Busca en el tutorial

Tutorial de PERL en castellano :Índice

¿Qué es?

Definiciones básicas de Perl

Perl significa Practical Extraction and Report Language, algo así como lenguaje práctico de extracción y de informes. Es un lenguaje creado por *Larry Wall* (quien, por cierto, es uno de los `net.gods` más conocidos en la actualidad) con el objetivo principal de simplificar las tareas de administración de un sistema UNIX; en realidad hoy en día (en su versión 5.005, y posiblemente pronto, la 5.6) se ha convertido en un lenguaje de propósito general, y una de las principales herramientas de un buen internetero y de un buen webmaster.

Larry Wall es el tipo de personas que han hecho de la Internet lo que es ahora: un lugar con libre intercambio de ideas, donde los que saben de algo, los *gurus*, ayudan al que no sabe. Larry (`larry@wall.org`) solía ser un habitual del grupo de usenet `comp.lang.perl`, y era habitual que el propio Larry contestara a una pregunta de un principiante o a un error con un parche para solucionarlo. Hoy en día, desgraciadamente, ya no es tan común, porque el esfuerzo de trabajar con la comunidad Perl es tal que no le deja mucho tiempo.

Perl es un lenguaje que hereda ciertas estructuras de los intérpretes de comandos de UNIX, especialmente el `csh`, y de otras utilidades estándar, como `awk` y `sed`. Aparte de esto, está diseñado para hacer todo lo que hacen cualquiera de ellos y todos ellos juntos, y la mayoría de las veces de forma más simple, comprensible y fácil de depurar (aunque algunas veces no muy fácil de entender). Si alguna vez habeis tratado de escribir un *script* para el *shell*, sabéis lo que quiero decir.

Perl es un lenguaje interpretado, aunque en realidad, el intérprete de **Perl**, como todos los intérpretes modernos, compila los programas antes de ejecutarlos. Por eso se habla de *scripts*, y no de programas, concepto referido principalmente a programas compilados al lenguaje máquina nativo del ordenador y sistema operativo en el que se ejecuta.

El mejor libro de referencia sobre el Perl es *Programming Perl, por Larry Wall y otros* (llamado el *libro del camello* por razones obvias). Aparte de la referencia, vienen ejemplos, reglas de estilo, y es bastante divertido.

A pesar de que la versión actual del intérprete de **Perl** es la 5, a veces se puede uno encontrar la versión 4.036, el último *patchlevel* de la versión 4 y probablemente el más estable. Actualmente, en enero del 2003, la versión estable es la 5.8, con una nueva versión 6 en desarrollo que va a ser la caña de España, pero que todavía se hará de esperar un cierto tiempo. La versión 5 es prácticamente compatible 100% con la 4; virtualmente todos los scripts que funcionan para la versión 4 lo hacen también en la 5. La mayor parte de los scripts presentados en este tutorial están desarrollados para la versión 4 (porque las primeras versiones de este tutorial son del año 94), pero deberían de funcionar, en principio, para la versión 5 o cualquier otra versión. Cuando son para la versión 5 de **Perl**, se indicará explícitamente con este signo v5.

Aunque desarrollado originalmente en un entorno UNIX, actualmente hay versiones para casi todos los sistemas operativos: Windows XP, Amiga, MacOS (ver Bibliografía/Recursos.). Los scripts son compatibles entre las diversas plataformas, de forma que es un verdadero lenguaje multiplataforma. Muchos fabricantes lo incluyen en sus versiones de UNIX; también el Linux lo incluye. Si quieres saber si está en tu UNIX, escribe simplemente

```
UNIX% which perl
/usr/local/bin/perl
```

y si contesta algo similar a lo anterior, es que está instalado en el sistema. En algún otro "sistema operativo", como Windows xx, acuérdate de si lo instalaste o no, debe de estar en algún lado en tu disco duro. Para bajarte la última versión, consultar el apartado de enlaces

Breve historia del lenguaje PERL

De cómo partiendo de la nada, se llegó a las cimas más altas de la miseria

La primera versión de **PERL** que llegó a ser suficientemente conocida fue la versión 4, dada a conocer al mundo por el *libro del camello* . Esta versión se estuvo desarrollando desde 1991 a 1993, y coincidió con la popularidad del **PERL** como lenguaje para programación de servidores de Internet; aunque originalmente se había diseñado como lenguaje para administración de sistemas.

La versión 5 estable no apareció hasta octubre de 1994, y ha sido tan popular que todavía se usa. Introdujo muchas de las características que hacen al **PERL** tan fácil de programar, incluyendo los módulos, las facilidades para programación dirigida a objetos, referencias y mucho mejor documentación. Aparecen muchos otros libros, tales como Learning Perl.

A partir de la versión 5.6, Perl sufrió una nueva transformación (comenzando por la eliminación de muchos números en sus versiones). Además, se incluye soporte pleno de caracteres internacionales, hebras, y mejor compilador. Se institucionaliza un sistema de *patch pumpkin*, o encargado de cada nueva versión, que es el que decide qué va a entrar de nuevo y qué no, sustituyendo a Larry Wall. Una empresa comercial, ActiveState, que ya participaba activamente en su desarrollo, comienza a controlar más de cerca al PERL, y a la vez, a crear herramientas más potentes (y comerciales) para desarrollo con **PERL**

A partir del año 2000, se empieza a discutir sobre la nueva versión, la 6, que será un gran salto sobre la versión anterior, pero todavía no está muy claro qué es lo que va a ser. Aparte de más rápida, más flexible y todo eso, todavía no se ha comenzado su desarrollo.

Una referencia mucho más completa se puede encontrar en la línea temporal de PERL.

¿Para qué sirve?

Aplicaciones del lenguaje Perl

Prácticamente, sirve para todo. Todas las tareas de administración de UNIX se pueden simplificar con un programa en **Perl**. Se usa también para tratamiento y generación de ficheros de texto. También hay proyectos completos y complejos escritos en **Perl**, pero son los menos.

La forma del lenguaje facilita la programación *rápida y sucia* , el hacer un programa rápido para que funcione. Esto hace también que se utilice para hacer prototipos rápidos de algún algoritmo que queremos ver funcionar antes que tomarnos la molestia de programarlo en un lenguaje más eficiente, como el c++. Y últimamente ha encontrado su aplicación en la escritura de CGI (*common gateway interface*), o scripts ejecutados desde páginas de la World Wide Web. La mayoría de los programas que se encuentra uno para procesar formularios en la Internet llevan la extensión `.pl`, lo cual denota que están escritos en **Perl**.

En general, los programas en **Perl** se ejecutan en el servidor, como todos los programas CGI, a diferencia de otros programas ejecutados por el cliente (generalmente un navegador como el Internet Explorer de Microchof o el Navigator), como aquellos escritos en JavaScript o Java. Existen además extensiones al Apache (`mod_perl`) que permiten ejecutar directamente programas en **Perl** desde el servidor de HTTP.

Mediante una serie de módulos adicionales, tales como el DBD o el ODBC, **Perl** puede servir para acceder a bases de datos, desde BD gratuitas como MySQL hasta el Microsoft SQL server usando ODBC. Por supuesto, esto se puede combinar con un CGI para hacer aplicaciones tales como un carrito de la compra para un sitio web. Pero esto se deja como ejercicio para el lector.

¿Cómo se usa?

Cómo conseguir e instalar programas en Perl

Primero, hay que bajarse e instalar alguna versión de **Perl**. Hay versiones para casi todos los sistemas operativos, o sea que no debería de ser complicado. Es conveniente consultar la sección de enlaces, para ver dónde podemos conseguirlo.

En todo caso, lo más probable es que si tienes una distribución de Linux más o menos decente, tal como RedHat , SuSE o Debian, venga ya incluido. Consulta los paquetes que tienes instalados (usando la herramienta correspondiente) o simplemente escribe:

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

Si contesta algo así como lo anterior, es que está instalado (y si quieres nota, escribe

```
bash$ perl -v

This is perl, v5.8.0 built for i386-linux-thread-multi

Copyright 1987-2002, Larry Wall

Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using 'man perl' or 'perldoc perl'.  If you have access to the
Internet, point your browser at http://www.perl.com/, the Perl Home Page.
```

lo que, como ves, te devuelve la versión, e información adicional.

En Windows, como siempre, es otro cantar. Sin embargo, hay una excelente versión de Perl para Windows, de la empresa ActiveState. No hay más que bajárselo, pulsar en el icono de SetUp correspondiente, y se lanza un procedimiento de instalación similar al de todos los programas. Desde ese momento, ya se puede usar desde la línea de comandos.

En MSDOS (¿hay alguien que todavía use MSDOS?), basta descomprimirlo, y añadir al PATH el directorio donde se encuentra `perl.exe`.

En otros sistemas operativos, seguir el procedimiento respectivo en caso de que exista una versión ya compilada, pero puede que no exista una versión binaria. Habrá entonces que bajarse los fuentes de alguno de los sitios web existentes, y luego compilarlos usando los siguientes comandos (si es que es un SO parecido a UNIX:

```
sh Configure
make
make test
make install
```

Para más información, consultar el fichero `INSTALL` o `perldoc INSTALL` que viene con los fuentes (malamente podrás usar perldoc si no tienes instalado Perl, pero en fin...).

