" crea una variable local que
              # solo existe en este bloque
     funcion_3();
              # $c no es visible dentro de funcion_3
}
print  $a;             #  imprime: 5
print  $b;             #  imprime: 7
print  $c;             #  No imprime nada: $c no existe

sub funcion_3 {
     print $a;         #  imprime: 5
     print  $b;        #  imprime: 7
     print  $c;        #  No imprime nada: $c no existe
}
```

Cuando definimos una variable por regla general tiene un ámbito global al script, a no ser que utilicemos las funciones ***my*** o ***local*** para limitar su alcance.

- La función ***my*** es la más utilizada para definir variables locales. Las variables declaradas con ***my*** son visibles sólo dentro del bloque, y no desde fuera. Tampoco son visibles a las funciones que se llaman desde el bloque.

- La función **local** se usa para definir otras variables locales, pero a diferencia de las anteriores, si son visibles a las funciones que se llamen desde el bloque.

```
$a = 5;  # variable global que nunca muere
{
     local($a)=3;
          #  El valor 5 se guarda temporalmente
```

```
        #  para reponerlo cuando el bloque termine
    local($b)=7;
        #  Como $b no existia, al salir del bloque
        #  no va a existir
    funcion_4();
        #  En funcion_4() se puede usar $a y $b
}
print  $a;       #  imprime: 5
print  $b;       #  No imprime nada: $b no existe

sub funcion_4 {
    print  $a;       #  Imprime: 3
    print  $b;       #  Imprime: 7
}
```

Funciones integradas en Perl

Además de las que ya hemos visto, podemos considerar como más importantes las siguientes:

MANEJO DE STRINGS

- **chop $a;**

  Borra el último caracter del string contenido en $a. Resulta útil para quitar el carácter "\n" al final de una línea que se lee de un archivo de texto. Ejemplo:

```
$a = "abcdef";
chop ( $a ) ;          # $a queda con "abcde";
```

- **length $a;**

  Devuelve la longitud del string contenido en $a. Ejemplo:

```
$a = "abcdf";
print  length($a);    # imprime: 5
```

- **index $a, $x;**

  Devuelve la posición del string **$x** en el string **$a**. Se asume que los índices comienzan en cero. Ejemplo:

```
$a = "abcdef";
$b = index ( $a, "cd" );
print  $b;
```

- **`uc $a;`**

  Devuelve un string con los caracteres de **$a** en mayúsculas, sin modificar **$a**.

- **`lc $a;`**

  Devuelve un string con los caracteres de **$a** en minúsculas, sin modificar **$a**.

- **`substr $a, $pos, $len;`**

  Sirve para extraer un string a partir de otro. El primer parámetro es el string de partida, el segundo parámetro es la posición de comienzo, y el tercer parámetro es la longitud del substring a extraer. Ejemplo:

```
$a = "abcdef";
print substr ($a, 2, 3);  # Imprime: cde
```

  Se puede usar ***substr*** al lado izquierdo de una asignación:

```
$a = "abcdef";
substr($a, 2, 3) = "xy";  # cambia "cde" por "xy"
print $a;                 # imprime: abxyf
```

MANEJO DE ARRAYS

En estas funciones, si no se especifica un array concreto, se utiliza el array por defecto.

- **`join expresion, array`**

  Convierte un array en un escalar concatenando todos sus elementos con el elemento indicado en **expresión**.

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

- **`split /regexp/, expresion;`**

  Convierte un escalar en un array; resulta muy útil para separar campos:

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

  el primer parámetro es una expresión regular, que más adelante veremos en qué consiste.

- **`shift array;`**

  Devuelve el primer elemento del array reduciendo en uno el tamaño del mismo.

```
@a = ('a'..'e');
```

```
$b = shift @a;     # $b queda con 'a'
                   # @a queda con ('b'..'e');
```

- **unshift array, lista;**

  Añade un elemento al princio del array (a la izquierda). La lista puede ser un escalar o una lista de valores.

```
@a = (“b”, “c”);
unshift @a, 'a';
# @a vale (“a”, “b”, “c”);
```

- **pop array;**

  Devuelve el último elemento y lo quita del array

```
@a = ('a'..'e');
$b = pop @a;  # $b queda con 'e'
              # @a queda con ('a'..'d');
```

- **push array, lista;**

  Añade un elemento al final del array

```
push @a, 'e';         # agrega 'e' al final del array
```

- **splice array, offset, longitud;**

  Permite extraer un subarray y modificar a la vez la matriz original.

```
@a = ('a'..'e');
@b = splice(@a, 1, 2);
#  @b queda con 2 elementos de @a: $a[1] y  $a[2];
#       ('b', 'c')
#  @a queda sin esos 2 elementos:
#       ('a', 'd', 'e' );
```

- **map expresion, @a;**

  Devuelve un array después de evaluar la expresión para cada elemento del array que se le pasa como parámetro (@a).

```
@a = ( 'a'..'f' );
@b =  map( uc(), @a );
print "@b";       # imprime: A B C D E F
```

  Otro ejemplo:

```
# Cálculo de las 10 primeras potencias de 2
sub Potencia2 { return (shift)**2;}
```

```
@p = map Potencia2($_), (1..10);
print "@p";
```

- **`grep expresion, @a;`**

  Devuelve un array que contiene los elementos de **@a** donde la expresión es verdadera. En este ejemplo, @b se queda con los que empiecen por "b":

```
@a = ("a1", "a2", "b1", "b2", "c1", "c2");
@b = grep /^b/, @a;
print  "@b";      # imprime: b1 b2
```

- **`sort BLOQUE @array;`**

  Devuelve un array ordenado. El array que se pasa a la función no sufre ninguna modificación. Si se omite el BLOQUE, se obtiene un orden ascendente utilizando una comparación en modo texto. Ejemplos:

```
@a = (3,2,7,8,1,4,6,9,5,10);
@b = sort @a;
@c = sort {$a<=>$b} @a;
@d = sort {$b<=>$a} @a;
print "@b";  # Imprime: 1 10 2 3 4 5 6 7 8 9
print "@c";  # Imprime: 1 2 3 4 5 6 7 8 9 10
print "@d";  # Imprime: 10 9 8 7 6 5 4 3 2 1
```

- **`reverse @array`**

  Devuelve un array invertido. Ejemplo:

```
@a = (1, 4, 5, 7);
@c = reverse @b;  # @c vale (7, 5, 4, 1)
```

FUNCIONES NUMÉRICAS

- `abs($x)` Valor absoluto
- `cos($x)` Coseno en radianes
- `exp($x)` Exponencial (e^x)
- `hex($x)` Transforma un numero Hexadecimal a decimal
- `int($x)` Devuelve la parte entera del número
- `log($x)` Logaritmo natural (base e)
- `rand($x)` Devuelve un número real en el intervalo [0,x)
- `sin($x)` Seno en radianes
- `sqrt($x)` Raíz cuadrada
- `srand($x)` Inicializa el generador de números aleatorios

Referencias

Las referencias son escalares que **apuntan** al valor de otra variable; por tanto, modificando una de ellas, tiene inmediato reflejo en las demás. Una referencia puede apuntar a una variable de cualquier tipo (escalar, array o hash).

```
$ra = \$a;  # referencia a escalar
$rb = \@b;  # referencia a array
$rc = \%c;  # referencia a hash
$rx = \$rb; # referencia a referencia
```

Tambien podemos crear referencias a función y referencias a objetos. Las referencias más interesantes son las referencias a los arrays y a los hashes.

Veamos otra forma de crear una referencias a un array (fijarse en el corchete):

```
$ref_1 = [ 'e1', 'e2', 'e3'];
# Los corchetes sirven para crear un array anónimo, al
# cual vamos a acceder mediante una referencia ($ref_1).
# Para imprimir el primer elemento, utilizaremos:
print $ref_1->[0];
```

Otra forma de crear una referencia a hash (fijarse en las llaves)

```
$ref_2 = { COCHES => 100, MOTOS => 23 };
# Las llaves sirven para crear un hash anónimo, al
# cual vamos a acceder mediante una referencia ($ref_2).
# Para imprimir el primer elemento, utilizaremos:
print $ref_2->{COCHES};
```

Cuando una referencia es dereferenciada se obtiene el dato real

```
$ra = \$a;  # referencia a escalar
$rb = \@b;  # referencia a array
$rc = \%c;  # referencia a hash
$rx = \$rb; # referencia a referencia

#  ${$ra}  Nos da el valor de $a
#  @{$rb}  Nos da el valor de @a
#  @{$ra}  Error, porque $ra apunta a un escalar
#  %{$rc}  Nos da el valor de %c
```

FUNCIONES ÚTILES CON REFERENCIAS

Veamos algunas funciones que resultan de utilidad cuando estamos trabajando con referencias:

- **ref**

  La función **ref** devuelve un string que indica el tipo del referenciado. Ejemplo:

```
$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, devuelve falso.

- **bless**

  Esta función cambia el tipo de una referencia a otro tipo. Es muy utilizado para crear clases en programación orientada a objetos, como veremos posteriormente.

```
$rc = { year=>1995, marca=>'renault', modelo=>'clio' };
$a = ref $rc;     #  $a vale "HASH"
bless $rc, "VEHICULO";
$b = ref ( $rc );
         # $b vale "VEHICULO"; esto tiene más
         # sentido cuando se hable de POO.
```

ARRAYS N-DIMENSIONALES

Uno de los problemas que se achaca a Perl es su falta de soporte directo para dar cabida a arrays de más de una dimensión. Para solucionar esto, se utilizan las referencias a los arrays.

La idea consiste en guardar en un array las referencias a otros arrays. Esto es posible porque un array sólo puede almacenar escalares, y una referencia es también un escalar.

```
@x1 = ( 5, 6, 7 );
@x2 = ( 2, 3, 4 );
@x = ( \@x1, \@x2 );
# @x es un array de 2 dimensiones
# Otra forma de escibir @x sería:
```

```
# @x = ([5,6,7], [2,3,4]);

print $x[0]->[0]  # Imprime: 5
print $x[1]->[2]  # Imprime: 4

# $x[0]->[0]  se puede escribir  $x[0][0]
# $x[1]->[2]  se puede escribir  $x[1][2]
```

V. FICHEROS

Uno de los puntos fuentes de Perl es su facilidad a la hora de manejar ficheros.

Abrir ficheros

Utilizamos la función open:

```
open f1, "c:/autoexec.bat";
```

De esta forma se abre un archivo de lectura. El fichero se maneja con el **descriptor** f1

Otras formas de open:

```
open f1, "<c:/autoexec.bat";    # abrir para leer (es lo mismo
                                # que no poner nada)
open f1, ">c:/autoexec.bat";    # abrir para escribir
open f1, ">>c:/autoexec.bat";   # abrir para agregar
open f1, "+<c:/autoexec.bat";   # abrir para leer y escribir
```

Lectura

En estos ficheros, cada línea termina en "\n". El operador diamante "<>" lee una línea del archivo cuyo descriptor le pasamos como parámetro.

```
open f1, "c:\autoexec.bat";
while( <f1> )     {
    print;
}
```

Aquí <f1> llena $_ con una línea del archivo, que posteriormente se imprime con print; en Perl casi todas las funciones asumen $_ cuando no se pone nada explícitamente.

