Jugando con Perl

Una breve introducción a Perl

Por Nacho Cabanes,
www.nachocabanes.com

Versión 1.00

Jugando con Perl – 1: Instalación y comprobación

Perl no es el lenguaje que más de moda esté en este momento, pero sigue ahí. Y dicen que es un lenguaje potente, adecuado para automatizar tareas en un servidor, para crear CGIs o para integrarlo con Apache en un servidor web, por ejemplo. Y de hecho, muchos servidores de Internet lo incluyen, además de otros lenguajes que actualmente se oyen mencionar más, como PHP, o como Python (otro que aprenderé pronto... ;-))

El caso es que, aprovechando las vacaciones, he decidido dedicar algo de tiempo a aprender cosas "nuevas". Y una de las que tenía en la lista es Perl. Cuento con la ayuda del libro "Perl in a Nutshell", que adquirí hace ya algún tiempo y que hasta ahora no había mirado más que por encima.

Iré avanzando y practicando, y dejando constancia de mis avances, por si a alguien más le resultaran útiles.

Comenzaremos con un **primer acercamiento**. Un primer vistazo a la sintaxis básica del lenguaje y al uso del intérprete bajo Windows.

Primer paso: **descargar** (e instalar) la versión "lista para usar" bajo Windows, desde

http://www.activestate.com/Products/ActivePerl/

Se trata de una descarga de unos 15 Mb de tamaño. En concreto, la versión que he descargado es la 5.8.8, en formato MSI (instalador de Windows). La instalación es poco más que hacer 3 clics en el botón "Siguiente". El resultado ocupa unos 78 Mb en el disco duro, y él mismo se encarga de situarlo en el Path (o eso se supone), para que podamos usarlo desde cualquier otra carpeta de nuestro ordenador.

Segundo paso: **probar** que funciona. Abro un intérprete de comandos (Inicio / Ejecutar / "cmd", o bien Inicio / Todos los programas / Accesorios / Símbolo del sistema, en ambos casos bajo XP).

Está instalado de c:\Perl así que voy hasta esa carpeta (concretamente hasta su subcarpeta "bin", que es donde se encuentra el intérprete):

```
c:
cd \perl\bin
dir
```

Y efectivamente, en esa carpeta hay 106 ficheros, en su mayoría ficheros EXE o ficheros BAT.

La **sintaxis** básica de Perl es muy **parecida a la de C**. Podemos escribir programas completos (eso mañana), pero también pedir al intérprete que ejecute una única orden, usando la opción "-e", así:

```
perl -e "print 'hola' "
```

Esto escribiría la palabra "hola" (sin las comillas, claro) en pantalla. Pero también podemos hacer algo ligeramente más complejo, como

```
perl -e "for ($i=0; $i<5; $i++) { print 'hola '; } "
```

que escribiría "hola hola hola hola hola " en pantalla.

Jugando con Perl – 2: Variables, print, if, while

Ahora vamos a ver alguna de las características básicas de Perl como lenguaje, para poder comparar con otros como C.

Un ejemplo de cómo dar valores a variables y mostrar estos valores podría ser éste:

```
$x = 1;
$nombre = 'Nacho';
print "Hola, $nombre. x vale $x\n";
```

Como se puede ver, las variables "normales" (lo que en Perl se suele llamar "variables escalares") tienen un nombre que comienza por el símbolo $, y no es necesario declararlas ni indicar qué tipo de datos almacenarán, sino que se deduce a partir del contexto.

Por ejemplo, *$x = 1* hará que la variable "x" almacene un número entero, de valor 1. De igual modo, *$nombre='Nacho'* guarda la cadena de texto "Nacho" en la variable "nombre".

Las cadenas de texto se pueden delimitar entre comillas dobles o comillas simples. La diferencia es que si se encierra entre comillas dobles, se sustituirán las variables que haya en su interior, escribiendo el valor de la variable en vez del nombre.