Mi primer programa en Perl

Cómo escribir y ejecutar un programa simple Perl

Escribir el archiconocido "Hola" en **Perl** no puede ser más fácil:

```
print "Passa, tio\n";
```

Y eso es todo. No hay necesidad de abrir el programa, ni de cerrarlo, ni de incluir librerías estándar o no, ni nada de nada. Solamente hay que tener cuidado de terminar todas las sentencias con; . `\n`, que se suele leer "escape-N", tiene el mismo significado que en C; es decir, un retorno de carro.

Ejecutarlo es otro cantar; o mejor dicho, muchos cantares, una cantata entera. El **Perl** siempre Hay Muchas Formas de Hacer Las Cosas™. Al menos tres, en este caso.

- Guarda el programa en un fichero, llamémoslo `passa.pl`. Se puede ejecutar con

```
C:\PROGS\Perl>perl passa.pl
Passa, tio
```

Todo esto suponiendo que **Perl** esté en el PATH de ejecución, claro está.

- Pasa de guardar el programa en un fichero y ejecútalo directamente. Se le da el switch `-e` al intérprete para indicar que se debe ejecutar el script que viene a continuación (y ojo con las comillas)

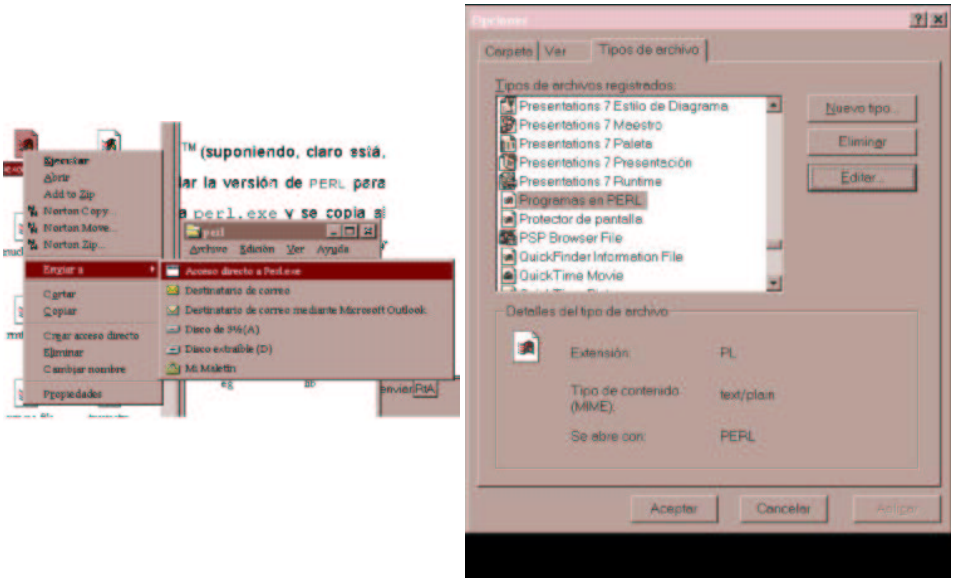
```
C:\PROGS\PERL>perl -e 'print "Passa, tio\n";'
Passa, tio
```

- Si estás en UNIX, se utiliza una sintaxis similar a los scripts del intérprete de comandos, es decir, guardar el fichero con una primera línea que indica dónde está el intérprete que va a ejecutar el resto del fichero. En el caso del **Perl**, puede ser algo como `#!/usr/local/bin/perl`, o `#!/usr/bin/perl` en RedHat Linux, por ejemplo) (los símbolos iniciales se suelen denominar *pound-bang*, o almohadilla-admiración, en román paladino). En todo caso, habrá que sustituirlo por el camino completo donde habita el intérprete de **Perl** (si no se sabe, recurre al comando de UNIX `which` (como hemos visto antes), o, por último, al operador de tu sistema; ahora, que si tú mismo eres el operador de tu sistema y no sabes como hacerlo, tienes un problema, tío... Bueno, siempre te queda la internet). Tras salvar el fichero, habrá que dar la orden

```
UNIX% chmod +x passa.pl
```

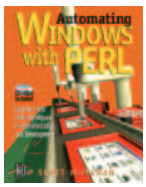
para convertirlo en ejecutable; hecho eso, se puede ejecutar directamente escribiendo

```
UNIX% passa.pl
Passa, tio
```



- En Windows y familia, hay al menos dos formas de hacerlo[TM](suponiendo, claro está, que le pille de buenas y te deje hacer algo). Tras instalar la versión de **Perl** para WinXX de la Internet, de la empresa ActiveState, se crea un acceso directo a `perl.exe` y se copia al directorio `\Windows\SendTo`; esto hará que aparezca **Perl** dentro del menú *enviar a*, que es una de las opciones del menú que surge al pulsar con el botón derecho del ratón.
  Otra forma es más específica para ficheros de tipo `.pl`, y es el mismo mecanismo que se usa para asignar iconos y programas ejecutables a los ficheros no ejecutables: en la carpeta MiPC, pasar a Ver - Opciones y pulsar sobre la pestaña Tipos de Archivo: Se crea un nuevo tipo que sean "Programas en Perl", y se le pueden asignar acciones como edición o ejecución; de esta forma, con sólo pulsar dos veces sobre el icono, se ejecuta el programa.

Para trabajar con Perl en Windows, se puede ver *Automating Windows with PERL, por Scott McMahan* .



Para editar un programa en **Perl**, lo más adecuado es conseguir un editor para programadores, porque el lenguaje no incluye un entorno integrado de programación. Lo más parecido a tal entorno integrado, tanto en Win95/NT como en UNIX, es el `emacs`, puesto que tiene un *modo **Perl*** de edición, que tiene en cuenta indentaciones y otras particularidades de su sintaxis, como el emparejamiento de paréntesis y llaves y los comentarios, y asigna diferente color a las palabras dependiendo del contexto. Otra opción con unos pocos menos megas puede ser cualquier editor de programador con diferentes opciones dependiendo de la extensión del programa, como el *Programmer´s File Editor* . Aunque, por supuesto, si puedes conseguirte el emacs para Windows, es con diferencia lo mejor.

Más difícil todavía

Variables, interpolación y cadenas

Ya que hemos hecho nuestro primer programa, vamos a por el segundo. Supongamos que somos un político corrupto cualquiera, y que, dado el alto número de transacciones diarias por las cuales cobramos comisiones ilegales, decidimos escribir un programa en **PERL** que nos resuelva la difícil papeleta de calcularlas sobre la marcha. Ni cortos ni perezosos, nos puede servir lo siguiente (comile.pl):

Listado : Programa comile.pl

```
print "Valor del inmueble? ";
$valor = <STDIN>;
$comision = $valor * 0.25;
print "Comision = $comision\n";
```

que da el resultado siguiente

```
C:\PROGS\PERL>perl comile.pl
Valor del inmueble 1000000 [Escrito por el usuario]
Comision = 250000
```

A partir de este momento, indicaremos en `rojo` las palabras clave de **PERL**, en `verde` las variables de **PERL** y en `azul` las órdenes de control.

En este ejemplo hay algunos elementos más de **PERL**. Como se ve, las variables se indican con un símbolo de $ (sumamente indicado para el ejemplo); en este caso se trata de variables escalares, que contienen un valor numérico o una cadena; a **PERL** le da exactamente igual que se trate de uno o de otro, la transformación se hace automáticamente. Y no hace falta ninguna declaración de variables; cada variable se declara en el momento que se usa, y si no se ha usado anteriormente, aparece automáticamente con el valor 0 o ´´ (cadena nula). Las variables en **PERL** tienen todas visibilidad global, si no se dice lo contrario.

El programa pide además, la intervención del usuario, leyendo en una variable lo que éste teclea. Para ello se utiliza un *filehandle* o puntero a un fichero abierto (el equivalente a un `FILE*` en C), que está disponible en todos los programas, la entrada estándar o `stdin` (standard input); de la misma manera, también existe la salida estándar o `stdout`, es decir, que es lo mismo `print STDOUT` que `print`). El nombre de esta variable no está precedido por ningún símbolo de $ para indicar que se trata de un *filehandle*, o una variable que representa un fichero. Los *angle brackets*, o paréntesis angulares `<>`, indican que se lee del fichero una línea completa, hasta que el que teclea pulsa un retorno de carro.

Por último, se está utilizando la operación denominada *interpolación de variables*. Una variable incluida dentro de una cadena entre comillas dobles será sustituida por su valor (no en el caso de tratarse de comillas simples).