En realidad <f1> en **contexto escalar** (el resultado se asigna a un escalar) lee una sóla línea, y en **contexto lista** (el resultado se pasa a un array) lee todo el archivo, introduciendo en cada celda del array una línea del fichero.

```
open f1, "c:\autoexec.bat";
@a = <f1>;    # @a tiene todo el archivo
              # cada elemento de @a es una línea del
              # archivo
```

Escritura

Se utiliza la función ***print***, y se le puede pasar un escalar o una lista. Se necesita un descriptor de fichero (previamente abierto con ***open***)

```
print descriptor $dato;
```

```
print descriptor @lista;
```

No hay coma entre descriptor y $dato ó @lista. Si no hay descriptor se asume ***STDOUT*** y si no se pone nada para imprimir, se asume la variable por defecto **$_**

También se puede utilizar la función ***printf***, que permite formatear la salida de forma similar a su homóloga en C. Por ejemplo:

```
printf "%5s %4d %3.2f\n", "hola", 12, 3.1416;
# Imprime: hola   12 3.14
```

Cerrar el fichero

Se utiliza la función **close**. Para el ejemplo anterior:

```
close f1;
```

Se pueden cerrar varios ficheros a la vez, separando cada descriptor por una coma.

Lectura/escritura binaria

Cuando el fichero es binario, carece de sentido hacer una lectura de líneas. En éste caso se recurre a una lectura de bloques de información, cuyo tamaño (en bytes) nosotros podemos definir.

La función utilizada en este caso es:

- **`sysread FICHERO, $buffer, $longitud`**

Esta función lee del fichero especificado, el número de bytes dado en $longitud, y deja dicha información el la variable escalar $buffer. La función devuelve el número de bytes que efectivamente se han leído.

Para la escritura:

- **`syswrite FICHERO, $buffer, $longitud`**

Esta función escribe en el fichero especificado, el número de bytes dado en $longitud, y toma los datos de la variable escalar $buffer. La función devuelve el número de bytes que efectivamente se han escrito, o undef si ocurrió algún error. Si no se proporciona $longitud, se intenta escribir el contenido completo de $buffer.

Funciones para el manejo de ficheros

Función	Significado

`mkdir $path`	Crea un directorio según se especifica en ***$path***. Si hay algún fallo, se actualiza la variable ***$!***
`rename $oldName, $newName`	Cambia el nombre de un fichero
`rmdir $path`	Borra un directorio si está vacío. Si hay algún fallo, se actualiza la variable ***$!***
`stat $fichero`	Devuelve un array de 13 elementos con información relativa al fichero
`unlink @files`	Borra una lista de ficheros.

Operadores para testear ficheros

Existen una serie de operadores que nos permiten saber si un fichero existe, si tiene longitud 0, etc:

- Comprobar si el fichero existe:

```
if(-e $fichero) {
    # Entra aquí si el fichero existe
}
```

- Comprobar si el fichero se puede leer

```
if(-r $fichero) {
    # Entra aquí si el fichero se puede leer
}
```

- Comprobar si el fichero se puede escribir

```
if(-w $fichero) {
    # Entra aquí si el fichero se puede escribir
}
```

- El fichero existe y tiene un tamaño superior a 0 bytes. Devuelve el tamaño del fichero

```
$size = (-s $fichero);
```

- Comprobar si se trata de un directorio

```
if(-d $fichero) {
    # Entra aquí si se trata de un directorio
}
```

VI. EXPRESIONES REGULARES

Una expresión regular (regex) es una forma general de describir un patrón de caracteres que queremos buscar en un string. Este patrón nos permite describir prácticamente cualquier ocurrencia de una cadena. Generalmente, el patrón se escribe entre barras de dividir (//). Su uso principal es para buscar y también para sustituir.

Operadores

Perl define dos operadores especiales (=~ y !~) que permiten testear si un patrón aparece dentro de un String:

```
$resultado = $var =~ /abc/;
```

El valor que toma **$resultado** puede ser:

- True, si se encuentra el patrón en el String
- False, si no se encuentra

En el ejemplo, se busca por el String que hay almacenado en **$var** para ver si se encuentra el string "abc". Si es así, **$resultado** tendrá un valor true.

El operador `!~` es justo la negación de `=~`

```
$result = $var !~ /abc/;
```

Si **$var** contiene el valor "abc", devuelve false.

Los operadores `=~` y `!~` tienen mayor preferencia que los de multiplicar y dividir, pero menor que la exponenciación (`**`).

Caracteres especiales en patrones

Perl da soporte a un amplio conjunto de carateres con un significado especial dentro de una regexp, los cuales suelen utilizarse a menudo

EL CARÁCTER +

El símbolo `+` significa "una o más ocurrencias del carácter precedente". Por ejemplo, la regexp `/de+f/` devuelve true con los siguientes strings:

```
def, deef, deeef, deeeeeeef
```

Los patrones que contienen el carácter +, intentan abarcar el mayor número de caracteres posible. Por tanto, la regexp /ab+/ sobre el string "abbc", considera "abb" y no solamente "ab".

Podemos utilizar el símbolo + para separar palabras por cualquier número de espacios. Por ejemplo, suponer el string:

```
$linea = "Esto es  un   String";
```

Si utilizamos:

```
@palabras = split (/ /, $linea);
```

Obtenemos un array con el siguiente contenido:

```
@palabras = ("Esto", "es", "", "un", "", "", "String");
```

Podemos corregir esta situación haciendo:

```
@palabras = split (/ +/, $linea);
```

Y en este caso,

```
@palabras = ("Esto", "es", "un", "String");
```

LOS CARACTERES []

Los corchetes sirven para definir patrones que ofrecen alternativas. Por ejemplo, la siguiente regexp sirve para buscar `def` o `dEf`:

```
/d[eE]f/
```

Se pueden especificar tantas alternatives como se desee:

```
/a[0123456789]c/
```

Esta regexp busca la letra "a", después un dígito seguido por una "c". Se pueden combinar los corchetes con el símbolo +, por ejemplo:

```
/d[eE]+f/
```

Esto nos permite hacer match en los siguientes strings:

```
def, dEf, deef, dEef, dEEEeeeEef
```

Cuando el carácter ^ aparece en la primera posición depués de `[`, indica que el patron debe hacer match de cualquier carácter excepto los que se indican entre corchetes. Por ejemplo, el patrón

```
/d[^eE]f/
```

Hace match si:

- El primer carácter es una "d".
- El segundo carácter es cualquier cosa excepto la "e" o la "E".
- El último carácter es una "f".

LOS CARACTERES * ? {}

Su funcionamiento es similar al del símbolo +, con la única diferencia de la multiplicidad de la búsqueda realizada.

El carácter `*` busca cero o más ocurrencias del carácter precedente. Por ejemplo, la regexp

```
/de*f/
```

Hace match sobre los strings

```
df, def, deef, deeeeeef
```

El carácter ? busca cero o una ocurrencia del carácter precedente. Por ejemplo, la regexp

```
/de?f/
```

Hace match sobre

```
df, def
```

Sin embargo, no lo hace sobre "deef", porque la "e" aparece dos veces.

Perl nos permite especificar el número de ocurrencias de un patrón, para lo cual utilizaremos las llaves {}. Entre las llaves especificamos dos números separados por ",”; el primero es el número de veces mínima que debe aparecer, y el segundo el número de veces máxima. Por ejemplo,

```
/de{1,3}f/
```

Hace match de una "d", seguido de una, dos o tres ocurrencias de la "e" y finalmente una "f". Para especificar un número exacto de ocurrencias, se pone un único número entre las llaves. Ejemplo:

```
/de{3}f/
```

Este poatrón solo hace match sobre el string "deeef".

Para especificar un mínimo de ocurrencias, dejamos en blanco el segundo número. Por ejemplo:

```
/de{3,}f/
```

Esta regexp busca una "d", seguida de al menos 3 "e" y finalmente una "f".

Similarmente, para esecificar un número de ocurrencias máximo pero no mínimo hacemos:

```
/de{0,3}f/
```

Este ejemplo hace martch de una "d", seguido de no más de 3 "e" y una "f".

EL CARÁCTER .

El punto (.) sirve para hacer match de cualquier carácter excepto el de retorno de carro. Por ejemplo, la regexp

```
/d.f/
```

Busca una "d", seguido de cualquier carácter (excepto \n) y finalmente una "f".

El carácter (.) se usa frecuentemente en combinación con el *. Por ejemplo, la siguiente regexp hace match de cualquier string que contenga en carácter "d" antes de la "f":

```
/d.*f/
```

EL CARÁCTER |

El carácter especial | nos permite especificar dos o más alternativas. Por ejemplo,

```
/def|ghi/
```

Busca tanto "def" como "ghi".

Otro ejemplo:

```
/[a-z]+|[0-9]+/
```

Hace match de una o más letras minúsculas o de uno o más dígitos.

SECUENCIAS DE ESCAPE

Si deseamos incluir en nuestra regexp un carácter especial de los vistos hasta ahora, o de los que más adelante serán mostrados, hay que ponerle precedido de la barra invertida (backslash) "\" Por ejemplo, para buscar si un string tiene uno o más asteriscos, haremos:

```
/\*+/
```

Para incluir un backslash en un patron, hay que poner dos backslashes:

```
/\\+/
```

Otra posibilidad es encerrar los caracteres especiales entre los comandos `\Q y \E`. Por ejemplo, la regexp

```
/\Q^ab*\E/
```

busca cualquier ocurrencia del string "`^ab*`", mientras que

```
/\Q^ab\E*/
```

busca el string "^a" seguido de cero o más ocurrencias del carácter "b"

HACER MATCH DE LETRAS O NÚMEROS

El siguiente patrón sirve para buscar un dígito

```
/[0123456789]/
```

Otra forma de escribir lo mismo es lo siguiente:

```
/[0-9]/
```

Similarmente, el rango `[a-z]` busca cualquier letra minúscula, y el rango `[A-Z]` hace match sobre cualquier letra mayúscula. Por ejemplo, la regexp

```
/[A-Z][A-Z]/
```

busca dos letras mayúsculas consecutivas

Para hacer match de cualquier letra mayúscula o minúscula o de un dígito, utilizaremos:

```
/[0-9a-zA-Z]/
```

LOS ANCHORS ^ $

Sirven para asegurar que el patrón se busca solamente al comienzo o al final del string. Por ejemplo, la regexp

```
/^def/
```

Busca "def" sólo si son los tres primeros caracteres en el String. Similarmente, el patrón

```
/def$/
```

Hace match de "def" solo si son los tres últimos caracteres en el string. Se pueden combinar ambos operadores para hacer match del string completo, por ejemplo

```
/^def$/
```

Es cierto si y solo si el string es "def".