Estas tres órdenes se podrían teclear en un fichero llamado ejemplo1.pl, y pondríamos este programa en funcionamiento con

```
perl ejemplo1.pl
```

El resultado que obtendríamos sería

```
Hola, Nacho. x vale 1
```

(Al igual que en C, existen ciertos caracteres especiales, como el \n que nos permite avanzar de línea).

Eso de que no sea necesario declarar variables tiene su parte negativa: en este programa

```
$nombre = 'Nacho';
print "Bienvenido, $nombr.";
```

lo que se muestra en pantalla no es "Bienvenido, Nacho." sino "Bienvenido, .", porque hemos escrito mal el nombre de la variable.

(para evitar estos problemas, se puede obligar a que haya que declarar variables, ya veremos cómo).

Muchas de las estructuras básicas (comparaciones, bucles) recuerdan mucho a las de otros lenguajes como C. Un ejemplo podría ser:

```
$x = 5;
if ($x == 5) { print "x vale 5\n"; }

print "La tabla de multiplicar del 2 es:\n";
for ($i=1; $i<=10; $i++) {
  print "2 x $i es " . $i*2 . "\n";
}

print "Y la tabla de multiplicar del 3 es:\n";
$i = 1;
while ($i<=10) {
  print "3 x $i es " . $i*3 . "\n";
  $i++;
}
```

La principal diferencia es que las llaves { y } después de las órdenes como "if" o como "for" son obligatorias, aunque sólo haya una orden en su interior. Por otra parte, el punto (.) se usa para crear una cadena formada por otras dos: *$saludo = "Hola" . "Nacho"*; haría que la variable "saludo" contuviese el texto "HolaNacho".

(Existen más variantes, como "unless" o "until", que veremos en la próxima entrega).

Jugando con Perl – 3: Arrays

Ahora vamos a ver un poco sobre cómo manejar arrays (matrices, vectores) desde Perl.

La primera idea importante es que para declarar un array, el nombre de la variable no se precede con el símbolo de dolar ($) sino con la arroba (@), y los valores se detallan entre paréntesis, separados por comas:

```
@lista = (4, 3, 2, 1);
```

Se puede acceder a los valores individuales usando corchetes: el primer elemento sería @lista[0] y el cuarto sería @lista[3]. También se puede contar "hacia atrás": el último dato sería @lista[-1] y el penúltimo sería @lista[-2].

El tamaño se puede saber simplemente asignando el valor de un array a una variable "normal" (una variable "escalar", siguiendo la nomenclatura de Perl):

```
$longitud = @lista;
```

También existe una orden que permite recorrer una array de forma más sencilla que un "for". Se trata de "foreach", que se usa

```
foreach variable (lista) {
 órdenes
}
```

por ejemplo

```
foreach $nombre (@listaNombres) {
 print $nombre . "\n";
}
```

Este sería un ejemplo completo de todo ello:

```
@listaNumeros = (10,20,30,35);
@listaNombres = ("Juan", "Pedro", "Alberto");

print "Los numeros son: ";
for ($i=0; $i<=3; $i++) {
  print @listaNumeros[$i] . " ";
}

print "\nLos numeros (de otra forma) son: ";
  foreach $i (@listaNumeros) {
print $i . " ";
}

print "\nLos nombres son: ";
foreach $j (@listaNombres) {
  print $j . " ";
}

print "\nEl primer nombre es: ".
$listaNombres[0] . "\n";

print "El ultimo numero es: ".
$listaNumeros[-1] . "\n";

$cantidadNumeros = @listaNumeros;
print "La cantidad de numeros es: ".
$cantidadNumeros . "\n";

@listaNumeros = ();
print "Ahora la cantidad de numeros es: ".
@listaNumeros . "\n";
```

que tendría como resultado:

```
Los numeros son: 10 20 30 35
Los numeros (de otra forma) son: 10 20 30 35
Los nombres son: Juan Pedro Alberto
El primer nombre es: Juan
El ultimo numero es: 35
La cantidad de numeros es: 4
Ahora la cantidad de numeros es: 0
```

En el próximo apartado... las variantes en el manejo de if, while, etc...