Dando vueltas sobre lo mismo

Bucles, lectura de teclado y ficheros

Cuando, tras las elecciones, nuestro político corrupto sale reelegido por méritos propios, viene inmediatamente la recuperación económica y con ello el boom inmobiliario (favorecido por la recalificación en terrenos construibles de los parques de la ciudad). Tantas comisiones tiene que calcular, que decide escribir un programa que lo haga continuamente, en vez de tener que ejecutar el programa anterior cada vez. Decide además guardar detalles de todo en un fichero, que llevará en un disquete escondido en el collar del perro, para que, en caso de que lo investigue una comisión del Congreso, pueda tirar de la manta tecnológicamente. Saldría entonces algo como lo incluido en el listado siguiente:

Listado : Programa manta.pl

```
#!/usr/bin/perl
open( MANTA, ">clientes");
while(1){
  print "Cliente\n";
  chop( $paganini = <STDIN> );
  last if !$paganini;
  print "Valor inmueble\n";
  chop( $valor= <STDIN> );
  $comision = $valor * 0.25;
  print "Comision = $comision\n";
  print MANTA "$paganini $comision\n";
}
close MANTA;
```

En este segundo programa, que llamaremos `manta.pl`, se introducen algunos conceptos nuevos más. Para empezar, se utiliza la orden `open` para abrir un fichero; a esta orden se le dan dos parámetros, el *filehandle* por el cual nos vamos a referir al fichero en lo sucesivo, y una cadena que incluye el nombre del fichero y la forma de manejar ese fichero. En este caso, se usa la expresión `">clientes"`, que indica que se va a abrir para escritura. Otras opciones son las que aparecen en la tabla 1. Si no se pone nada, se supone que el fichero se abre para lectura. Y esto es fuente de continuas confusiones.

Como siempre, hay Más de Una Forma de Hacerlo ™, aunque no necesariamente más corta. Se puede meter el nombre del fichero en una variable tal como `$MANTA`, y abrirlo de la forma siguiente:

```
$MANTA='>prueba';
open MANTA;
```

Es decir, cuando no se pone una cadena detrás del *filehandle*, se usa el contenido de una variable que tenga el mismo nombre. Véase también que en este caso no he usado paréntesis detrás de open; son opcionales. Hay muchas más cosas sobre open leyendo el tutorial incluido en la documentación, se accede a él escribiendo `perldoc perlopentut` o `man perlopentut`.

Tabla : Modos de apertura de ficheros

>	Abrir para escritura
>>	Abrir para concatenar
<	Abrir para lectura
>+	Abrir para lectura/escritura
¦ orden	Ejecutar un programa que filtrará lo que se imprima en el fichero.
orden |	Ejecutar una orden, de la cual se leerá la salida.

A continuación se comienza un bucle con la orden `while`, que se ejecutará mientras la expresión entre paréntesis sea cierta, o sea, en este caso, en principio, siempre. Los valores "verdaderos" se indican en **Perl** con un número distinto de 0 o una cadena no nula. Tras `while` va siempre un bloque, que se comienza y termina con llaves. Dado que es un bloque, y no la alternativa orden|bloque (como sucede, por ejemplo, en el lenguaje C), se tendrán que utilizar siempre las llaves, aunque se trate de un solo

comando. También se podría sustituir esta línea por `until(0) {`que tendría exactamente el mismo significado (recuerda, Hay Más De Una Forma de Hacerlo). O por dos o tres formas más, al menos.

En la línea 4 se hacen dos cosas juntas: se asigna a una variable lo que introduce el usuario, y a esta variable se le elimina el último carácter (`chop`, que significa *trocear*, como en el chopped). Esto es necesario porque, a diferencia de otros lenguajes, **Perl** incluye el retorno de carro en la cadena que lee. Lo eliminamos entonces para que quede todo bonito.

Tabla : Tipos de bucles en Perl

```
ETIQUETA while (EXPR) BLOQUE
ETIQUETA while (EXPR) BLOQUE continue BLOQUE
ETIQUETA for (EXPR;EXPR;EXPR) BLOQUE
ETIQUETA foreach VAR (MATRIZ) BLOQUE
```

En la *línea 5* se utiliza una construcción típica:

```
Perl <comando> <condicional> <condición>.
```

En este caso, se sale del bucle (`last`) en caso de que lo que se haya leido sea la cadena vacía (recordad que previamente hemos eliminado el retorno de carro). Esta línea se podía haber sustituido por la siguiente`last unless $paganini;`que tiene exactamente el mismo significado (`unless` significa a menos que); en general, una sentencia en la que se utiliza `if` con una expresión verdadera se puede sustituir por otra en la que se utiliza `unless` con la misma expresión negada. Otras expresiones que regulan bucles son `next`, que ejecuta la iteración siguiente sin pasar por el resto de las órdenes del bucle, y `redo`, que vuelve a comenzar el bucle sin evaluar la condición.

Tabla : operadores de cadenas en Perl

lt, gt, le, ge	Lexicográficamente menor, mayor, menor e igual, mayor e igual
eq, ne	Igual, distinto
cmp	Comparación de cadenas; devuelve -1, 0 o 1
x	Multiplicación de cadenas "az" x 2 eq "azaz"

En esta misma línea se usa el operador `!`, de negación. Este operador, como otros muchos, están sacados directamente del lenguaje C, en general, los operadores en **Perl** son los mismos que en C, y además tienen la misma prioridad. Además, hay una serie de operadores específicos para cadenas alfanuméricas, sacados más bien del FORTRAN, y otros para ficheros que se verán más adelante.

En la línea 10 se escribe en el fichero `MANTA`; como se ve, simplemente se incluye el *filehandle* delante de la expresión que se va a escribir. Para terminar, después de concluir el bucle se cierra el fichero.

Estos primeros programas nos permiten ya tener una idea de cómo funciona **Perl**. En realidad, salvo algunos asuntos menores de sintaxis de variables y su declaración, es muy similar al C: por eso siempre, la primera aproximación a un programa será hacerlo tal como uno lo haría en C, para luego, cuando se tenga mayor dominio del lenguaje **Perl**, hacerlo de forma más perlera.

El político corrupto se da cuenta de que tiene en un fichero guardadas una serie de sobornos anteriores en un fichero; pero no le puso nombre; así que, antes de que se le vayan las cosas de la memoria y no diga nada salvo en presencia de su abogado, decide hacer un programa que le vaya preguntando cantidad por cantidad quién fue la que se la dió. Y decide hacer el siguiente programa (`recuerda.pl`):

Listado : Programa recuerda.pl

```
1 #!/usr/bin/perl
2
3 open FICHERO, "cantidades";
4 open SALIDA, ">salida";
5 while($linea = <FICHERO>) {
6   chop($linea);
7   print "Quien te ha dado $linea pelas? Eh?\n";
8   chop( $menda = <STDIN> );
9   print SALIDA "$menda $linea\n";
10 }
11 close FICHERO;
12 close SALIDA;
```

Este programa utiliza las mismas estructuras que el anterior: ficheros de entrada y salida, y un bucle. Sin embargo, se usan de forma diferente. Para empezar, la sintaxis de apertura de los ficheros en las líneas 3 y 4 es diferente: se evitan los paréntesis, casi 4 bytes, que en una memoria de 512 megabytes, quieras que no, es un ahorro. El fichero `cantidades` se abre para lectura, por lo que no es necesario poner el símbolo <, y el fichero `salida` se abre para escritura.

Lo que sigue es bastante típico de la lectura de ficheros en Perl: se lee una línea del fichero abierto (usando la construcción `<>`, se asigna a la variable `$line`. Además, la línea está dentro de una condición de continuación de un bucle; efectivamente, `$line` será la cadena vacía, y por lo tanto falso, cuando el fichero deje de tener elementos.

El interior del bucle no aporta nada nuevo: simplemente se le eliminan los retornos de carro a lo que se lee usando `chop`, y se combina la entrada con lo que introduce el usuario para escribirlo en la salida. Finalmente, en las líneas 11 y 12, se cierran los ficheros.

Ejercicios

- Realizar un programa que vaya calculando cuadrados de números hasta que se introduzca una línea en blanco (solo retorno de carro).
- Realizar un programa que permita al usuario introducir un nombre, y el ordenador le escriba "Hola >mismo nombre en mayúsculas<".
- Realizar un programa que solicite nombres y apellidos del usuario, y los imprima en pantalla en formato formal: apellidos y nombre, separados por coma. Escribir en un fichero todos aquellos que se llamen Juan de nombre.

Ahorrando energías

Operadores, matrices y la variable por defecto `$_`

Como nuestro político corrupto, por su experiencia en el ramo de la construcción, ha sido nombrado delegado de Obras Públicas, tiene que pasar la mayor parte del tiempo visitando carreteras, caminos, puentes y veredas. Nada mejor, pues, que comprar un portátil AMD K6-2 para ir introduciendo las mordidas en el propio lugar donde se originen. El problema es el limitado rango de las baterías, que hacen que si hay que viajar de Albacete a Cuenca, se gaste la batería a la altura de Motilla del Palancar. Para ahorrar energía, decide modificar su programa de forma que escriba menos en el disco; es decir, que cargue todos los datos en memoria, y los guarde sólo cuando termine el bucle. Además, para no poner la excusa a la comisión del Congreso (que ya pide Anguita a voces) de que no se acuerda de nada, decide grabar también las fechas y horas de los sucesos. Escribe, con su pericia creciente en **Perl**, el programa `memoria.pl` (listado 3).