Las secuencias de escape `\A` y `\Z` son equivalentes a `^` y `$`, respectivamente

LOS ANCHORS DE PALABRAS

Los anchors, `\b` y `\B`, especifican si un patrón debe coincidir con un límite de una palabra o debe estar dentro de la misma. (Un límite de una palabra es el comienzo o el final de la misma). Se consideran como caracteres que pueden formar una palabra las letras, los dígitos y el carácter subrayado (_). Los demás se toman como separadores de palabras.

El código `\b` especifica que el patrón debe estar en el límite de la palabra. Por ejemplo, la regexp

```
/\bdef/
```

Hace match solo si "def" está en el comienzo de la palabra. Por tanto, son válidos tanto "`def`" como "`defghi`", pero "`abcdef`" no lo es.

También se puede emplear `\b` para indicar el final de una palabra. Por ejemplo,

```
/def\b/
```

Hace match sobre "`def`" y "`abcdef`", pero no sobre "`defghi`". Finalmente, la regexp

```
/\bdef\b/
```

Busca únicamente la ocurrencia de la palabra "def".

El código `\B` es el opuesto de `\b`. `\B` hace match solo si el patrón está contenido en una palabra. Por ejemplo, la regexp

```
/\Bdef/
```

Hace match sobre "`abcdef`", pero no sobre "`def`".

Sustitución de variables en patrones

Se puede utilizar el valor de una variable escalar dentro de una regexp. Por ejemplo, el siguiente código divide el contenido de **`$linea`** en palabras:

```
$patron = "[\\t ]+";
@palabras = split(/$patron/, $linea);
```

Rangos de caracteres

Existen ciertos rangos de caracteres que aparecen frecuentemente en Perl, y para los que existe definida una secuencia de escape. Por ejemplo,

```
/[0-9]/
```

Es equivalente a

```
/\d/
```

En la siguiente tabla se listan las secuencias de escape para rangos de caracteres más utilizadas:

Secuencia de escape	Descripción	Rango
\d	Cualquier dígito	[0-9]
\D	Cualquier carácter que no sea un dígito	[^0-9]
\w	Cualquier carácter de palabras	[_0-9a-zA-Z]
\W	Negación del anterior	[^_0-9a-zA-Z]
\s	Espacio en blanco, tabulador, retorno de carro	[\r\t\n\f]
\S	Negación del anterior	[^ \r\t\n\f]

Reuso de porciones de patrones

Suponer que deseamos hacer match de lo siguiente:

- Uno o más dígitos o letras minúsculas.
- Seguido de dos puntos o punto y coma.
- Seguido de otro grupo de uno o más dígitos o letras minúsculas.
- Seguido de dos puntos o punto y coma.
- Y otro grupo de uno o más dígitos o letras minúsculas.

Una forma de hacer esto sería la siguiente:

```
/[\da-z]+[:;][\da-z]+[:;][\da-z]+/
```

Sin embargo, Perl proporciona una forma más fácil de especificar patrones repetitivos, y consiste en encerrar la parte que deseemos entre paréntesis:

```
([\da-z]+)
```

Perl guarda la sequencia que hemos puesto entre paréntesis en memoria, y nos podemos referirnos a ellas utilizando la sintaxis \num, don de *num* es un número entero que representa el orden (comenzando en 1) del patrón.

Así, el patrón anterior queda de la siguiente forma:

```
/([\da-z]+])[:;]\1[:;]\1/
```

También se puede almacenar el [:;]. quedando

```
/([\da-z]+)([:;])\1\2\1/
```

Extraer substrings de una regexp

Una vez fuera de la expression, podemos extraer las partes que nos interesan, para lo cual Perl proporciona una serie de variables en las cuales almacena los valores que coinciden con las expresiones de la regexp encerradas entre paréntesis.

Por ejemplo,

```
$string = "Un string con un número: 25.11.";
$string =~ /-?(\d+)\.?(\d+)/;
$integer_part = $1;
$decimal_part = $2;
```

Los valores $1, $2, etc. se borran cuando se ejecuta otra regexp. Existe otra variable especial, $& que contiene el match completo. Podemos hacer, por tanto:

```
$string = "Un string con un número: 25.11.";
$string =~ /-?(\d+)\.?(\d+)/;
$number = $&;
```

Precedencia de los caracteres especiales

Perl define reglas de precedencia para determinar el orden de ejecución. Por ejemplo, la regexp

```
/x|y+/
```

Hace match de o bien "x" o bien una o más ocurrencias de y, ya que el operador + es más prioritario que el operador |.

La precedencias se resumen en la siguiente tabla, de mayor a menor

Carácter	Descripción
`()`	Memoria de match
`+ * ? {}`	Número de ocurrencias
`^ $ \b \B`	Anchors
`|`	Alternativas

Especificar un delimitador de patrón

Podemos determinar que el separador de regexp sea un carácter diferente a la barra de dividir.

```
/de*f/
```

Si deseamos utilizar la exclamación, podemos hacer

```
m!de*f!
```

De esta forma se minimiza el efecto "diente de sierra" que surje al combinar las barras / y \.

Opciones de match

Podemos especificar unas opciones adicionales para determinar como se va a realizar la búsqueda del patron en el string. Se resumen en la siguiente tabla.

Opción	Descripción
g	Match todas las posibles ocurrencias
i	Insensible a mayúsculas y minúsculas
m	Trata un string con multiples líneas
s	Trata un string como una única línea
x	Ignora los espacios en blanco en la regexp

OPERADOR G

El operador "g" dice a Perl que haga match sobre todas las posibles ocurrencias en el String. Por ejemplo:

```
$str = "patata";
$str =~ /.a/g;
```

Hace match de "pa", "ta" y "ta". Si asignamos el resultado a un array, obtenemos todos los match que se han realizado. Por tanto,

```
@matches = $str =~ /.a/g;
```

Hace que @matches contenga:

```
("pa", "ta", "ta")
```

OPERADOR I

La opción "i" nos habilita para hacer búsquedas case-insensitive. Por ejemplo, el siguiente patrón hace match de "de", "dE", "De", "DE".

```
/de/i
```

OPERADOR M

Esta opción le dice al intérprete de Perl que el String contiene multiples líneas de texto. Con este operador, si se pone el carácter especial ^, se busca bien al principio del string o al principio de cada línea. Por ejemplo,

```
/^Un/m
```

Hace match en

```
Este patron tiene\nUn par de líneas
```

Igualmente, el carácter $ busca al final del string y al final de cada línea.

OPCIÓN S

Hace que el string sea tratado como una única línea de texto, y obliga a que el carácter (.) incluya el carácter de retorno de carro.

OPERADOR X

Si la regexp es muy compleja, podemos decir a Perl que ignore los espacios en blanco que pongamos en ella con el objeto de clarificar su contenido. Por ejemplo,

```
/\d{2}([\W])\d{2}\1\d{2}/
```

Es equivalente a:

```
/\d{2} ([\W]) \d{2} \1 \d{2}/x
```

Si se necesita un espacio en blanco, se puede hacer escape con la barra \.

El operador de sustitución

Perl permite reemplazar una parte de un string por otra, apoyándose en las expresiones regulares. La sintaxis es la siguiente:

```
s/pattern/replacement/
```

El intérprete de Perlbusca por el patron especificado, y si lo encuentra lo reeemplaza por lo que hayamos especificado en la segunda parte del operador. Por ejemplo:

```
$string = "abc123def";
$string =~ s/123/456/;
```

Aquí, `123` es reeemplazado por `456`, por lo que `$string` vale ahora "`abc456def`".

Podemos utilizar expresiones como las vistas anteriormente, p.e.

```
s/[abc]+/0/
```

Busca una secuencia consistente en una o más ocurrencias de las letras `a`, `b`, y `c` (en cualquier orden) y reemplaza dicha secuencio por el valor "0". Si lo que queremos es borrarla, en lugar de reeemplazarla, haríamos:

```
s/abc//
```

VARIABLES

En la parte de reemplazar, se pueden utilizar variables que se refiere a la parte del string sobre la que se ha hecho match. Por ejemplo:

```
s/(\d+),(\d+)/$2,$1/
```

Esta regexp busca la ocurrencia de uno o más dígitos. Al estar encerrado entre paréntesis, se almacena en la variable escalar $1 y $2, que se pueden utilizar en la parte de reemplazado. Podemos hacer los siguiente:

```
$numeros = "123,456";
$numeros =~ s/(\d+),(\d+)/$2,$1/;
```

Ahora $numeros vale "456,123";

Para la sustitución se pueden emplear los mismos operadores que vimos con anterioridad, más el operador "e".

OPERADOR E

La opción "e" trata el string de reemplazdo como una expression que debe ejecutar. Por ejemplo, consideremos lo siguiente:

```
$string = "F-12";
$string =~ s/(\d+)/$1*2/e;
```

La segunda parte de la expression de sustitución se ejecuta al establecer la opción "e", por tanto, la variable $string queda finalmente con "F-24".

El operador de traslación

Existe una alternativa para sustituir un grupo de caracteres por otro: el operador **tr**. Tiene la siguiente sintaxis:

```
tr/string1/string2/
```

Aquí, `string1` contiene una lista de caracteres a ser reemplazados, y `string2` contiene los caracteres que los sustituyen. El primer carácter en `string1` es reemplazado por el primero en `string2`, y así sucesivamente. Ejemplo:

```
$string = "abcdefghicba";
$string =~ tr/abc/def/;
```

Y hace lo siguiente:

- Todas las ocurrencias de "a" se cambian por "d".
- Todas las ocurrencias de "b" se cambian por "e".
- Todas las ocurrencias de "c" se cambian por "f".

Al final, $string tiene el valor "`defdefghifed`".

VII. VARIABLES ESPECIALES

Perl tiene toda su maquinaria a la vista.

VARIABLES RELATIVAS A LOS ARRAYS

- **$[** es el índice base de los arrays (default es 0)
- **$"** el separador de elementos cuando se interpola un string de comilla doble (por defecto, es un espacio en blanco).

VARIABLES UTILIZADAS EN ARCHIVOS

- **$.** contiene el último número de línea leído
- **$/** terminación de registro de entrada (default es '\n')
- **$|** si es diferente de 0, se vacía el buffer de salida después de print o write (default es 0)

VARIABLES USADAS CON EXPRESIONES REGULARES

- **$&** contiene el último string que hizo match
- **$+** contiene el string que coincidió con el último paréntesis que hizo match
- **$1, $2, $3** memoria de los matches de los paréntesis

VARIABLES USADAS EN IMPRESIÓN

- **$** se agrega al final del print (por defecto, nulo).