Jugando con Perl – 4: unless, until, next, last, redo

Hemos visto el manejo básico de las condiciones "if" y "while", pero estas órdenes dan más juego del que parece:

Para que el código fuente sea más legible, tenemos también órdenes parecidas pero que comprueban el caso contrario, como

```
unless (condición) {órdenes}
```

(las "órdenes" se ejecutaran en caso de que NO se cumpla la condición).

De igual modo, existe una orden similar a while, pero que repite mientras las condición no se cumple (o "hasta que la condición se cumpla"):

```
until (condición) {órdenes}
```

Además, estas 4 ordenes (if, while, unless, until) se pueden usar también como modificadores, al final de otra orden:

```
$y =5 if ($x <>
$y -- unless ($y==0);
print $y while($y <10);
...
```

pero hay más: también podemos encadenar varios "if" con "elsif":

```
if ($x <>
elsif ($x > 0) print "Es positivo";
else print "Es cero";
```

Si alguien viene de C y echa en falta "break" y "continue", también tiene equivalencias:

- "last" hace lo que el "break": termina el bucle (for, while o el que se esté usando).
- "next" equivale a "continue": termina la iteración actual del bucle y comienza la siguiente pasada.
- También existe "redo", que da una nueva pasada sin volver a comprobar la condición.

Por cierto, la orden "goto" existe, pero como su uso es evitable casi siempre, no comento mas sobre ella... ;-)

Jugando con Perl – 5: Expresiones regulares

Una de las grandes ventajas de Perl sobre otros lenguajes "más antiguos" como C es el manejo más sencillo de cosas frecuentes, como las cadenas de texto. Otro es la enorme cantidad de funciones incorporadas, que permiten hacer gran cantidad de cosas con una cierta sencillez. Otro es el uso de expresiones regulares directamente desde el lenguaje. Eso es lo que vamos a ver hoy: las expresiones regulares, y de paso, algo más sobre el manejo de cadenas.

En C, para ver si una cadena empieza por "a", haríamos algo como

```
if (texto[0] == 'a') ...
```

Si queremos ver si empieza por "ab", la cosa se va haciendo más engorrosa

```
if ((texto[0] == 'a') && (texto[1] == 'b')) ...
```

Esto, además de ser más largo y más difícil, tiene "problemas menores": ¿si mi cadena tiene sólo longitud 1, debería yo acceder a la segunda posición? La mayoría de los compiladores, si no se cumple la primera parte de una condición unida por &&, no se molestarían en seguir comprobando, pero... ¿y si mi cadena es "a"? Podría ir afinando un poco la comparación haciendo

```
if ((strlen(texto) >=2) && (texto[0] == 'a') && (texto[1] == 'b')) ...
```

Está claro que apenas una comparación tan sencilla como "empieza por ab en minúsculas" ya va siendo engorrosa de programar en C.

En Perl todo esto es mucho más sencillo, gracias al uso de expresiones regulares. Vamos a ver cómo se haría y luego detallamos qué hemos hecho:

```
$frase1 = "abcde";

if ($frase1 =~ m/ab.*/) {
print "La frase 1 contiene ab y quizá algo más\n";
}
```

Cosas a tener en cuenta para empezar:

- Para **comparar** una cadena con una expresión regular usaremos **=~** (si queremos ver si cumple la condición) o !~ (para comprobar si NO se cumple).
- La expresión regular se indica **entre / y /**
- La operación más sencilla con cadenas es comparar ("match"), así que antes de la primera barra añadiríamos **una m**.
- Un punto (.) quiere decir "**cualquier carácter**".
- Un asterisco (*) quiere decir "el carácter anterior, repetido **0 o más veces**".

(luego en la expresión anterior estamos buscando: a, b, 0 o más veces cualquier carácter).

Si eso queda claro, sigamos...