Listado : Programa memoria.pl

```
1 until(0) {
2       print "Cliente\n";
3       chop( $paganini = <STDIN>);
4       last if !$paganini;
5       print "Valor inmueble\n";
6       chop($valor = <STDIN>);
7       $comision = $valor * 0.25;
8       ($seg, $min, $hora, $dia, $mes, $anho, @zape) = localtime(time);
9       $mes++;
10       $anho+=1900;
11       $zipi = "$paganini $comision $hora:$min $dia/$mes/$anho\n";
12       push( @mem, $zipi );
13 };
14
15 open (MANTA,">clientes.mas");
16 foreach (sort @mem ) {
17       print MANTA;
18 }
19 close (MANTA);
```

En esta pequeña modificación del programa anterior, y en la línea 8, se introduce una nueva estructura de **Perl**: el *array* o lista. Un *array* se indica con el símbolo @ (arroba), aunque a cada elemento de un array nos referiremos con la notación `$mem[0]`, ya que es un escalar. En general, tanto los operadores como las funciones en **Perl** se comportan de forma diferente si se trata de un entorno *escalar* o si se trata de un entorno de *lista* o vectorial. En este caso, el operador `localtime` devuelve una lista compuesta por los segundos, minutos y demas, tomándolo de la función time que devuelve el número de segundos transcurridos desde hace mucho tiempo (1970, para ser exactos). Además, estamos utilizando la lista `@zape` para contener el resto de los elementos de la hora, que no nos interesan (serían el día de la semana y cosas así). Por supuesto, también se podría haber hecho de otra forma, mucho más larga

```
@fecha= localtime(time);
$seg = $fecha[0];
...
```

Las matrices empiezan en **Perl** en 0 habitualmente, como sucede en C; pero esto se puede cambiar (usando la variable global `$[`). Hay que tener cuidadito con la variable `$anho`, que devuelve, al contrario de lo que se pudiera pensar, el número de años transcurridos desde 1900. Eso da lugar a todo tipo de pequeños efectos 2000, que todavía se ven por algún sitio Web. Por eso, en la línea 10, se le suma 1900, para que dé el año actual con cuatro cifras. Tal como en el lenguaje C, `$anho+=1900` equivale a `$anho=$anho+1900`;

En la línea 12 hay un nuevo operador, `push`. Este operador hace precisamente eso, achuchar un escalar o una lista al principio de otra lista (recordad que lista y *array* son prácticamente sinónimos). En este caso, estamos metiendo la cadena que hemos creado, `$zipi`, al principio de la matriz `@mem` (que, por cierto, tampoco hemos tenido que declarar ni dimensionar). Si queremos eliminar el primer componente de una matriz, se hace con el operador obvio, `$cadena = pop(@mem);`

En la línea 16 aparece una nueva orden de control de bucle: `foreach`, *para cada*, que repite el cuerpo del bucle para cada uno de los elementos de la lista que hay contenida entre paréntesis. Se puede abreviar por `for`, aunque también se puede comportar como lo hace en C. ¿Y cuál es la lista a la que se aplica? La que hemos creado anteriormente, pero por orden (de ahí el `sort`). En este caso la ordenará por orden alfabético, saliendo algo como esto

```
Uno 750 12:0 31/1/2000
otro 1402654 12:0 31/1/2000
otro mas 4020267 12:1 31/1/2000
y otro mas todavia 4040.25 12:1 31/1/2000
```

Sin embargo, dentro del bucle no parece que pase nada; y, ¿dónde diablos se ha metido la variable de bucle?. Lo que ocurre es que en estos casos **Perl** tiene una *variable por defecto*, `$_`, que es la variable de bucle por defecto y sobre la que actúan los operadores y funciones también por defecto. Es decir, que el bucle anterior equivaldría a

```
foreach $_ (sort @mem) {
    print MANTA $_;
}
```

Y aunque sé que ya estáis esperando que lo diga, se puede hacer de otra forma, esta vez menos **Perl**ística; utilizando bucles normales y corrientes

```
for ($i = 0; $i<=$#mem; $i++) {
    print MANTA $mem[$i];
}
```

si bien en este caso el fichero de salida no estará ordenado. En este caso se utiliza la construcción `$#<nombre de matriz>`, que devuelve el último índice existente para esa matriz. Ojo, se trata del último índice, no del número de elementos de la matriz; por eso los bucles en **Perl** usan un `<=` para terminar.

Ejercicios

- Hacer un programa que imprima su entrada estándar en orden inverso,empezando por la última línea y acabando por la primera. Se ejecutará con

  ```
  unix% cat <nombre del fichero> ¦invert.pl
  ```

  *Pistas*Usar la orden pop, que extrae el primer elemento de un array. *Reto* Hacerlo en una sola línea.
- $. es una variable que contiene la línea del fichero de la que se está leyendo. Teniendo esto en cuenta, crear un filtro (es decir, un programa que lea de entrada estándar y escriba en salida estándar, tal como el anterior) que imprima un fichero con el número de línea al principio de cada una.
- Hacer un histo(pro)gra, es decir, un programa que recoja una serie de valores numéricos (con valores reales entre 0 y 100), los distribuya en cinco cubos (del 0 al 20, entre 20 y 40, y así sucesivamente), calcule las frecuencias de cada uno de los cubos, e imprima barras horizontales cuya longitud es función de la frecuencia.

Recordando, que es gerundio

La orden `split`, matrices asociativas y matrices bidimensionales

Al final del día, nuestro político corrupto reflexiona. ¿De qué sirve tanto trabajo, sin una buena contabilidad consolidada? (Además, tiene que presentarle las cuentas al señor X a fin de mes). Idea, pues, el programa que aparece en el listado 4.

Listado : Programa totales.pl

```
1 die "Y el fichero de clientes, ein?\n" unless $ARGV[0];
2 while(<>) {
3   @linea = split;
4   @fecha=split(/\//,$linea[$#linea]);
5   $mesdia = "$fecha[1]-$fecha[0]";
6   $pasta=$linea[$#linea - 2];
7   $totalDia{$mesdia}+=$pasta;
8 }
9
10 foreach (sort keys %totalDia) {
11   print "Trinque total del dia $_ = $totalDia{$_}\n";
12 }
```

Este programa, aplicado sobre el fichero `clientes.mas`, (resultado de una incursión en la construcción de diversas viviendas para los cuerpos de seguridad del estado y sus departamentos de investigación y desarrollo, así como otros procedentes del mundo de la enseñanza) da el siguiente resultado (o algo parecido)

```
C:\PROGS\PERL>perl totales.pl clientes.mas
Trinque total del dia 3-24 = 598454.75
Trinque total del dia 4-25 = 1100987
Trinque total del dia 4-26 = 487775
```

Este programa empieza con una advertencia: "muere si no me metes un fichero como argumento". La orden `die` termina el programa con un mensaje; mientras que el condicional que lo sigue comprueba que exista al menos un argumento para el programa; la matriz `@ARGV` contiene los argumentos pasados al programa; de forma que `$#ARGV` dará el índice del último argumento, o sea que si es -1, indicará que no se ha pasado ningún argumento. Y otra forma de hacerlo sería

```
die "Sin argumento me lamento\n" if $#ARGV < 0;
```

O incluso

```
$ARGV[0] || die "Te has quedado sin argumentos\n";
```

que mira lo que hay a la izquierda del `||` (que es el "o" lógico), y si es cierto, ejecuta lo que hay a la derecha. Recuerda, en **PERL** hay más bla, bla.

El siguiente bucle, que comienza en la *línea 2*, tiene una extraña condición para que el bucle siga; sólo los dos ángulos enfrentados. Teóricamente, debería de haber un *filehandle* dentro de esos ángulos (como se ha visto en un ejemplo anterior), pero en este caso, se está tomando el fichero por defecto, que es el fichero que se introduce como argumento; en caso de que no se hubiera introducido ninguno tomaría entrada estándar, es decir, que habría que introducirle cada línea mediante teclado. A la vez, y como se ha visto, esa orden toma una línea del fichero y la deposita en la variable por defecto, aunque, ojo, no le quita el retorno de carro final. Hay que tener en cuenta que los paréntesis angulares sin argumento extraen elementos del array `@ARGV` usando `pop`, y abren un fichero con ese nombre; o sea que si hubiera varios nombres de fichero en la línea de comandos, los iría abriendo uno por uno y disminuyendo consecuentemente el tamaño de `@ARGV`; conclusión, que si necesitas `@ARGV` para algo, mejor que lo guardes antes de meterte en un bucle de esta guisa.

Sobre la variable por defecto actúa la orden `split` (una de las cosas más usadas en **PERL**), dividiéndola en una serie de cadenas separadas por espacios y depositando cada una de esas cadenas en un elemento de la matriz `@linea`. Y dese cuenta el venerado público de con qué facilidad hemos hecho algo que requeriría al menos 10 líneas de cualquier otro lenguaje. No hay que dimensionar matrices, no hay que recorrer la cadena caracter por caracter... ¡Nada![(3)]. Perdón, me he dejado llevar por el entusiasmo.