VARIABLES RELACIONADAS CON PROCESOS

- **$0** el nombre del script de Perl.
- **$!** número de error o string con el texto del error.
- **%ENV** hash que tiene las variables de ambiente del programa
  por ejemplo, $ENV{QUERY_STRING}

VARIABLES DIVERSAS

- **$_** variable por defecto en la mayoría de las operaciones que realiza Perl.
- **@ARGV** Argumentos de la linea de comandos con que se llama al script.
- **@_** Array con los parámetros que se pasan en la llamada a una función.
- **$@** El error que se ha producido en el último bloque ***eval*** o ***do*** ejecutado

VIII. PAQUETES Y MÓDULOS

Paquetes

Un paquete es un espacio de nombres. Los espacios de nombres nos permiten utilizar código de otros programadores sin que nuestras variables se confundan con las variables declaradas con el mismo nombre por otras personas en otras partes del código.

El uso más común de los paquetes es el de agrupar funciones que tienen algo en común. Veamos un ejemplo:

```
package PAQ_1;    # Estamos en el espacio de nombres PAQ_1
$a = 5;           # variable del paquete PAQ_1
sub fun1 {        # función del paquete PAQ_1
    print  "$a\n";
}

package PAQ_2;    # Estamos en el espacio de nombres PAQ_2
                  # (salimos del paquete PAQ_1)
$a = 7;           # Variable $a del paquete PAQ_2
print $a;         # imprime 7
print $PAQ_1::a; # imprime 5

PAQ_1::fun1();    # Llama a fun1 de PAQ_1; imprime: 5
PAQ_1->fun1; # Llama a fun1 de PAQ_1; imprime: 5
```

Observa las dos formas equivalentes de llamar la función.

Cuando no usamos ***package*** estamos trabajando en el espacio de nombres ***main***. Como un paquete generalmente se hace para ser reutilizado muchas veces, se guarda en un archivo librería con la extensión ***.pl***, y los programas que lo quieren usar lo invocan con **require**. Por ejemplo,

```
require "cgilib.pl";
```

la función ***require*** lee el archivo ***cgilib.pl*** si este no ha sido leído antes. El archivo no tiene que tener ***package*** pero sí debe devolver verdadero; por tanto, lo mejor es que termine con:

```
return 1;
```

o simplemente

```
1;
```

Las librerías ya no se usan tanto, pero si que conforman la base de lo que son los módulos sobre los que se sustenta la programación orientada a objetos en Perl.

Las funciones de un paquete reciben un parámetro adicional según el operador utilizado para llamarlas:

```
package  PAQ_1;
sub fun2 {
    print "fun2 recibió @_\n";
}

package PAQ_2;
PAQ_1::fun2("xyz");
     # Llama a fun2() e imprime: fun2 recibió xyz
     # ésta forma no se utiliza usualmente para llamar
     # funciones de módulos porque las funciones de
     # módulos se escriben para utilizar un parámetro
     # adicional.
PAQ_1->fun2("xyz");
     # llama a fun2() e imprime: fun2 recibió PAQ_1 xyz
```

Observe que cuando se llama con PAQ_1->fun2, fun2 recibe un parámetro adicional, que es el nombre del paquete ("PAQ_1").

Si ***$r*** es una refencia a un objeto, y hacemos

```
$r->fun2()
```

en este caso el parámetro adicional que recibe fun2() es la referencia ***$r***.

Ejemplo de uso de un package:

```
require Cwd;
$currentDir = Cwd::getcwd();
print "Directorio actual = $currentDir\n";
```

Módulos

Un **módulo** es un paquete en un archivo de su mismo nombre y extensión ***.pm***. Se trata de una particularización de los packages, en el cual se agrupan una serie de funciones y/o variables sobre las cuales se pueden fijar unas reglas a la hora de exportar su contenido. Los módulos constituyen divisiones lógicas de un programa que tiene su funcionalidad completamente definida y diferente del resto, y un módulo se puede utilizar en más de una aplicación.

Los nombres de los módulos suelen empezar por letra mayúscula. Por ejemplo, el módulo `Vehiculo` debe estar en el archivo `Vehiculo.pm`

Vemos un ejemplo de cómo se define un módulo para agrupar la funcionalidad de conversión de divisas entre Pesetas y Euros:

```
package Divisas;
```

```
    use strict;
    use Exporter;

    use vars qw($VERSION @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS
    $CAMBIO);

    $VERSION = 1.00;
    @ISA = qw(Exporter);

    @EXPORT = qw();
    @EXPORT_OK = qw(Euro_Pesetas Pesetas_Euro);
    %EXPORT_TAGS = ( DEFAULT => \@EXPORT,
                TODO => [qw(Euro_Pesetas Pesetas_Euro)] );

    $CAMBIO = 166.386;

    sub Euro_Pesetas {
        return int($_[0]*$Divisas::CAMBIO);
    }

    sub Pesetas_Euro {
        return int(100*$_[0]/$Divisas::CAMBIO)/100;
}
```

Veamos paso a paso que significa cada cosa:

- En primer lugar, obtenemos un *namespace* declarando el nombre del paquete
- Es una buena idea utilizar el *use strict;* en nuestros módulos para restringir el uso de variables globales.
- El módulo requiere el uso del módulo Exporter, el cual proporciona funcionalidades necesarias para el módulo de cara a definir las funciones y/o variables que deseamos exportar al namespace que esté utilizando éste módulo.
- El array ***@EXPORT*** contiene todos los símbolos que son exportados por defecto. Por tanto, la función ***funcion_1*** estará disponible sin más que indicar que se está usando el módulo.
- El array ***@EXPORT_OK*** contiene los símbolos que serán importados bajo demanda.
- El hash ***%EXPORT_TAGS*** agrupa diferentes funciones o variables para ser exportados por grupos en lugar de uno a uno como sucede en el caso anterior. Existe una clave especial que es DEFAULT, y que debemos asignar a lo que se exporta por defecto, es decir al array @EXPORT. Las demás claves pueden ser las que deseemos, y en este ejemplo ponemos una (TODO) en la que se exporta todo. Lo más lógico es hacer grupos de funcionalidades con diferentes claves.

- Cualquier otra función o variable definida en el módulo y que no haya sido incluída en estas listas, será privada y no podrá ser invocada desde fuera del módulo a no ser que utilicemos el prefijo del módulo.

Para importar un módulo en un programa se utiliza **use**. la función ***use*** es similar a ***require*** pero además ejecuta una función del módulo llamada **import**. También se puede descargar un módulo una vez cargado, utilizando la sentencia **no**. Por ejemplo, para utilizar el módulo anterior

```
use Divisas;
# Se importa todo lo que figura por defecto (en @EXPORT);
# Como no se ha puesto nada, hay que utilizar obligatoriamente
# el prefijo Divisas:: al usar las funciones

print Divisas::Euro_Pesetas(1),
print "\n";
print $Divisas::CAMBIO;
```

Otra posibilidad es traer los grupos de funciones que nos interesen (definidos en %EXPORT_TAGS)

```
use Divisas qw(:TODO);
# Se importa todo lo que figura en el tag TODO;
# Ahora se puede suprimir el prefijo Divisas::
print Euro_Pesetas(1);
print "\n";
print Pesetas_Euro(1000);
```

El uso más común de un módulo es para permitir la programación orientada a objetos.

IX. PROGRAMACIÓN ORIENTADA A OBJETOS

Perl no proporciona un soporte directo a la programación orientada a objetos, sino que es una cuestión adicional que se ha añadido al lenguaje.

Clases

Las clases son abstracciones de los objetos que va a utilizar nuestro programa, y en Perl se sustentan sobre los módulos que hemos visto con anterioridad.

Consideremos el siguiente ejemplo:

```
package Toro;
sub dice {
    print "Un toro hace muuuuu!\n";
}

package Caballo;
sub dice {
    print "Un caballo hace hiiiii!\n";
}

Toro::dice;
Caballo::dice;
```

En él, estamos utilizando dos funciones que se llaman igual (sonido) pero situadas en dos paquetes diferentes, cada uno de los cuales representa una clase. Otra forma de invocar las funciones es mediante el operador flecha:

```
Toro->dice;
Caballo->dice;
```

E incluso se puede hacer lo siguiente:

```
$t = "Toro";
$t->dice;
```

Cuando se utiliza el operador flecha sobre una clase, la función a la cual estamos invocando recibe un parámetro adicional, que es el nombre de la clase. Es decir, una llamada del tipo:

```
Clase->funcion(@args)
```

Es equivalente a:

```
Clase::funcion("Clase", @args);
```

Por tanto, podemos rescribir las clases para hacerla más genéricas de la siguiente forma:

```
package Toro;
sub sonido { "muuuuu" }
sub dice {
```

```
    my $clase = shift;
    print "Un $clase hace ", $clase->sonido, "!\n";
}

package Caballo;
sub sonido { "hiiii" }

sub dice {
    my $clase = shift;
    print "Un $clase hace ", $clase->sonido, "!\n";
}
```

Si observamos las funciones ***dice*** en ambas clase vemos que son exactamente iguales. Lo correcto en este caso es crear una clase ***Animal*** en la que ubiquemos las partes comunes a ambas clases y heredemos dicha funcionalidad.

```
package Animal;
sub dice {
    my $clase = shift;
    print "Un $clase hace ", $clase->sonido, "!\n";
}

package Toro;
@Toro::ISA = ("Animal");
sub sonido { "muuuuu" }

package Caballo;
@Caballo::ISA = ("Animal");
sub sonido { "hiiii" }
```

Como vemos, el mecanismo que tiene Perl para heredar de una clase consiste en definir el contenido del array ***@ISA***. Su nombre ("is a") deja bien claro que la clase que lo utiliza es también la clase que hereda.

La secuencia de acciones que suceden cuando invocamos

```
Caballo->dice;
```

es la siguiente:

1. Perl construye la lista de argumentos para el método ***dice***, que en este caso se reduce al nombre de la clase "Caballo".
2. Se busca el método ***Caballo::dice***.
3. Al no encontrarlo, se busca en el array ***@ISA*** la lista de clases padre, y encuentra ***Animal***.
4. Se busca el método ***Animal::dice***.
5. La variable interna ***$clase*** toma el valor "Caballo".
6. Se ejecuta el método ***Caballo->sonido***, el cual existe en la clase ***Caballo***.

Supongamos que ahora queremos añadir un burro a nuestra granja particular, pero deseamos que al llamar al método ***dice***, nos aparezca un mensaje como éste:

```
Un burro hace iiaaaaa!
Resulta ensordecedor
```

No podemos actuar de la misma forma que en los dos casos anteriores, ya que el comportamiento por defecto del método ***dice*** no nos sirve, aunque si podemos utilizarlo para imprimir la primera parte del mensaje.

La nueva clase sería:

```
package Burro;
@Burro::ISA = ("Animal");
sub sonido { "iiaaaaa" }
sub dice {
    my $clase = shift;
    $clase->SUPER::dice;
    print "Resulta ensordecedor";
}
```

Con la sintaxis ***$clase->SUPER::dice***; obligamos a buscar el método ***dice*** entre las clases padre enumeradas en el array ***@ISA***.