Para ver si se ha tecleado en mayúsculas o en minúsculas, usaríamos **corchetes**. Los corchetes indican **opcionalidad**, es decir, que nos sirve cualquiera de los elementos que indicamos entre ellos:

```
$frase2 = "aBcde";

if ($frase2 =~ m/[aA][bB].*/) {
 print "La frase 2 contiene ab (mayúscula o minúscula) y quizá algo
más\n";
}
```

Si se trata de un **rango** de valores, podemos usar un **guión**. Por ejemplo, podríamos comprobar si se ha tecleado un número, haciendo

```
$numeroEntero = "12345";
```

```
if ($numeroEntero =~ /[0-9]*/) {
 print "El número entero sólo contiene cifras\n";
}
```

(0 o más cifras del 0 al 9)

Pero siendo estrictos, eso daría una cadena vacía como número válido, y eso es un tanto discutible. Si queremos que al menos haya una cifra, en vez de usar el asterisco (0 o más), usaríamos el símbolo de la suma (+), que indica "**1 o más veces**":

```
if ($numeroEntero =~ /[0-9]+/) {
 print "El número entero sólo contiene cifras\n";
}
```

(1 o más cifras del 0 al 9)

Realmente, esto aceptaría cualquier cadena que contenga un número entero. Si queremos que SÓLO sea una número entero, podemos indicar el **principio de la cadena** con ^ y el **final** de la cadena con $, de modo que haríamos

```
if ($numeroEntero =~ m/^[0-9]+$/) {
 print "El número entero sólo contiene cifras\n";
}
```

(la cadena contiene sólo 1 o más cifras del 0 al 9)

¿Cómo lo haríamos para un número real? 1 o más cifras del 0 al 9, seguidas quizás por un punto decimal (pero sólo 1) y, en ese caso, seguidas también por 1 o más cifras del 0 al 9.

Para eso necesitamos saber alguna cosa más:

- El "." es un comodín, de modo que si queremos buscar si aparece un punto dentro de nuestra cadena, deberemos indicar que no nos referimos al comodín, sino al símbolo del punto ("."), y para eso lo precedemos con la **barra invertida**: \
- Si queremos que algo se repita como mucho una vez (**0 o 1 veces**), añadiremos una interrogación ("?") a continuación.
- Si necesitamos **agrupar** cosas (que se repita "el punto y una secuencia de cifras") lo haremos usando paréntesis antes del símbolo "?" (o el que nos interese en nuestra comprobación).

Con esas consideraciones, lo podríamos escribir así:

```
if ($numeroReal1 =~ m/^[0-9]+(\.[0-9]+)?$/) {
 print "El número real 1 parece correcto\n";
}
```

Un ejemplo que agrupe todo esto:

```
$frase1 = "abcde";
```

```
$frase2 = "aBcde";
$numeroEntero = "12345";
$numeroEnteroFalso = "123a45";
$numeroReal1 = "123.45";
$numeroReal2 = "123.4.5";
$numeroReal3 = "123.";

if ($frase1 =~ m/ab.*/) {
  print "La frase 1 contiene ab y quizá algo más\n";
} else {
  print "La frase 1 no contiene ab\n";
}

if ($frase2 =~ m/ab.*/) {
  print "La frase 2 contiene ab y quizá algo más\n";
} else {
  print "La frase 2 no contiene ab\n";
}

if ($frase2 =~ m/[aA][bB].*/) {
  print "La frase 2 contiene ab (mayúscula o minúscula) y quizá algo más\n";
} else {
  print "La frase 2 no contiene por ab (mayúscula o minúscula)\n";
}

if ($numeroEntero =~ m/^[0-9]+$/) {
  print "El número entero sólo contiene cifras\n";
} else {
  print "El número entero no sólo contiene cifras\n";
}

if ($numeroEnteroFalso =~ m/^[0-9]+$/) {
  print "El falso número entero sólo contiene cifras\n";
} else {
  print "El falso número entero no sólo contiene cifras\n";
}

if ($numeroReal1 =~ m/^[0-9]+(\.[0-9]+)?$/) {
  print "El número real 1 parece correcto\n";
} else {
  print "El número real 1 no parece correcto\n";
}

if ($numeroReal2 =~ m/^[0-9]+(\.[0-9]+)?$/) {
  print "El número real 2 parece correcto\n";
} else {
  print "El número real 2 no parece correcto\n";
}

if ($numeroReal3 =~ m/^[0-9]+(\.[0-9]+)?$/) {
  print "El número real 3 parece correcto\n";
} else {
  print "El número real 3 no parece correcto\n";
}
```

que daría como resultado:

```
La frase 1 contiene ab y quizá algo más
```

```
La frase 2 no contiene ab
La frase 2 contiene ab (mayúscula o minúscula) y quizá algo más
El número entero sólo contiene cifras
El falso número entero no sólo contiene cifras
El número real 1 parece correcto
El número real 2 no parece correcto
El número real 3 no parece correcto
```