La fecha es, en todos los casos, la última cadena de la línea; es decir, que será el último elemento de la matriz (cuyo subíndice es siempre `$#<nombre-matriz>`), pero a su vez tendremos que dividirlo, esta vez por la barra de separación, para luego poder poner el mes delante y que salga todo bellamente ordenado por meses en vez de por días.

Esta cadena con la fecha, más la pasta, que está siempre 2 posiciones más a la izquierda (independientemente de la longitud de los nombres), se utiliza en la línea 7 en una **matriz asociativa**. Esta es otra de las características más potentes del **PERL**, se pueden usar matrices cuyo índice es una cadena cualquiera, no sólo números enteros positivos. Estas matrices asociativas encierran sus índices o *claves* entre llaves (¿os dáis cuenta del sutil mnemónico?). En esa línea, se le añade al componente de la matriz indexado por la fecha la pasta de la entrada correspondiente. Así, hasta que se termina el fichero de entrada.

Para imprimir el informe, tres cuartos de lo mismo que en el programa anterior, salvo que en este caso, una matriz asociativa completa se indica con `%`, en vez de la arroba.

Ejercicios

- Leer un fichero con el formato Primer_Apellido Segundo_Apellido, Nombre y escribirlo como
  Nombre Primer_Apellido Segundo_Apellido
- En el mismo fichero, presentar al final el número de veces que aparece cada apellido.
- Realizar un programa que, haciendo lo mismo que el manta.pl, lo haga en la mitad de líneas.
- Escribir un programa que haga lo mismo que la orden wc de UNIX, es decir, para cualquier fichero, contar el número de líneas, de palabras y de bytes, y presentarlo en 3 columnas con el nombre del fichero en la cuarta. Hacer que funcione para varios ficheros, introducidos en la línea de comandos.
- A partir del fichero de registro de visitas de un sitio Web, o log, poner cuántasveces han consultado las páginas desde dominios de primer y segundo orden, presentarlos por orden, con una opción que permita seleccionar primer o segundo orden.
- Realizar un programa que, a partir de una lista del tipo DNI Apellidos, Nombre genere un fichero de password, con el formato
  username:password:uid:gid:Nombre y Apellidos
  Calcular el username con la inicial del nombre y el primer apellido; si existe, usar el segundo apellido, y si existe también, añadir un número de orden. Para el password, encriptar el DNI; el UID se genera por orden alfabético a partir del número 1000, el gid es un entero común, y el nombre y apellidos es elindicado. Al terminarlo, ofrecerlo amablemente al administrador del sistema de tu Escuela o Facultad.

[v5]A partir de la versión 5 de **perl**, se pueden usar matrices bidimensionales, mediante un mecanismo que se llama referencias; este mecanismo no nos interesa ahora mismo, pero sí como podemos usarlo en nuestros programas, tales como el siguiente (totales-v5.pl):

Listado : Programa totales-v5.pl

```
1  #!/usr/bin/perl
2  $ARGV[0] || die "¡Dime el nombre del fichero de clientes, cohone!\n";
3  while(<>) {
4    @linea = split;
5    @fecha=split(/\//,$linea[$#linea]);
6    $pasta=$linea[$#linea - 2];
7    $totalDia[$fecha[1]][$fecha[0]]+=$pasta;
8  }
9
10 for ( $i = 1; $i <= $#totalDia; $i++ ) {
11   @dias = @{$totalDia[$i]};
12   for ( $j = 1; $j <= $#dias; $j ++ ) {
13     print "Trinque total del dia $j del $i = $totalDia[$i][$j]\n" if $totalDia[$i][$j];
14   }
15 }
```

Este programa es bastante parecido al anterior, salvo que, en vez de almacenarse los resultados en un array asociativo, se almacenan en un array bidimensional; la primera dimensión es el mes, y la segunda representa el día del mes. La línea 2 cambia simplemente para indicar otra forma de detectar si hay un fichero en la línea de comandos o imprimir un mensaje de error.

La línea 7 es la que usa una matriz bidimensional. Hasta aquí, todo normal; lo único relevante es que no hay que dimensionarla. Como primer índice de la matriz se usa el elemento 1 de `$linea`, es decir, el mes, y como segundo índice el primer elemento (el 0), es decir, el día.

Donde sí se nota un cambio es a partir de la línea 10; el bucle lo hacemos sobre una matriz normal y corriente. El problema es que, tal como sucede en las matrices bidimensionales en C, en **PERL** las matrices bidimensionales son en realidad una matriz de matrices, o mejor una matriz de referencias. Por eso, en la línea 11 lo que se hace es dereferenciar el elemento correspondiente y convertirlo en una matriz que podamos usar, la matriz `@dias`. En la línea 13, usando interpolación, se imprimen todos los elementos que son no nulos (`if $totalDia[$i][$j]`) la mayoría serán nulos , y los dejamos sin imprimir. [/v5]

Ejercicios

- Realizar un programa que lea un fichero organizado en filas y columnas, tal como este

  ```
  uno dos tres
  cuatro cinco seis
  siete ocho nueve
  ```

  y genere un documento HTML con los contenidos del fichero organizados en una tabla; cada elemento del fichero deberá ir en una celda diferente, así

uno	dos	tres
cuatro	cinco	seis
siete	ocho	nueve

  Aunque no es estrictamente necesario, usad una matriz bidimensional para hacerlo.

Purgando los pecados.

Usando el depurador

No es precisamente lo que tenía en mente nuestro político cuando decidió ver como aumentaba su capital con cada aportación de su clientela; o sea, que hacer un programilla en **PERL** para calcular sumas parciales, y sumas totales, no estaría mal; quizás incluso podía añadirle que sonara una campanita cada vez que sume cien talegos más. Así pues, hace el programa `purga.pl`, para aplicarlo sobre el fichero `clientes.mas`

```
while(<>) {
    split(/ /);
    print "Total parcial ", $total+=$_[1], ",\n";
}
```

que divide cada línea en sus componentes separados por espacios, y suma el segundo componente, índice 1 de la matriz. Pero cual no sería su desagradable sorpresa al ver que el dinero no aumentaba, sino que daba este resultado:

```
C:\PROGS\PERL>purga.pl clientes.mas
perl purga.pl clientes.mas
Total parcial 150000,
Total parcial 474999.75,
Total parcial 598454.75,
Total parcial 1154120.75,
Total parcial 1699441.75,
Total parcial 1699441.75,
Total parcial 1699441.75,
```

¡Diablos! 3 clientes no habían aportado su granito de arena a la grandeza del partido. ¿Cómo podía ser eso? Por mucho que lo ejecutaba no funcionaba de otra manera (como es natural), así que tuvo que aprender a manejar algo llamado purgante o purgador o algo así. Para ejecutarlo, hay que añadir la opción `-d` en la línea de comandos al llamar al **PERL** o la primera línea del script en UNIX:

```
perl -d purga.pl clientes.mas
Loading DB routines from perl5db.pl patch level 0.95
Emacs support available.
Enter h or 'h h' for help.
main::(purga.pl:1): while(<>) {
DB<1>
```

Este es el depurador o *debugger* de **PERL**, con el cual se puede hacer lo habitual en cualquier depurador, aunque quizás de una forma un poco menos amistosa Supuestamente, se puede usar desde emacs, aunque yo no he visto ninguna ventaja en ello. Sin embargo, desde `xemacs` es mucho más fácil de usar; se accede directamente desde la opción `Perl->Debugger` del menú.. También se puede usar el ActivePerl Development Kit, de ActiveState, pero todavía no lo he usado tampoco . Por ejemplo, se pueden poner puntos de ruptura con `b` (*breakpoint*), para examinar qué es lo que va mal:

```
DB<1> b 3
```

Al ejecutar el programa con `c` (continuar) o `r` (ejecutar), el programa parará en esa línea:

```
DB<2> r
main::(purga.pl:3): print "Total parcial ", $total+=$_[1],",\n";
```

imprimiendo la línea de que se trata. Entonces podemos examinar variables, matrices o lo que se tercie, con la orden `p`, de print o imprimir. Incluso, con la orden `<`, nos ejecuta un comando antes de pararse, por ejemplo:

```
DB<3> < print $_[1]
```

Al llegar a las líneas problemáticas, nos encontramos que

```
main::(purga.pl:3): print "Total parcial ", $total+=$_[1], ",\n";
Bacterio DB<5> c
Total parcial 1699441.75,
main::(purga.pl:3): print "Total parcial ", $total+=$_[1], ",\n";
DB<5>
```

De esta forma, observamos que en algunos casos, no se cumple que el segundo componente sea numérico: en un caso es una cadena, y en otro caso nada... Si nos fijamos en el fichero original, vemos que hay dos espacios tras Ofelia (será para hacerle sitio), con lo cual **PERL**, con toda su ilusión, divide la línea y no mete nada entre ellas. En realidad, esa línea es bastante rara, porque además aparece dos veces la cantidad, pero es igual. En cualquier caso, eso nos dice que en vez de usar el segundo componente desde el principio, debemos de usar el tercero desde el final, es decir, `$_[$#_-2]`. Con lo cual el programa funciona perfectamente, y nuestro político lo ha purgado.