Objetos

Hasta ahora nos hemos centrado en las clases, y el método ***dice*** que hemos visto es un método de clase. Sin embargo, lo realmente interesante es poder disponer de instancias diferentes de una clase, cada una con su propia identidad; es decir, objetos.

En Perl, podemos utilizar un escalar para almacenar una referencia a un objeto o instancia de una clase. La forma de crear un objeto consiste en dotar a la clase de una función que construya el objeto y devuelva una referencia al mismo.

Por convenio, se aume que el método constructor tenga el nombre **new**. La clase Burro queda finalmente así:

```
package Burro;
@Burro::ISA = ("Animal");
sub sonido { "iiaaaaa" }
sub dice {
    my $clase = shift;
    $clase->SUPER::dice;
    print "Resulta ensordecedor";
}
sub new {
    my $clase = shift;
```

```
        my $self = {};
        $self->{NOMBRE} = undef;
        $self->{EDAD} = undef;
        bless $self, $clase;
        return $self;
    }
```

Y para crear un objeto, hacemos:

```
        $b = Burro->new;
```

Podemos acceder a las propiedades de este objeto poniendo:

```
    $b->{NOMBRE} = "Platero";
    $b->{EDAD}   = 3;
```

Si creamos otra instancia de la misma clase con el constructor **new**, sus atributos NOMBRE y EDAD son completamente independientes de los anteriores.

UNIVERSAL: La raíz de todas las clases

A partir de la versión de Perl 5.004, se añade de forma automática al final del array @ISA un elemento extra: la clase **UNIVERSAL**, que es por tanto la clase base de toda la jerarquía de clases en Perl.

En la clase UNIVERSAL tenemos definidos los siguientes métodos:

- **`isa()`**

  Indica si un objeto o clase es una instancia del nombre de la clase que se pasa como parámetro. Esto es, si hay una relación jerárquica ascendente. Devuelve "1" si es cierto y *undef* sin no lo es. Ejemplo

  ```
  $b = Burro->new();
  print $b->isa("Animal");  # Imprime: 1
  ```

- **`can()`**

  Sirve para determinar si una determinada función se puede ejecutar sobre una instancia. Si es así, devuelve una referencia a la función.

X. MÓDULOS DE USO COMÚN

En esta sección se presentan brevemente algunos de los módulos que tenemos disponibles en Perl y que resultan muy útiles.

Mail

Una de las tareas que frecuentemente realiza Perl es la lectura y/o envío de correo electrónico.

ENVIAR MAIL

Se basa en el uso de la función sendmail

```
use Mail::Sendmail;

%mail = ( To      => 'su_direccion@alli.com',
          From    => 'mi_direccion@aqui.com',
          Message => "Mensaje de prueba"
);

if (sendmail %mail) {
    print "Mail enviado correctamente.\n";
}
else {
    print "Error al enviar mail: $Mail::Sendmail::error\n";
}
```

Si precisamos enviar adjunto un fichero, utilizaremos:

```
use Mail::Sender;

$sender = new Mail::Sender {
             smtp => 'smtp.servidor.com',
             from => 'mi_direccion@aqui.com'
};

$sender->MailFile( {to => 'la_direccion@alli.com',
                    subject => 'Envío un fichero',
                    msg => "Texto del mensaje",
                    file => 'fichero.txt'}
);
```

LEER MAIL

Necesitamos conocer la dirección el servidor de POP3, un nombre de usuario y su contraseña. Utilizamos la funcionalidad del módulo Mail::POP3Client.

```
use Mail::POP3Client;

$pop = new Mail::POP3Client(
                      USER     => "el_usuario",
                      PASSWORD => "la_contraseña",
                      HOST     => "pop3.servidor.com"
);

foreach ( $pop->HeadAndBody( 1, 10 ) ) {
      print $_, "\n";
}
$pop->Close;
```

Este ejemplo imprime la cabecera de cada mensaje y las 10 primeras líneas del mismo. El módulo dispone de otras funciones para extraer únicamente la cabecera del mensaje o el cuerpo, identificando cada mail por su número.

LWP::Simple

Library for WWW access in Perl.

Este módulo proporciona una simplificación de la librería de accesso al WWW libwww-perl, y permite poder acceder al contenido de documentos HTML a través de su URL. Las funciones más importantes que tenemos disponibles son las siguientes:

- **`get($url)`**

  Busca el documento identificado por $url (un string con la dirección) y lo devuelve. Si no lo encuentra, devuelve undef.

- **`head($url)`**

  Devuelve las cabeceras del documento especificado por su URL. Si tiene éxito la llamada, retorna los dsiguientes cinco valores:

  ($content_type, $document_length, $modified_time, $expires, $server)

  En caso de fallo, devuelve un array vacío.

- **`getprint($url)`**

  Obtiene e imprime el documento por la salida estándar (STDOUT) identificado por su URL. El documento es impreso tal y como se recibe por la red. Si la solicitud fallla, se imprime el código de estado y el mensaje de error por la salida estándar de error (STDERR).

  La función devuelve el código de respuesta HTTP.

- **`getstore($url, $file)`**

  Obtiene el documento especificado por su $url y lo gusrda en el fichero indicado en $file. La función devuelve el código de respuesta HTTP.

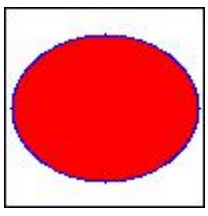
GD

Interface to Gd Graphics Library

Mediante éste modulo podemos generar de forma sencilla cualquier dibujo sencillo y exportarlo a los formatos gráficos más comunes como JPEG o PNG. Podemos combinar esta librería con GD::Graph para realizar gráficos de barras, líneas o de tarta.

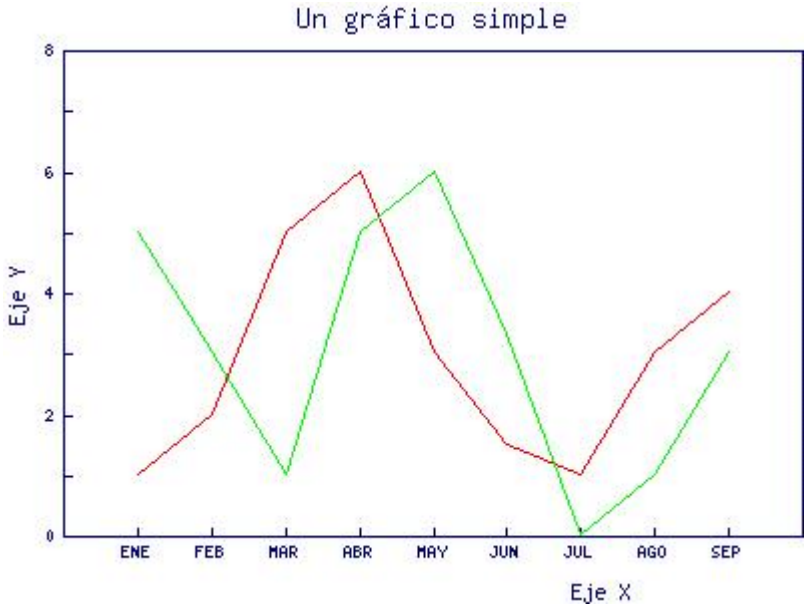
Ejemplo 1: Dibujar una elipse de color rojo

```
use GD;

# Creamos el objeto Image que contiene el dibujo
$im = new GD::Image(100,100);

# Establecemos los colores a utilizar
$white = $im->colorAllocate(255,255,255);
$black = $im->colorAllocate(0,0,0);
$red = $im->colorAllocate(255,0,0);
$blue = $im->colorAllocate(0,0,255);

# Hacemos el fondo transparente y entrelazado
$im->transparent($white);
$im->interlaced('true');

# Dibujamos un rectángulo Negro que bordea la figura
$im->rectangle(0,0,99,99,$black);

# Dibujamos un óvalo azul
$im->arc(50,50,95,75,0,360,$blue);

# Rellenamos la figura en rojo
$im->fill(50,50,$red);

# Grabamos la imagen en un fichero
open IMG, ">imagen.jpg";
binmode IMG;
print IMG $im->jpeg;
close IMG;
```



Ejemplo 2: Un gráfico de líneas

```
use GD;
use GD::Graph::lines;

@data = (
  ["ENE","FEB","MAR","ABR","MAY","JUN","JUL", "AGO", "SEP"],
  [    1,    2,    5,    6,    3,  1.5,    1,     3,     4],
  [    5,    3,    1,    5,    6,  3.3,    0,     1,     3]
);

$graph = GD::Graph::lines->new(400, 300);

$graph->set(
      x_label           => 'Eje X',
      y_label           => 'Eje Y',
      title             => 'Un gráfico simple',
      y_max_value       => 8,
      y_tick_number     => 8,
      y_label_skip      => 2
);

$gd = $graph->plot(\@data);

open IMG, '>file.jpeg' or die $!;
binmode IMG;
print IMG $gd->jpeg;
close IMG;
```



XI. DBI: BASES DE DATOS

El módulo DBI se usa para manipular una base de datos relacional, como Oracle, Access, SQL Server, MySQL, etc. DBI significa Data Base Interface, y supone una capa de alto nivel para acceder a una base de datos. Requiere tener instalado el modulo DBD (Data Base Driver) de la base de datos correspondiente; p.e., si queremos acceder a Oracle, precisaremos de DBD::Oracle.

Para utilizar este módulo es necesario un conocimiento del lenguaje de consultas SQL. En el apéndice A se puede consultar un referencia rápida sobre dicho lenguaje.

Instalación

Si tenemos la distribución de Active State para plataformas Windows, debemos obtener el paquete DBI.zip para instalavar a través de la herramienta PPM. Además, debemos instalar el driver correspondiente a la base de datos a la que nos vamos a conectar. Si se trata de MSAccess, podemos instalar el módulo DBD::ODBC, o bien DBD::ADO.

Conexión a la base de datos

Supongamos que tenemos una base de datos MSAccess. Lo primero que tenemos que hacer es una DSN de sistema que enlace a dicha base de datos (se hace con el administrador de fuentes de datos ODBC, en el panel de control). Si la ponemos el nombre AccessPerl,

```
use DBI;
use DBD::ODBC;
$db = DBI->connect('dbi:ODBC:AccessPerl','', '');
     # $db contiene la conexión a la base de datos
if( ! defined $db ) {
     die "No se puede conectar a la base de datos\n";
}
```

Operación de consulta (SELECT)

Se limita a ejecutar consultas SQL basadas en la instrucción Select. Veamos dos ejemplos de consulta:

- Se cargan los resultados en un array

```
$stm = $db->prepare("select * from prueba");
     # $stm es la sentencia SQL que deseamos utilizar
$stm->execute();
while( @data = $stm->fetchrow_array() ) {
```

```
        # @data contiene en cada iteración una lista con
        # todas las columnas que hemos cargado en la Select
        print "DATA= @data\n";
    }
    $stm->finish();
```

- Se cargan los resultados en variables individuales

```
    my $id, $name;
    $stm = $db->prepare("select codigo, nombre from prueba");
    $stm->bind_columns(\$id, \$name);
    $stm->execute();
    while( $stm->fetch() ) {
        print "$id   $name\n";
    }
    $stm->finish();
```

Operaciones de actualización (INSERT, UPDATE, DELETE)

Después de preparar la setencia, simplemente se ejecuta. Por ejemplo:

```
    $stm = $db->prepare("insert into prueba (codigo, nombre)
values (5,'Perl')");
    $stm->execute();
```

Sin embargo, existe una instrucción que permite aunar las dos anteriores:

```
    $rc = $db->do("insert into prueba (codigo, nombre) values
(5,'Perl')");
```

Si la variable `$rc` no queda definida, se ha producido algún error.