Otras posibilidades que no hemos comentado (los ejemplos, más abajo):

- Si queremos que la búsqueda no distinga entre **mayúsculas y minúsculas**, añadiremos una "i" a continuación de la segunda barra,
- Para expresar que algo debe repetirse un **número concreto** de veces, lo indicamos entre llaves: {2,4} querría decir "un mínimo de 2 veces y un máximo de 4 veces"; {3} sería "exactamente 3 veces"; {7,} indicaría "7 o más veces".
- Una barra vertical ("|") indica **opcionalidad**.
- Un acento circunflejo ("^") nada más comenzar un corchete indica que buscamos el **caso contrario.**

Algunos ejemplos más, para poner casi todo a prueba:

- Una secuencia de una o más letras "a": /a+/
- Una secuencia enre 3 y 5 "a": /a{3,5}/
- Cualquier letra: /[a-zA-z]/
- Una secuencia de cero o más puntos: /\.*/
- Una secuencia de letras "a" o "b": /[a|b]+/
- Las palabras "Jorge" o "Jorgito": /Jorg(e|ito)/
- Las palabras "Pedro", "pedro", "Juan", "juan": /[Pp]edro|[Jj]uan/
- Las palabras "Pedro" o "Juan", en mayúsculas o minúsculas: /Pedro|Juan/i
- Cualquier cosa que no sea una cifra: /[^0-9]/

Finalmente, quizá sea interesante comentar que para ahorrarnos trabajo en cosas frecuentes como "cualquier cifra" tenemos abreviaturas similares a "\d". Estas son las más habituales:

```
\d  Un dígito numérico
\D  un carácter que no sea un dígito numérico
\w  Un carácter "de palabra" (alfanuméricos y "_")
\W  Un carácter "no de palabra" (opuesto al anterior)
\s  Un carácter de espacio en blanco
\S  Un carácter que no sea de espacio en blanco
\t  Un tabulador
\n  Un avance de línea
\r  Un retorno de carro
\043  Un carácter indicado en octal (en este caso, 43 octal =  35 decimal)
\x3f  Un carácter hexadecimal (éste es 3f hexadecimal = 63 decimal)
```

(Se puede hacer más cosas aún, pero como introducción es bastante...)

Jugando con Perl – 6: Hash

Otra de las novedades de Perl frente a otros lenguajes "clásicos" es el soporte de "tablas **hash**",

una estructura de datos parecida a los "arrays", pero en la que para acceder a cada elemento no usaremos un número, sino una palabra asociada.

Las variables de tipo "hash" se declaran precediendo su nombre por un símbolo de porcentaje (%). Una primera forma de **definirlas**, bastante compacta, es ésta:

```
%edad = ('Juan', 23, 'Pedro', 31);
```

Para **ver** la edad de Pedro haríamos

```
print "La edad de Juan es " . $edad{'Juan'} . "\n";
```

(cuidado: para acceder a cada dato individual sí se emplea $, y la referencia que buscamos se encierra entre llaves).

Existe otra forma de **declarar** este tipo de variables, que ocupa más espacio pero puede resultar más legible:

```
%estatura = (
  Juan => 182,
  Pedro => 178
);
```

Así se ve más claramente que son dos pares de datos.