En realidad, el debugger de **PERL** es bastante potente, a pesar de su presencia espartana, sobre todo porque incluye un intérprete y puede ejecutar sobre la marcha cualquier expresión. Incluso es aconsejable ejecutar un programa la primera vez desde este depurador, para corregir sobre la marcha cualquier problema.

El resto de las órdenes del depurador, que incluyen puntos de ruptura condicional, ejecución paso a paso (con `s` o simplemente dándole a enter), se consiguen con el comando de ayuda, `h`.

Pero ya puestos...

La orden `grep`

... se puede hacer de una forma más simple. Nuestro político se dio cuenta rápidamente que muchas de esas operaciones se pueden hacer en una sola línea, y así ahorra uno espacio en disco duro, que va caro. Por ejemplo, el programa `congrep.pl`

```
@zipi=<>;
grep( ( split(/ /) && print "Total parcial ",$total+=$_[$#_-2],"\n"),@zipi);
```

; en un par de líneas. En general, la función `grep`, que recuerda al `grep` del sistema operativo, pero no tiene mucho que ver con ella, sirve para procesar matrices enteras. Como primer argumente se le pasa una expresión, pero ya se ve que en perl el concepto de expresión es un poco amplio; de hecho, el programa anterior es uno de los ejemplos de programa críptico que los perl-hackers tanto conocemos y amamos.

Pero en fin, vamos a ello: la expresión, para empezar, está entre paréntesis; luego, consiste en dos partes separadas por `&&`; este operador nos garantiza el orden de evaluación, y además no evalúa la parte de la derecha si no ha podido ejecutarse la de la izquierda; en resumen, la expresión hace lo mismo que las dos líneas del interior del bucle anterior... Y, ¿qué hace `grep`, entonces? Simplemente, le asigna a `$_` cada uno de los componentes de la matriz que se le pasa como segundo argumento, y crea otra lista que tiene como componentes aquellos para los cuales la expresión es verdadera. Por ejemplo, podemos crear a partir del anterior otro programa que imprima sólo los que hayan pagado por encima de una pasta determinada, y además los meta en una matriz (`mailing.pl`):

```
@zipi=<>;
@ricos=grep( split(/ /) && ($_[$#_-2] > 200000) , @zipi );
print @ricos;
```

Ejercicios

- Hacer un programa que extraiga de un array todos aquellos elementos que aparezcan más de una vez, y los coloque en otro array. Hacerlo usando `grep`.

Regularizando la situación

Expresiones regulares: comparación y sustitución

Una forma todavía más directa de trabajar con estos ficheros de texto que tienen una estructura regular (que son la mayoría), es usar expresiones regulares. Una expresión regular es una forma de expresar gramaticalmente la estructura de cualquier cadena alfanumérica. Por ejemplo, una cadena compuesta por una letra inicial, con una letra o un número a continuación se podría expresar de la forma siguiente

letra {letra|número}*

donde `|` expresa una alternativa y `*` indica que puede aparecer 0 o más veces. Pues bien, estas expresiones regulares se utilizan enormemente en PERL (de hecho ya hemos usado una, aunque compuesta por un solo carácter, en la línea 4 del programa anterior), y cada vez que hemos usado `split`. Las expresiones regulares se usan para hacer comparaciones, es decir, hallar si un texto sigue o contiene una determinada expresión regular, y también para sustituir una subcadena que cumpla una expresión regular por otra cadena.

La expresión regular más simple es la propia cadena con la que se quiere comparar; es decir, una cadena coincidirá consigo misma. Las expresiones regulares en PERL siempre van encerradas entre `//`, o bien `m{}` (las llaves pueden ser sustituidas por cualquier otro elemento que no esté incluido en la expresión regular), y si no se indica nada, comparará la expresión regular con la variable por defecto, `$_`, es decir, que `/pepe/` será cierto si `$_` contiene íntegra la cadena `pepe`. El programilla

```
$_ = "pepeillo"; print "si" if /pepe/;
```

imprimirá

```
si
```

Las expresiones regulares usan símbolos convencionales para referirse a grupos de caracteres y otros para repetición o señales de puntuación. En algunos casos, si el símbolo significa algo dentro de la expresión regular (o en PERL), se precede por `\` (escape). Por ejemplo, `.` significa "cualquier carácter" dentro de una expresión regular; luego para referirnos al punto como tal, usaremos `\.`. También se usa `\/`,`\?` y `\*`, por ejemplo. Otras expresiones más complicadas incluirían repeticiones de símbolos; por ejemplo, `\w+` casaría con cualquier palabra (un carácter alfanumérico repetido uno o más veces).

Tabla 4: operadores de expresiones regulares

.	Describe cualquier carácter, excepto newline.
( )	Agrupa una serie de patrones en un simple elemento.
+,*,?	Coinciden con el elemento al que preceden repetido 1 o más veces, 0 o más, ó 0 ó 1. Seguidos por {N,M}, indican el número mínimo y máximo que debede aparecer; {N} significa exactamente N veces; {N,}, como mínimo N veces.
[..]	Indica una clase de caracteres, [^...] niega la clase, - indica un rango decaracteres, como [a-z].
(..|..|..)	coincide con una de las alternativas.
\w,\W	coincide con los alfanuméricos, \W con los no-alfanuméricos.
\s,\S	con los espacios en blanco, \S con los que no lo son.
\d \D	con los numéricos, \D no-numéricos.
\b,\B	con límites de palabra, \B con el interior de una palabra.
$,^	con el final de una línea o cadena y con el principio.

Para agrupar símbolos se usa el paréntesis, que además sirve para indicarle a PERL que con lo que coincida con la expresión en su interior se va a hacer algo. Para empezar, se asigna a la variable `$&`; pero además, podemos asignar a otra variable el resultado de la comparación; en general, una comparación con una expresión regular que incluya paréntesis devuelve una lista con todo lo que coincida con el contenido de los paréntesis

```
$zipi = "Pepeillo Gonzalez MacKenzie 8000";
@parejas = ($zipi =~ /(\D+) (\d+)/);
```

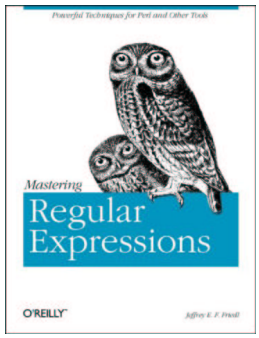
(el nombre de la variable y el símbolo `=~` se pueden suprimir si la comparación se hace sobre la variable por defecto `$_`); `@parejas` contendrá `("Pepeillo Gonzalez MacKenzie",8000)`, ya que la primera expresión regular indica "uno o más caracteres no numéricos", mientras que la segunda representa "uno o más caracteres numéricos".

Por ejemplo, el fichero de paganinis usado en ejemplos anteriores anterior tiene la estructura siguiente: una o más palabras, separadas por un espacio, una cantidad (números y puntos), una hora (números y dos puntos) y una fecha (números y /). Esto se dice en PERL mediante la expresión siguiente`(\D+) (\d+\.?\d+) (\S+) (\d+)\/(\d+)\/(\d+)`que puede parecer un poco críptica (y en realidad lo es), pero cuyo significado se puede resolver mirando la tabla 4. Con esta modificación, el bucle central del programa `totales.pl` se queda reducido a

```
while(<>) {
    ($paganini, $pasta, $hora, $dia, $mes) = /(\D+) (\d+\.?\d+) (\S+) (\d+)\/(\d+)\/(\d+)/;
    $totalDia{"$mes-$dia"}+=$pasta;
}
```

Más compacidad no se puede pedir. Además, ya de camino, nos vamos ahorrando algunas variables. En la primera línea del interior del bucle se asigna la parte de la cadena que coincide con lo descrito en el interior de los paréntesis a cada una de las variables, es decir, divide la línea en cinco campos, cada uno de los cuales tiene un tipo diferente. Hay cinco pares de paréntesis, para cinco variables. Veamos cada expresión regular por partes.

Las expresiones regulares se usan en casi todos los lenguajes de programación; el ligro *Mastering Regular Expressions: Powerful Techniques for Perl and Other Tools* , de Jeffrey E. Friedl (Editor), Andy Oram (Editor), el libro del búho. Es un libro avanzado, pero útil para quien quiera descifrar el arcano arte de las expresiones regulares



La primera expresión regular es `\D+`. Como se ve en la tabla 4, `\D` coincide con todo lo que no sea numérico, y en particular letras y espacios. Este campo es problemático, porque puede incluir una o varias palabras; sin embargo, sabemos que el siguiente campo comienza por un número; por tanto, incluiremos en este campo todo lo que haya hasta que encontremos un número. Se puede insertar el código siguiente `print $paganini;` en el bucle para ver qué se incluye dentro de ese campo.

La siguiente expresión regular, `(\d+\.?\d+)`, no describe otra cosa que un número real en notación de coma flotante, es decir, una o más cifras `(\d+)`, seguidas o no por un punto (`\.?`, no olvidar que el punto tiene significado especial en las expresiones regulares), que a su vez debe de estar seguido por una o más cifras `(\d+)`. Esta expresión regular coincide además con aquellos números sin punto enmedio.