Transacciones

Es necesario modificar los parámetros de la conexión, haciendo:

```
    $db = DBI->connect('dbi:ODBC:AccessPerl',
                '',
                '',
                { AutoCommit => 0 } );
```

Se recomienda dividir las sentencias en bloques dentro de eval, y luego chequear si ha habido error para cancelar el bloque de acciones (rollback) o ejecutarlo definitivamente (commit)

```
$sql1 = "update ventas set num_ventas=num_ventas+1";
$sql2 = "update stock set num_articulos= num_articulos-1";
eval {
        $stm1= $db->prepare($sql1);
        $stm2= $db->prepare($sql2);
```

```
        $stm1->execute();
        $stm2->execute();
        $db->commit();
        $stm1->finish();
        $stm2->finish();
    }
    if( $@ ) {
        warn "Database error:", $DBI::errstr, "\n";
        $db->rollback();
    }
```

Desconexión de la base de datos

Es muy importante no olvidar cerrar la conexión, ya que en caso contrario estamos consumiendo recursos inadecuadamente. En plataformas Windows puede originar, bajo algunas circunstancias errores de memoria graves. Para cerrar la conexión, haremos:

```
$db->disconnect();
```

Control de errores

Cuando se genera un error en la comunicación con la base de datos (lo más común es por errores de programación SQL), Perl genera un error que muestra por la sálida estándar de error (por defecto, la consola) y termina la ejecución inmediatamente. Podemos controlar el grado de errores que deseamos generar:

1. Sólo mensaje de error, sin finalizar la ejecución

```
$db = DBI->connect('dbi:ODBC:AccessPerl', '', '',
                   { PrintError => 1, RaiseError => 0 } );
```

2. Mensaje de error y se aborta la ejecución inmediatamente (lo que se hace por defecto)

```
$db = DBI->connect('dbi:ODBC:AccessPerl', '', '',
                   { PrintError => 1, RaiseError => 1 } );
```

3. Ningún mensaje de error (habrá que consultar la variable ***$DBI:err***)

```
$db = DBI->connect('dbi:ODBC:AccessPerl', '', '',
                   { PrintError => 0, RaiseError => 0 } );
```

Información

- Información sobre el resultado de un prepare o execute

```
$DBI::err
    # N° del error: es falso (no definido) si no hay error
$DBI::errstr
```

```
    # Texto del error: es falso (no definido) si no hay error
```

- Información que se puede obtener después del execute

```
$DBI::rows
    # N° de filas afectadas, que puede ser 0
```

- Información sobre el nombre y tipo de las columnas:

```
$stm->{NAME}
    # Referencia a un array con los nombres de las columnas
    # $stm->{NAME}->[0]   da el nombre de la 1ª columna.

$stm->{TYPE}
    # Referencia a un array con los tipos de las columnas
    # $stm->{TYPE}->[0]   da el tipo de la 1ª columna.
```

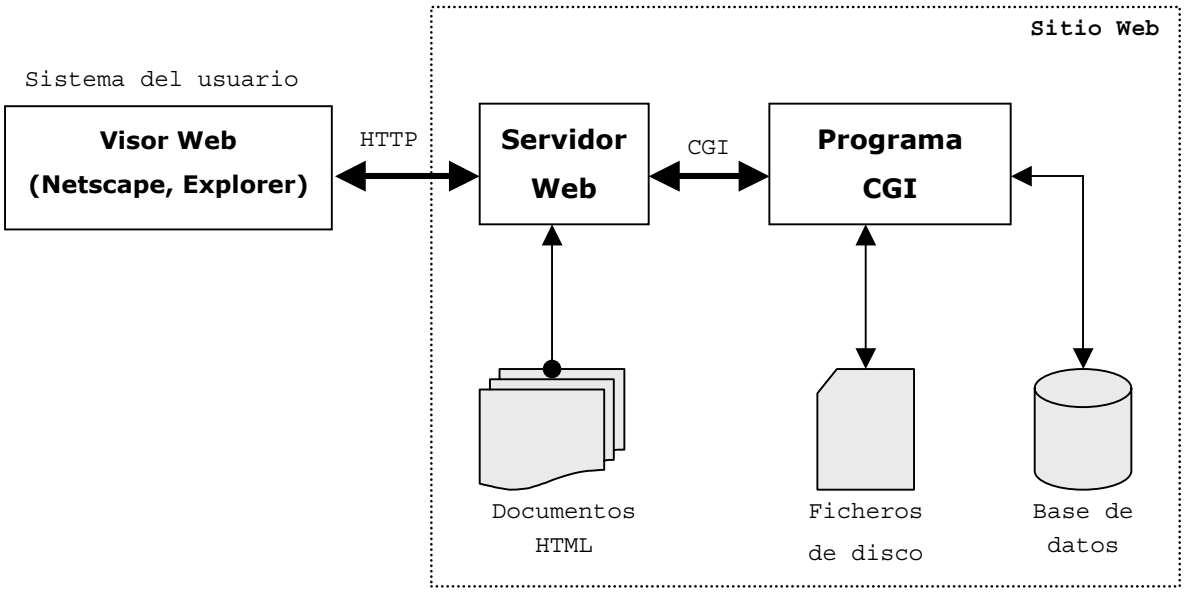
XII. CGI: COMMON GATEWAY INTERFACE

En todo servidor Web, disponemos de un directorio sonde podemos ubicar nuestras páginas HTML. El servidor puede ofrecer una gran variedad de dosumentos HTML, pero una característica clave de estos ficheros es que su contenido es estático, es decir, el documento no cambia a menos que el administrador lo edite y lo cambie explícitamente.

Sin embargo, a menudo el contenido de los documentos no puede conocerse por adelantado. Por ejemplo, si un sitio Web porporciona búsqueda de documentos (como Altavista), el resultado depende de las palabras clave que el usuario haya introducido en el formulario de búsqueda. Para permitir esto, el servidor Web cuenta con programas externos llamados de pasarela o gateway.

Un programa de pasarela admite una entrada del usuario y reacciona devolviéndole los datos que había pedido formateados en un documento HTML. A menudo, el programa de pasarela actúa como un puente entre el servidor Web y otro depósito de información, como una base de datos.

Los programas de pasarela trabajan en el servidor Web. Para permitir que cualquiera pueda escribir un programa de este tipo, es necesaria una especificación de describa las normas de interacción entre el servidor Web y el programa de pasarela. Aquí es donde interviene la Interface de Pasarela Común (CGI, Common Gateway Interface en inglés). CGI define la comunicación entre el servidor Web y los programas de pasarela externos. La siguiente figura describe la interrelación entre el browser, el servidor Web y los programas CGI:



Como se puede observar en la figura, el visor Web intercambia información con el servidor Web utilizando el protocolo ***HTTP***. El servidor Web y los programas CGI normalmente funcionan en el mismo sistema en el que reside el servidor Web. Dependiendo del tipo de

petición que realice el visor, el servidor Web proporciona un documento de su propio directorio de archivos o ejecuta un programa CGI. Conviene destacar que el protocolo CGI no impone el uso de un determinado lenguaje de programación cuando se construya un rpgrama de pasarela, por lo que podremos utilizar el que nos resulte más conveniente.

Secuencia de acciones CGI

Cuando el usuario recupera un documento dinámico HTML a través de CGI, la secuencia básica que se sigue es la siguiente:

1. El usuario selecciona un enlace que provoca que que el visor Web solicite un documento HTML que contiene un formulario.
2. El servidor Web envía el formulario HTML, que es mostrado por el visor.
3. El usuario cumplimenta los campos del formulario y pulsa el botón de envío (***submit***). A su vez, el visor envía los datos del formulario usando un determinado método (GET o POST), como se especifica en la etiqueta METHOD del FORM. Independientemente del método elegido, el browser solicita el URL que figura en el parámetro ACTION del FORM.
4. A partir del URL, el servidor Web detemina la ejecución del programa CGI correspondiente y envía la información a ese programa.
5. El programa CGI procesa la información y devuelve la respuesta HTML as servidor Web (el cual lee la salida del programa CGI). Este programa puede realizar consultas o actualizaciones en bases de datos, lectura o escritura en disco, etc. El servidor, a su vez, devuelve el texto HTML al visor Web.
6. El visor Web muestra el documento recibido.

Métodos de envío GET y POST

Como hemos visto, en el atributo METHOD del FORM podemos especificar dos métodos, GET y POST.

GET

El visor Web envía los datos del formulario como una parte del URL. Se utiliza el comando HTTP GET para enviar los datos. La forma en la que se genera la URL se:

1. Los valores de todos los campos se concatenan en el URL especificado en el atributo ACTION de la etiqueta <FORM>. Cada valor de campo aparece en el formato ***nombre=valor*** (ahora vemos lo importante que es proporcionar un valor para el atributo NAME de los componentes de un formulario).
2. Para asegurarse de que el servidor Web no confunde los caracteres especiales que podrían aparecer en los datos del formulario, cualquier carácter con un significado especial se codifica usando un encriptado especial.

Ejemplo:

Si en la página Web de Altavista introducimos la palabra "Java" y pulsamos en *Search*, aparece la URL:

Cuando el servidor Web tiene que ejecutar el rpograma CGI, tiene además que pasarle la información recibida. Esto lo realiza a través de las variables de entorno del sistema, lo cual impone una restricción en cuanto al tamaño de los datos a pasar, ya que una variable de entorno en algunos sistemas no puede superar los 1024 bytes de longitud.

En los primeros años de la Web, el método GET era el único disponible.

POST

En el método POST de envío de datos, el visor Web utiliza el comando HTTP POST, e incluye los datos del formulario en el núcleo del comando, por lo que no aparecen en la propia URL como en el caso GET. Dadas las limitaciones del método GET, surgió el método POST como una forma de superar todas sus trabas. POST permite gestionar cualquier cantidad de datos, pues el visor envía los datos en un flujo independiente.

Además, el servidor Web no utiliza las variables de entorno para pasar la información al programa CGI, sino que utiliza la entrada estándar del programa CGI, lo cual no imponen restricciones de tamaño en cuanto al volumen de información a transferir.

¿CUÁNDO UTILIZAR GET Y CUANDO POST?