En este caso, también se accedería a cada dato de la misma forma

```
print "La estatura de Pedro es " . $estatura{'Pedro'} . "\n";
```

Se puede **modificar** el valor de una variable de tipo hash así:

```
$estatura{'Juan'} = 183;
```

Si usamos este mismo formato con una referencia que no existe, se **añadirá** a la tabla hash:

```
$estatura{'Alberto'} = 169;
```

Podemos **borrar** un dato con delete:

```
delete $estatura{'Juan'};
```

O **eliminar todo** el hash con "undef":

```
undef %estatura;
```

Finalmente, podemos juntar todo esto en un pequeño programa de prueba:

```
%edad = ('Juan', 23, 'Pedro', 31);
print "La edad de Juan es " . $edad{'Juan'} . "\n";
%estatura = (
  Juan => 182,
  Pedro => 178
);
print "La estatura de Pedro es " . $estatura{'Pedro'} . "\n";
$estatura{'Juan'} = 183;
print "La estatura de Juan es " . $estatura{'Juan'} . "\n";
$estatura{'Alberto'} = 169;
delete $estatura{'Juan'};
print "La estatura de Juan es " . $estatura{'Juan'} . "\n";
undef %estatura;
print "La estatura de Juan es " . $estatura{'Juan'} . "\n";
```

En este ejemplo, los dos últimos "print" muestran "La estatura de Juan es" y nada más, porque no hay definido ningún valor que mostrar.

Jugando con Perl – 7: Creando funciones

Ahora vamos a ver cómo crear **funciones** en Perl.

La forma de **declararlas** es usando la palabra "sub" seguida del nombre:

sub suma { ... }

Los parámetros se leen a través del array "@_", que podemos recorrer de varias formas:

- Accediendo a cada elemento: el primer parámetro sería @_[0], el segundo @_[1], el tercero @_[2] y así sucesivamente.
- Usando la orden "foreach", que lee cada valor del array y lo guarda en una variable: foreach (&variable) (@array)
- Usando la orden "shift" para acceder al primer valor y "desplazar" los demás valores, de modo que el que antes era el segundo pase a ser el nuevo primero.

Las variables locales se preceden con "my":

my $valorTemporal = 0;

Si queremos **devolver** un valor se usa "return".

En la forma "clásica" de **declarar** la función, no se usan paréntesis, y la llamada se precede con un símbolo "&", así:

sub suma { ... }
&suma(3,4);

En la forma "moderna" (utilizable a partir de Perl 5) se puede declarar con paréntesis (e indicar el número de parámetros con símbolos "$") y llamar sin paréntesis:

```
sub suma ($) { ... }
suma(3,4);
```

Vamos a agrupar todo esto en un **ejemplo**:

```
# Función para saludar a varios, al antiguo estilo
sub saludo
{
  print "Esto es un saludo para: ";
  foreach $actual (@_) {
    print $actual . " ";
  }
  print "\n";
}

# Función para elevar al cuadrado, al antiguo estilo
sub cuadrado {
  my $numero = shift;
  $numero **= 2;
  return $numero;
}

# Función para sumar dos números, al modo moderno, 2 parámetros
sub suma($$)
{
  return $_[0] + $_[1];
}


# Cuerpo del programa

# Primero llamamos a la función saludo, al antiguo estilo
&saludo("Nacho", "Juan");

#Luego, elevamos 3 al cuadrado
print "3 al cuadrado es : " . &cuadrado(3) ."\n";

#Finalmente, sumamos 4 y 6
print "4 + 6 = " . suma(4,6) ."\n";
```

El resultado de este programa sería

```
Esto es un saludo para: Nacho Juan
3 al cuadrado es : 9
4 + 6 = 10
```

Jugando con Perl – 8: Funciones de biblioteca

Se acaba el tiempo que puedo dedicar a experimentar con Perl.