Y la siguiente `\S+` coincide con la hora, aunque quizás podría coincidir con cualquier cosa, simplemente coge todo lo que haya entre los dos espacios. En realidad, si pusiéramos toda la expresión regular anterior como`(\S+) (\S+) (\S+) (\S+)` funcionaría perfectamente.

Y para terminar,`(\d+)\/(\d+)\/(\d+)`coincide con el día, el mes y el año, que están separados por una barra diagonal.

Con estas expresiones regulares se pueden construir programas superpotentes, que convierten casi cualquier texto en casi cualquier otro. Incluso, si uno se atreve (nuestro político corrupto no creemos que se atreva) un analizador sintáctico de un lenguaje de alto nivel, aunque sea simplificado. Otra utilidad es hacer filtros de correo electrónico, o de noticias de USENET (de hecho, el killfile o fichero que incluye todas las expresiones regulares de mensajes que deben de ser borrados, usa expresiones regulares). También se pueden usar expresiones regulares en archie y en la Web; el problema es que la mayoría de las veces usan convenciones diferentes, pero en cualquier caso, nunca viene mal saberlas.

El político corrupto decide no contarle al señor X todo el dinero obtenido (por no mencionar a Hacienda), y poniéndose a trabajar sobre los ficheros generados anteriormente (como `cliente.mas`), elabora el siguiente programilla (`changec.pl`)

```
#!/usr/local/bin/perl -p
s/(\d+\.?\d+)/$&*0.85/e;
```

que ni siquiera pongo en un cuadro aparte, porque no merece la pena.

En este programa se introducen novedades desde la primera línea. Para empezar, se usa PERL con un switch en la línea de comandos, `-p`. Este switch indica que alrededededor de lo que hay debajo hay que sobreentender el siguiente bucle

```
while(<>)
{
    s/(\d+\.?\d+)/$&*0.85/e;
    print;
}
```

Tabla 5: Switches o interruptores que se pueden usar en la línea de comandos

-n	Incluye el bucle while(<>) {}alrededor de las órdenes del fichero.
-p	Incluye el bucle while(<>) {print;} alrededor de las órdenes del fichero.
-a	Modo autosplit, es decir, que al principio del bucle anterior incluye la orden split.
-d	Modo de depuración. En vez de ejecutar el programa, se ejecuta el depurador sobre el programa.
-i[extensión]	En combinación con -p y -n, en vez de imprimir en salida estándar, trabaja sobre el mismo fichero, creando una copia de seguridad con la extensión que se indica,

; es decir, un bucle que abre y lee del fichero que se pasa en la línea de comandos, ejecuta las órdenes encontradas en el fichero, y al final imprime la variable por defecto. Si el switch fuera `-n` en vez de `-p` no imprimiría.

También se incluye la nueva orden `s///[switches]`tomada, como otras órdenes, del editor de UNIX `sed`. En concreto, esta orden lo que hace es sustituir la expresión regular encontrada entre los primeros `//` por la expresión en los segundos. En este caso actúa sobre la variable por defecto, pero en general`$zipi="pepeillo"; $zipi=~ s/p/q/g;`daría "qeqeillo". El switch `e` que aparece al final de la orden en el programa indica que se tiene que evaluar la expresión que aparece en la segunda parte de la orden; el `g` que aparece aquí indica que se tiene que hacer una sustitución global, es decir, que tienen que sustituirlo todas las veces que aparezca en la primera expresión, no solo una vez.

Por último, la variable `$&` contiene la cadena que coincida con la expresión regular en la primera parte de la orden; los paréntesis indican qué parte de la expresión hay que asignar a tal variable. Si hay varios paréntesis, las cadenas correspondientes serán asignadas a las "variables" `\1, \2` (dentro de la orden `s`), o `$1, $2...` fuera de la expresión (estas variables sólo funcionan un ratito, asi que no van a estar disponibles hasta que uno quiera; en caso de duda, es mejor asignarlas inmediatamente a otra variable.

Toda la información sobre expresiones regulares está en la página de manual `perldoc perlre`.

Ejercicios

- Dado un fichero que contenga varias líneas en el formato siguiente:

  ```
  Apellido 1 Apellido 2, Nombre          Nota como un número real
  ```

  poner los nombres en el formato Nombre Apell1 Apell2 Nota como Calificación (Notable, Sobresaliente, etc), ordenándolos por orden de nota.
- Dado un fichero que contenga URLs, o direcciones internet, generar otro que incluya alrededor de las mismas una referencia del tipo URL.
- Hacer un programa que imprima todos los ficheros de cabecera incluidos en un programa en código fuente C. Estos tienen el formato #include <nombre-de-fichero.h>. Comprobarlo sobre algún fichero el directorio /usr/include (en UNIX), o cualquier otro que pille a mano.
- En un programa en código fuente C o PERL, imprimir las líneas en las que comience un bucle for, e imprimir la variable de bucle que se usa.
- En un programa en código fuente C, sustituir todas las constantes definida usando la directiva de precompilación #define por su valor, definido en la misma orden.

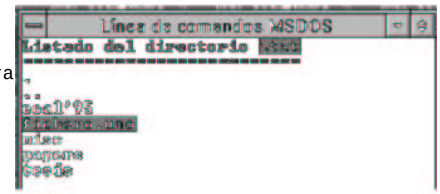
Presentando lo impresentable

Usando subrutinas y secuencias de escape ANSI

Tanto trabajo está acabando con la salud de nuestro político. La campaña electoral está cerca, tiene que inaugurar unas cuantas obras inacabadas todavía, y decide descargarse de un poco de trabajo. Para ello contrata a un militante de base como experto de Informática con cargo a los fondos reservados. Este se encuentra un trabajo ingente, y para empezar decide facilitar un poco su labor escribiendo una utilidad que le permita sacar directorios un poco más monos. Y, ni corto ni perezoso ("Este chico llegará lejos", comenta nuestro político) escribe el programa `dir.pl`,que aparece en el listado siguiente.

Listado : Programa dir.pl

```
1 #!/usr/bin/perl
2
3 sub inverso {
4 #Añade la secuencia ANSI de escape para video inverso
5       return " [7@_";
6 }
7
8 sub negrita {
9 #Añade la secuencia ANSI de escape para letra resaltada
10      return " [1@_";
11 }
12
13 sub borra {
14 #Borra pantalla
15      print  " [2J";
16 }
17
18 sub normal {
19 #Restablece modo normal
20      print " [0m";
21 }
22
23 &borra;
24 $dirSep = "/";                              #Solo para MS-DOS
25
26
27 if ($ARGV[0] eq "" ) {
28      $dir = $dirSep;
29 }else {
30      $dir = $ARGV[0];
31 }
32
33 #Abre directorio
34 opendir( DIR, $dir );
35 while( $fichero = readdir( DIR ) ) {
36      push( @ficheros, $fichero );
37 }
38 closedir( DIR);
39
40 #Cabecera del listado
41 $msg = "Listado del directorio ".&inverso( $dir );
42 print "$msg\n";&normal;
43 print "=" x (length($msg) - length(&inverso(""))); print "\n";
44
45 if ( $dir !~ m!\\$! ) {
46      $prefijo = $dir.$dirSep;
47 } else {
48      $prefijo = "$dirSep";
49 }
50
51 foreach (sort @ficheros ) {
52      print &inverso( $_ ) if -f "$prefijo$_";
53      print &negrita( $_ ) if -d "$prefijo$_";
54      &normal;
55      print "\n";
56 }
```



En este programa ya estamos hablando de palabras mayores. Para empezar, es un poco más largo. Para continuar, es sólo para MS-DOS o Win95, pero se puede adaptar más o menos fácilmente a cualquier otro sistema operativo cambiando el separador de directorios contenido en la variable `$dirSep` (: para el Mac, / para UNIX). El programa saca un listado del directorio, ordenado en orden alfabético, resaltando los directorios con un tipo de letra más brillante, y los ficheros en video inverso.

Se le llama de la forma siguiente

```
C:\> perl dir.pl <nombre del directorio>
```

Si no se da ningún nombre de directorio, toma el directorio raíz; para el directorio actual hay que introducir

```
perl dir.pl .
```

.

Veremos primero el truco de sacar las cosas en diferentes tonos, y que se expresan en el programa con extraños signos (para obtener la flechica p'atrás, se pulsa control-P, y sin dejar de pulsarlo, control-[; difícil, pero es así). El programa usa secuencias de escape ANSI, que son órdenes estándar para trabajar con terminales; son tanto aplicables a terminales UNIX (por ejemplo, los tipo vt-100 o ansi), como MS-DOS; en este último habrá que añadir la línea

```
device=c:\dos\ansi.sys
```

al fichero `config.sys`. Con estas secuencias de escape se puede imprimir en diferentes lugares de la pantalla, borrar líneas o la pantalla completa, y cambiar, como aquí, los atributos de partes de la pantalla, todo ello de forma más o menos independiente del sistema. Es algo así como el interfaz gráfico de usuario de los pobres.