Debemos utilizar el modo ***GET*** cuando:

1. El volumen de información a transferir sea pequeño. Por ejemplo, GET es apropiado para formularios de búsqueda que requieren del usuario unas pocas palabras clave.
2. Se desea acceder a un programa CGI sin usar un formulario.
3. La llamada a una URL con el método GET no debería ser capaz en teoría de alterar nada en el servidor, por ejemplo, una base de datos. Es algo así como "ver pero no tocar".

Por el contrario, debemos utilizar ***POST*** cuando:

1. El volumen de información a transferir sea elevado. Por ejemplo, cuando tengamos un formulario con un campo de sugerencias que no impone una restricción al usuario en cuanto al número de caracteres a introducir.
2. Debería emplearse siempre en operaciones más complejas que las de sólo lectura, como podría ser la actualización de registros en una base de datos.

Paso de parámetros del servidor al programa CGI

Algunos detalles de este paso de información dependen del método utilizado (GET o POST), pero en cualquier caso, sel servidor utiliza las variables de ambiente o de entorno para proporcionar información útil al programa CGI. Las variables de entorno más utilizadas son las siguientes:

Variable de entorno	Significado
CONTENT_LENGTH	Número de bytes de información enviados (método POST).
CONTENT_TYPE	tipo MIME del contenido (sólo con el método POST).
GATEWAY_INTERFACE	Nombre y número de versión del CGI (normalmente, CGI/1.1).
HTTP_ACCEPT	Tipos MIME que acepta el visor Web.
HTTP_REFERER	URL del documento desde el cual parte la solicitud.
HTTP_USER_AGENT	Nombre y número de versión del browser que realiza la petición.
PATH_INFO	Cualquier otro nombre de ruta que sigue al nombre del programa CGI en la URL.
PATH_TRANSLATED	PATH_INFO anexado al directorio raiz del documento del servidor.
QUERY_STRING	Todo lo que sigue al signo ? en la URL (método GET).
REMOTE_ADDR	Dirección IP del sistema desde el que parte la solicitud.
REMOTE_HOST	Nombre del sistema donde el usuario ejecuta el visor Web.
REQUEST_METHOD	Indica si es GET o POST
SCRIPT_NAME	Nombre del programa CGI
SERVER_NAME	Dirección IP o nombre del sistema donde se ejecuta el servidor Web.
SERVER_PORT	Número de puerto, generalmente el 80 u 8080.
SERVER_PROTOCOL	Nombre y versión del protocolo http.

La diferencia más sustancial entre GET y POST radica en el hecho de que GET utiliza la variable ***QUERY_INFO*** para pasar los datos al programa CGI, mientras que POST utliza la entrada estándar, y además la variable de entorno ***CONTENT_LENGTH*** para saber la longitud de los datos.

Procesado de la información en el programa CGI

Un programa CGI puede estar preparado para aceptar la información en uno de los dos métodos posibles, aunque lo usual es que esté preparado para trabjar tanto con GET como con POST. Para ello, el programa debería seguir los siguientes pasos:

1. Comprobar el valor de la variable de entorno REQUEST_METHOD para determinar si la solicitud es de tipo GET o POST.
2. Si se trata de GET, usar el valor de la variable QUERY_STRING como entrada. Comprobar también cualquier nueva información sobre la ruta en la variable de ambiente PATH_INFO, y continuar en el paso (4).
3. Si se trata del método POST, obtener la longitud de la entrada (en número de bytes) a partir de la variable de ambiente CONTENT_LENGTH. Después, leer los bytes a partir de la entrada estándar.
4. Extraer los pares ***nombre=valor*** de varios campos dividiendo los datos de entrada por el carácter **&**, que separa los valores de los campos.
5. En cada par ***nombre=valor***, convertir todas las secuencias ***%xx*** en los caracteres ASCII equivalentes (aquí, ***xx*** representa un par de dígitos hexadecimales).
6. En cada par ***nombre=valor***, convertir todos los caracteres **+** en espacios.

Una vez que tenemos la información de entrada, es cuando comienza realmente el trabajo del programa CGI. Por ejemplo, puede actualizar una base de datos, mandar un correo, etc.

Devolución de datos desde el programa CGI

Independientemente de cómo se transfiere la información desde el servidor Web al programa CGI, éste siempre devuelve información al servidor escribiendo en la salida estándar. En otras palabras, si el programa CGI quiere devolver un documento HTML (normalmente construído dinámicamente), el programa debe escribir dicho documento en la salida estándar. El servidor Web procesa después esa salida y envía los datos de vuelta al visor Web que originalmente envió la solicitud.

El programa CGI envía, además de los datos, una pequeña información de cabecera que debe figurar al comienzo. Esa cabecera incluye el tipo MIME de los datos mediante una línea como la siguiente:

```
Content-type: text/html
```

Hay que dejar una línea en blanco antes de comenzar a escribir el contenido de la información de vuelta. Existen varios tipos MIMES, pero el más común es el anterior, el que se refiere a un documento HTML. Ejemplo de salida de un programa CGI:

```
Content-type: text/html
```

```
<HTML>
     <HEAD>
          <TITLE>Prueba CGI</TITLE>
     </HEAD>
     <BODY>
     Respuesta del programa CGI
     </BODY>
</HTML>
```

Perl y CGI

Sin duda alguna, Perl es el mejor candidato a la hora de elegir un lenguaje de programación en el desarrollo de programas CGI, fundamentalmente debido a las siguientes razones:

1. Perl es un lenguaje disponible de forma gratuíta para todas las plataformas.
2. Dispone de una gran cantidad de módulos adicionales que facilitan sobremanera la programación de CGI's.
3. La potencia de Perl en el manejo de ficheros y la facilidad de la comunicación con las bases de datos es prácticamente inigualable.
4. Fuerte integración de Perl dentro de los servidores Web más populares: Apache y Microsoft IIS.

Importante: Los scripts CGI en Perl deben contener en la primera línea del fichero la ubicación del intérprete de Perl. Si lo tenemos en C:\PERL, la línea en cuestión sería:

```
#! C:\PERL\BIN\PERL
```

El módulo CGI

El módulo CGI.pm ofrece una interfaz de alto nivel que permite realizar scripts CGI rápidamente.

El módulo CGI.pm puede ser utilizado de dos formas diferentes:

- Procedural. Adecuada para scripts pequeños

```
use CGI qw/:standard/;
print   header(),
        start_html(-title=>'Saludos'),
        h1('Saludos'),
        'Hola, mundo !',
        end_html();
```

- Orientada a objetos. Más adecuada para scripts grandes; además permite disponer de varios objetos CGI dentro del mismo programa, con estados diferentes.

```
use CGI;
```

```
$q = new CGI;
print   $q->header(),
        $q->start_html(-title=>'Saludos'),
        $q->h1('Saludos'),
        'Hola, mundo !',
        $q->end_html();
```

CABECERA HTTP

El método header imprime la cabecera (por defecto text/html).

```
print $q->header();
```

COMIENZO DEL DOCUMENTO HTML

Genera la cabecera HTML, pone un título a la página, y abre el BODY:

```
print $q->start_html(-title=>'Prueba Perl', -
BGCOLOR=>'white');
```

FINAL DEL DOCUMENTO HTML

Escribe el cierre del tag BODY de del tag HTML

```
print $q->end_html();
```

TAGS DE FORMATO

```
print $q->hr;                  # imprime <hr>
print $q->i("cursiva");        # imprime <i>cursiva</i>
print $q->b("negrita");        # imprime <b>negrita</b>
print $q->h1("Encabezado");    # imprime
<h1>encabezado</h1>
```

LECTURA DE PARÁMETROS

El uso más frecuente del módulo CGI es la lectura de los parámetros que recibe de un formulario, independientemente de si se han enviado a través de GET o de POST.

```
$name = $q->param('nombre');
$age  = $q->param('edad');
```

TAGS CON ATRIBUTOS

Para añadir atributos a un tag html, se puede pasar una referencia a un array asociativo como primer argumento; las claves y valores del array se convierten en los nombres y valores de los atributos. Por ejemplo:

```
print $q->a({-href=>"enlace.html"}, "Pulsa para ir al enlace");
```

```
# imprime: <a href="enlace.html">Pulsa para ir al enlace</a>
```

TABLAS

Disponemos de los métodos

```
start_table()     # imprime <TABLE>
end_table()  # imprime </TABLE>

start_Tr()        # imprime <TR>
end_Tr()     # imprime </TR>

start_th()        # imprime <TH>
end_th()     # imprime </TH>

start_td()        # imprime <TD>
end_td()     # imprime </TD>
```

FORMULARIOS

Para abrir el tag del formulario (***<FORM>***):

```
print $q->startform($method, $action);
```

Para insertar una caja de texto:

```
print $q->textfield( -name=>'NombreDelCampo',
                     -default=>'valor por defecto',
                     -size=>20,
                     -maxlength=>40 );
```

Para insertar un botón de submit:

```
print $q->submit( -name=>'button_name',
                  -value=>'caption');
```

Para cerrar el tag del formulario (***</FORM>***)

```
print $q->endform();
```

URL

Para obtener la URL del script, utilizaremos la función `url()`

DEPURAR SCRIPTS EN PERL

Las labores de debug en Perl no están tan refinadas como en otros entornos de programación como Visual Basic, por ejemplo. Sin embargo, al tratarse de un lenguaje

interpretado, hace que la localización de errores sea rápida mediante la escritura de mensajes con ***print***.

Podemos probar un script de Perl para CGI sin más que ejecutarlo desde la línea de comandos (sin necesidad de tener un servidor Web activo). Aparecerá el siguiente mensaje:

```
(offline mode: enter name=value pairs on standard input)
```

ahora podemos introducir parejas de de datos ***nombre=valor***, que simulan los datos que recibiría este script del formulario html. Cada línea es una pareja distinta, y para finalizar la introducción de datos pulsaremos ***Ctrl Z*** . Lo que tecleamos podemos escribirlo en un fichero y ahorrarnos un considerable esfuerzo, haciendo que la entrada estándar del script sea tomada de dicho fichero. Por ejemplo:

```
perl miScript.pl < miFormulario.txt
```

Siendo ***miFormulario.txt*** un fichero que podría contener valores como los siguientes:

```
nombre=Jose Miguel
apellidos=Prellezo Gutierrez
centro=CESINE
fecha=02/10/1970
```

XIII. PROGRAMACIÓN EN RED: SOCKETS

Un socket es un punto de conexión a la red que comunica dos procesos. Podemos considerar que un socket se conecta con otro en algún punto de la red, y cualquier cosa que se escribe en uno de ellos se puede leer en el otro.