Aun así, como pequeña muestra de alguna otra de las posibilidades que permite, voy a incluir una lista con algunas de las funciones que incorpora, por categorías. No daré detalles de su uso, pero si a alguien le interesa, al menos ya tiene un poco más fácil para encontrar el resto de la información que pueda necesitar:

Matemáticas

- abs - valor absoluto
- atan2 - arco tangente de Y/X en el rango -PI ... PI
- cos - coseno
- exp - eleva e a una cierta potencia
- hex - convierte una cadena a número hexadecimal
- int - parte entera de un número
- log - logaritmo
- oct - convierte una cadena a número octal
- rand - devuelve un número pseudoaleatorio
- sin - seno
- sqrt - raíz cuadrada
- srand - semilla para números aleatorios

Cadenas

- chop - elimina el último carácter de una cadena
- crypt - encriptación en un sentido, como en passwd
- lc - devuelve una cadena convertida a minúsculas
- lcfirst - devuelve una cadena con sólo la primera letra convertida a minúsculas
- length - devuelve el número de bytes en una cadena
- reverse - da la vuelta a una cadena o una lista
- uc - devuelve una cadena convertida a mayúsculas
- ucfirst - devuelve una cadena con sólo la primera letra convertida a mayúsculas

Sistema operativo y de ficheros

- chdir - cambia el directorio de trabajo
- chmod - cambia los permisos de una lista de ficheros
- chown - cambia el propietario de una lista de ficheros
- exec - abandona este programa para ejecutar otro
- exit - termina este programa
- fork - crea un nuevo proceso hijo, como el actual
- getlogin - nombre del usuario activo
- gmtime - convierte la hora de UNIX a un registro o cadena, usando la hora Greenwich
- link - crea un enlace en el sistema de ficheros
- localtime - convierte la hora de UNIX a un registro o cadena, usando la hora local
- lock - bloquea un hilo en una variable, subrutina o método
- mkdir - crea un directorio
- readlink - determina dónde apunte un enlace simbólico

- rename - cambia un nombre de fichero
- rmdir - borra un directorio
- sleep - espera un cierto número de segundos
- system - ejecuta otro programa
- time - número de segundos desde 1970
- utime - cambia la fecha de acceso y modificación de un fichero
- wait - espera a que cualquier proceso hijo termine
- waitpid - espera a que un proceso hijo concreto termine

Manipulación de ficheros

- close - cierra un fichero
- closedir - cierra un directorio
- eof - comprueba si un fichero ha llegado al final
- getc - lee el siguiente carácter de un fichero
- flock - bloquea todo un fichero
- open - abre un fichero o un "pipe"
- opendir - abre un directorio
- print - envía una lista a un fichero
- printf - envía una lista formateada a un fichero
- read - lectura de datos de tamaño fijo mediante buffer desde un fichero
- readdir - lee un directorio
- readline - lee una línea de un fichero
- seek - salta a otra posición de un fichero (acceso directo)
- seekdir - salta a otra posición de un directorio
- tell - devuelve la posición actual en un fichero
- telldir - devuelve la posición actual en un directorio
- truncate - trunca un fichero

Redes y comunicaciones

- accept - acepta una conexión de socket entrante
- connect - conecta a un socket remoto
- gethostbyaddr - devuelve un host de red, dada su dirección
- gethostbyname - devuelve un host de red, dado su nombre
- msgrcv - recibe un mensaje SysV IPC de una cola de mensajes
- msgsnd - envía un mensaje SysV IPC a una cola de mensajes
- recv - recibe un mensaje sobre un socket
- send - envía un mensaje sobre un socket

...

Pero quedan muchas más. Tienes una lista alfabética detallada -en inglés- en:

http://www.sunsite.ualberta.ca/Documentation/Misc/perl-5.6.1/pod/perlfunc.html

Versiones de este texto

La última versión se podrá localizar en mi página web, www.nachocabanes.com. Las versiones publicadas hasta el momento han sido:

- 10-oct-2007. Primera versión PDF (1.00) de este texto, publicada en mi web (www.nachocabanes.com).
- 08-oct-2007. Publicado el apartado 8 en mi blog
- 07-oct-2007. Publicado el apartado 7 en mi blog
- 29-sep-2007. Publicado el apartado 6 en mi blog
- 27-sep-2007. Publicado el apartado 5 en mi blog
- 16-sep-2007. Publicado el apartado 4 en mi blog
- 24-jul-2007. Publicado el apartado 3 en mi blog
- 17-jul-2007. Publicado el apartado 2 en mi blog
- 12-jul-2007. Publicado el apartado 1 en mi blog (nachocabanes.blogspot.com).