Existen varias librerías de comunicaciones en Perl, pero para preservar la sencillez, utilizaremos el paquete IO::Socket, el cual suele figurar en todas las distribuciones de Perl. Este paquete proporciona una interface muy sencilla, y para crear un socket basta con llamar al constructor de la clase IO::Socket::INET.

```
use IO::Socket;
# Se crea un socket de escucha en el puerto 1234.
$socket = IO::Socket::INET->new(Proto=>"tcp",
                                LocalPort=>"1234",
                                Listen=>"1")
            or die "No se puede abrir el socket\n";
# ...
close $socket;
```

Como parámetros para el constructor, le podemos pasar los siguientes:

- PeerAddr — Dirección IP del host remoto.
- PeerPort — Puerto IP del host remoto.
- LocalPort — Puerto IP local.
- Proto — Protocolo a utilizar (tcp, udp).
- Listen — Hay que definirlo para los sockets de recepción.
- Reuse — Permite reutilizar el socket.
- Timeout — Tiempo de espera para las operaciones.

Entrada/Salida simple

Para enviar y recibir información sobre un aocket, utilizamos el *handle* que nos devuelve el constructor como si de un fichero se tratara. Veamos un ejemplo que hace una petición http por un documento a un servidor Web, y a continuación imprime todo lo que recibe:

```
use IO::Socket;
$server = $ARGV[0];
$document = $ARGV[1];

$remote = IO::Socket::INET->new(Proto=>"tcp",
                                PeerAddr=>$server,
                                PeerPort=>"80",
```

```
                                    Reuse=>1)
                    or die "No se pudo conectar a $server";
print $remote "GET $document http/1.0\n\n";
while(<$remote>) {print}
close $remote;
```

Veamos otro ejemplo en el cual se espera a recibir datos, que son presentados en pantalla inmediatamente:

```
use IO::Socket;
$local = IO::Socket::INET->new(Proto=>"tcp",
                                    LocalPort=>"1234",
                                    Listen=>"1")
             or die "No se puede abrir el socket\n";

# Se espera a una petición de conexión
$remote = $local->accept;
while(<$remote>) { print }
close $local, $remote;
```

Para esperar la petición de conexión, hay que llamar a la función accept del socket. Esta llamada bloquea la ejecución hasta que llega la petición desde un socket, momento en el cual devuelve un nuevo *handle* de socket que podemos usar.

Información sobre una conexión

Una vez que tenemos un socket que se comunica con otro a través de la red utilizando un determinado protocolo, podemos tener información sobre la identidad de quién está conectándose:

- $handle->peerhost

  Devuelve la dirección IP del socket que se está conectando

- $handle->peerport

  Devuelve el puerto IP del socket que se está conectando

Ejemplo: Servidor Web

Veamos un Servidor WWW mínimo que recibe el comando GET de http.

```
use IO::Socket;
$root = "C:/WebServer/Paginas";
$port = 80;
$maxconn = SOMAXCONN();
$server = IO::Socket::INET->new(Proto=>"tcp",
```

```
                                    LocalPort=>$port,
                                    Listen=>$maxconn,
                                    Reuse=>1)
            or die "Perl-WebServer: no se puede arrancar";
while( $client = $server->accept ) {
    # La primera línea de una petición http 1.0/1.1 es del
    # tipo: "GET document HTTP/1.x"
    @header = split(/ /, <$client>);
    $url = $header[1];
    $httpVer = $header[2];
    if( $header[0] eq "GET" ) {
        if(open FILE, $root.$url) {
            print $client "$httpVer 200 OK\n\n";
            binmode FILE;
            while(<FILE>) {
                print $client $_;
            }
            close FILE;
            print $client "";
        }
        else {
            print $client "$httpVer 404 File not found\n\n";
        }
    }
    close $client;
}
```

XIV. OLE EN WINDOWS

OLE (Object Linking and Embedding) es una tecnología clave desarrollada por Microsoft para sus sistemas operativos Windows. La terminología cambia tan rápido como la tecnología, y no todo el mundo se pone de acuerdo en la utilización de términos como ActiveX y OLE. Podemos considerar que OLE es un subconjunto de la tecnología ActiveX, encargada de la vinculación e incrustación de objetos, y ambas se sustentan sobre COM (Component Object Model).

COM proporciona un mecanismo para permitir la comunicación entre los objetos de una aplicación o entre distintos procesos, proporcionando mecanismos para que un objeto pueda mostrar su funcionalidad a través de una interface.

Por tanto, COM nos proporciona las conexiones y los interfaces que serán utilizados desde OLE para conseguir la automatización, esto es que una aplicación pueda ofrecer una interface progamable.

Podemos utilizar un lenguaje de sripting para poder manejar y controlar las aplicaciones con interfaces OLE y realizar operaciones permitidas por dicha interface.

Las aplicaciones que vienen con Microsoft Office (Word, Excel, Access), el propio Microsoft Internet Explorer, etc. soportan la automatización OLE.

Existe un módulo para Perl que permite realizar scripts capaces de manejar y controlar cualquier aplicación que soporte la automatización OLE.

Para realizar scripts de éste tipo resulta imprescindible conocer las interfaces OLE que nos proporciona cada aplicación y utilizar el módulo de Perl Win32::OLE.

Veamos algunos ejemplos:

Control de Explorer

Este script abre el navegador Explorer y nos lleva a la página principal de el diario El País.

```
use Win32::OLE;
$browser = Win32::OLE->new('InternetExplorer.Application');
$browser->Navigate('http://www.elpais.es',1);
```

Control de Excel

El siguiente script crea una nueva hoja de cálculo, accede a dos celdas para establecer dos números y genera una fórmula para sumar esos dos números en una tercera celda. Accede al resultado para imprimirlo en la salida estándar.

```
use Win32::OLE;

$excel = Win32::OLE->new( 'Excel.Application' )
    or die "No se puede arrancar Excel\n";
```

```
$excel->{'Visible'} = 1;
$newBook = $excel->Workbooks->Add();
$newBook->{Title} = "Ventas 2001";
$newBook->{Subject} = "Ventas";


$newBook ->Worksheets(1)->Range('A1')->{Value}   = '1';
$newBook ->Worksheets(1)->Range('B1')->{Value}   = '2';
$newBook ->Worksheets(1)->Range('C1')->{Formula} = '=A1+B1';

print $newBook ->Worksheets(1)->Range('C1')->{Value};

$newBook->SaveAs({Filename =>"C:/temp/perl/Ventas2001.xls"});

$excel->Quit();
```

Control de Word

El siguiente ejemplo utiliza un documento Word que actúa a modo de plantilla y busca y sustituye unas marcas especiales por el texto correspondiente. Todos los datos originales figuran en un fichero (Aplicaciones.txt), con los campos separados por #. Por cada línea del fichero se genera un documento Word diferente.

```
use File::Copy;
use Cwd;

use Win32::OLE;
use Win32::OLE::Const 'Microsoft Word';

open RESP, "Aplicaciones.txt"
   or die "No se puede abrir el fichero de aplicaciones";

sub CargarInfoWord {
     my $dest = shift;
     my $app  = shift;
     my $desc = shift;
     print "=> $dest\n";
     my $doc = $word->{Documents}->Open("$dest");
     my $search =  $doc->Content->Find;
     my $replace = $search->Replacement;

     $search->{Text} = '@NOMBRE';
     $replace->{Text} = $app;
     $search->Execute({Replace => wdReplaceAll});

     $search->{Text} = '@DESCRIPCION';
```

```
        $replace->{Text} = $desc;
        $search->Execute({Replace => wdReplaceAll});

        $search->{Text} = '@i';
        $replace->{Text} = $num_doc++;
        $search->Execute({Replace => wdReplaceAll});

        $doc->Save;
        $doc->Close;
}

$num_doc = 1;
$word = Win32::OLE->new('Word.Application');
$word->{visible} = 1;

while($linea = <RESP>)
{
        chomp $linea;
        ($resp,$app,$desc) = split(/#/, $linea);
        $dest = cwd . "/$resp/$app.doc";
        copy cwd . "/../plantilla.doc", $dest;
        CargarInfoWord $dest, $app, $desc;
}
```

XV. XML

El lenguaje XML es una de las opciones preferidas en la actualidad para intercambiar información entre aplicaciones, por lo que disponer de una herramientsa capaz de extraer la información de un fichero en éste formato es muy importante.

XML::Parser

Perl dispone del modulo XML::Parser, el cual actúa como un interface compatible con *expat*, el parser XML de James Clark, y permite encontrar o filtrar aquellas partes de un documento XML en las que estamos interesados.

El modulo XML::Parser viene con la distribución estándar de ActiveState, y se trata de un modulo orientado a eventos, lo que significa que analiza el fichero XML y a medida que va encontrando tags de comienzo o final, o cualquier información entre tags se va a llamar a la función manejadora que hayamos establecido.

Para saber como podemos usarlos, debemos conocer los eventos generados por el XML::Parser y sus parámetros.

EVENTOS

Veamos los eventos más comunes y sus parámetros junto con una breve descripción. El primer parámetro siempre es una instancia de Expat, un módulo de uso interno utilizado para procesar el documento, y que a menos que tengamos buenads razones para manipularlo, es mejor ignorarlo.

Handler (parámetros)	**Cuándo sucede**	**Ejemplo**
Init (Expat)	Al comenzar el procesado	
Final (Expat)	Al terminar el procesado	
Start (Expat, Element [, Attr, Val [,...]])	Cuando se detecta el comienzo de un tag XML	<TAG attr1="val1" attr2="val2">
End (Expat, Element)	Cuando se detecta el final de un tag XML	</TAG>
Comment (Expat, Data)	Para los comentarios	<!-- comentario -->

Default (Expat, String)	Cuando no hay un handler específico se llama al de por defecto.	

Los tags del estilo <foo/>, lanzan tanto el evento Start como el End.

MANEJO DE LOS EVENTOS

Disparar un evento quiere decir que una función en nuestro programa va a ser invocada. Para ello, hay que comunicar al módulo XML::Parser cuáles son las funciones manejadoras de cada clase de eventos que estamos interesados en utilizar.

En el siguiente ejemplo, podemos ver como se puede leer un fichero en formato XML, y cómo se llaman las funciones *Start_handler* y *End_Handler* a medida que se realiza la lectura del fichero.

```
use XML::Parser;

# Creamos el objeto parser de XML
my $parser = new XML::Parser ();

# Establecemos los handlers
$parser->setHandlers (
                Start => \&Start_handler,
                End => \&End_handler,
                Default => \&Default_handler
);


# Analizamos un fichero XML obtenido de la línea de comandos
my $filename = shift;
die "No existe '$filename'\n" unless -f $filename;

$parser->parsefile ($filename);
# La llamada a parsefile hace que se vayan llamando a las
# funciones manejadoras de eventos que hemos definido con
# ayuda de setHandlers


# Manejadores de eventos

sub Start_handler {
    my $p  = shift;
    my $el = shift;
```

```
    print "<$el>\n";
    while (my $key = shift) {
        my $val = shift;
        print "  $key = $val\n";
    }
    print "\n";
}


sub End_handler {
    my ($p,$el) = @_;
    print "</$el>\n";
}


sub Default_handler {
    my ($p,$str) = @_;
    print "  default handler found '$str'\n";
}
```