El siguiente concepto que introduce este programa es el de subrutinas. Las subrutinas en PERL se pueden declarar en cualquier parte del programa, al principio o al final. Para declararlas, simplemente se precede el nombre de la subrutina con la palabra `sub` y se le sigue del bloque de instrucciones que se van a ejecutar. Pueden no utilizar parámetros (como `borra` y `normal`). En este caso se llaman precediéndo el nombre con el símbolo `&`, o bien escribiendo `do borra`. También se puede llamar a una subrutina con parámetros, en cuyo caso forman todos parte de la matriz `@_`, como se ve por ejemplo en la subrutina `negrita`. Los elementos de esta matriz son, como es natural, `$_[0]`, `$_[1]` y así sucesivamente, aunque se pueden asignar todos de una vez a parámetros locales de la subrutina, de esta forma`local($zipi, $zape) = @_;`En esta sentencia se declaran simultáneamente `$zipi` y `$zape` como variables locales y se les asignan los dos primeros parámetros que se le han pasado a la subrutina. Si no se declararan como `local`, las variables serían visibles por todo el programa, como sucede en el resto de las variables en PERL.

[v5]En realidad, `local` crea variables globales pero con valores locales; es decir, se crea una variable que se puede ver en todo el programa, pero cuyo valor se guarda al principio de la subrutina y se recupera al final de la subrutina. En la versión 5 se introduce la palabra clave `my`, que declara verdaderas variables locales. En la versión 5, por tanto, se suele preferir `my` para declarar las variables de las subrutinas. [/v5]

La lectura del directorio se hace a partir de la línea 31; para eso utiliza funciones similares a las que hay en C o Pascal; una funcion, `opendir`, "abre" el directorio, y las otras van leyendo secuencialmente los nombres de todos los ficheros de ese directorio y metiéndolos en la matriz `@ficheros`.

Tabla : Operadores de comprobación de ficheros

-e	Comprueba si el fichero existe o no
-f	Comprueba si es un fichero normal
-d	Comprueba si es un directorio
-T/-B	Comprueba si es de texto o binario. Pero no funcionan fuera de Linux.

Luego, a partir de la línea 49, y utilizando los operadores `-f` y `-d`, que comprueban si el fichero cuyo nombre viene después es un fichero normal o un directorio, se va imprimiendo con los atributos correspondientes. Estos son un ejemplo de los operadores de fichero que se utilizan mucho en PERL (ver la tabla 4), y que vienen heredados de comandos similares en los shell de UNIX (como `csh` y `bash`), nosotros tenemos la suerte de poder usarlos en ms-dos también. Otros operadores figuran en la tabla 4.

Decodificando

Usando `pack` y `unpack`

En realidad, la contratación de este nuevo genio de la informática, al que ya le está buscando también colocación en la empresa que controla los ordenadores de la campaña electoral, le ha dejado algún tiempo libre a nuestro [censurado] político. Para emplear este tiempo libre, decide conectarse con eso que le ha dicho su amigo Borrell, que se llama internez o algo así; dentro de la internez, y de una parte especialmente yanqui llamada usanez o algo así, hay unos grupos de discusión de temas de máxima actualidad. En concreto, un grupo denominado `alt.binaries.pictures.erotica.politics` publica interesantes fotos de políticas de la oposición en actitud de debate (al menos eso es lo que le han contado).

Pero su gozo se encuentra en un pozo cuando sólo ve letras y más letras en los mensajes que le van llegando. Su informático residente le explica que vienen codificadas con un programa denominado `uuencode`, y que tiene que salvarlas cada una por separado y decodificarlas después, antes de verlas. Con la pericia que dan las ganas, escribe el siguiente programa, `uudec.pl`.

Listado : Programa uudec.pl

```
1 #!/usr/bin/perl
2
3 while( <$ARGV[0]>) {
4
5     open( IN, "<$_" );
6     while (<IN>) {
7       if ( /begin \d+\s+(\S+)/ ) {
8           $fileName = $1;
9           open ( OUT, ">$fileName );
10      }
11      print OUT, unpack("u",$_) if /^[MC]/;
12     }
13 }
14 close OUT;
```

Este programa utiliza la orden `unpack` para decodificar cadenas, y las escribe en salida estándar. La descripción es incluso más corta que el programa. Los mensajes en ciertos foros de USENET vienen divididos en varias partes, cada una de las cuales contiene un pedazo de un fichero codificado con `uuencode` (puede ser un .GIF, un .WAV o cualquier otro). Para usar el programa, se salvan todas las partes del mensaje correlativamente en un fichero. El programa busca primero el nombre del fichero codificado, que aparece en una línea de la forma`begin 644 nomfich.gif`, abre un fichero con ese nombre, y va escribiendo en él las cadenas decodificadas.

Para llamar a este fichero, se escribe

```
UNIX% uudec.pl fichero.uu
```

y se crea el fichero codificado, que luego podrá uno contemplar con su visor de GIFs preferido (o escuchar con la SoundBlaster). Se pueden dar comodines en la línea de comandos; el bucle lo irá abriendo uno por uno.

En realidad, la orden `unpack` decodifica de formatos de codificación internos a cadenas; la orden `pack` hace exactamente lo contrario. La utilidad de todo esto se me escapa un poco, salvo que, por ejemplo, se quieran almacenar números reales de forma que pueda leerlos directamente un programa en C. En ese caso, se escribiría

```
print pack( "f", $zipi);
```

Un número real escrito así se podría leer directamente con la orden `read` del C (pero, ojo, no con la `scanf`, que sirve solamente para ficheros de texto). Otra forma de usarlo es para formatear ficheros de salida; usando

```
print
pack("A7",$cadena)
```

, haremos que la salida ocupe justamente 7 caracteres,ni más mi menos; así aparecerá todo en bonitas columnas. Por ejemplo, nuestro archiconocido memoria.pl se podría convertir en este memoria-col.pl

Listado : Programa memoria-col.pl

```
1 #!/usr/bin/perl
2 until(0) {
3       print "Cliente, cantidad\n";
4       chop($_ = <>);
5       last if !$_;
6       ($paganini, $cantidad) = split(/,/,$_);
7       $comision = $cantidad * 0.25;
8       ($seg, $min, $hora, $dia, $mes, $anho, @zape) = localtime(time);
9       $mes++;
10       $zipi = pack("A16",$paganini).pack("A7",$comision)."$hora:$min $dia/$mes\n";
11       push( @mem, $zipi );
12 };
13
14 print @mem;
```

Este programa, sobre el fichero siguiente

```
uno, 1605168
este es el dos, 166166
este puede ser el tres, 1616136
```

produciría la siguiente salida

```
uno            401292 19:54 25/7
este es el dos  41541.519:54 25/7
este puede ser e404034 19:54 25/7
```

(dependiendo, claro está, de cuando se hayan producido las susodichas transacciones cambiarán horas y fechas).

Todo bajo control

Usando tuberías para controlar otros programas

Cuando nuestro político corrupto descubre que UNIX tiene ya preprogramadas muchas de las funciones que le hacía el militante de base, lo despide y decide usar esos programas de UNIX desde, por supuesto, un programa en PERL; además, decide tirar Windows95 a la basura e instalalarse Linux. Por ejemplo, para contar las palabras que hay en un fichero basta con usar la orden `wc` de UNIX. Pero si se presenta en modo bonito, puede salir el siguiente programa, llamado `wc.pl`

```
#!/usr/bin/perl
while (<*>) {
    ($zipi, $lineas, $palabras, $bytes, $nombref) = split(/\s+/,`wc $_`);
    print "El fichero $nombref tiene $palabras palabras y $lineas lineas\n";
}
```

La primera línea del bucle usa una bonita capacidad de PERL, llamada globbing o generalización. Lo que hace es que, si se pone algún comodín de ficheros del tipo `*` repetirá el bucle para cada fichero que cumpla ese patrón, asignando el nombre del fichero a la variable por defecto. Ese nombre se le pasa al comando wc, usando ``, es decir, comillas invertidas, que ejecutan un comando en UNIX devolviendo su salida estándar. Esa orden completa agarra la salida del comando, del tipo

```
lineas palabras bytes nombre del fichero
```

y la parte en sus componentes. Como las diversas palabras están separadas por varios espacios o tabuladores, se una el patrón `\s+`. Al principio de la línea hay una serie de espacios, por lo que el primer componente devuelto no tendrá nada.

Hay otra forma diferente de controlar programas en UNIX, esta vez interactivamente, usando los denominados pipes o tuberías, que se suelen indicar con el carácter `|`. Por ejemplo, la orden `open( FICHERO, "|pepe" );` abre una tubería a la orden `pepe`, que a partir de entonces ejecutará los comandos que reciba del programa en PERL, ya que éste programa le suministrará la entrada estándar. Esto se puede usar de la siguiente forma (si se dispone del programa `gnuplot`):

```
open( GNU, "|gnuplot" );
print "Fichero a trazar? "; chop($fileN = <STDIN>);
print GNU "plot \"$fileN\" with lines 1\n";
```

que abrirá una tubería al programa y le enviará la orden para trazar un fichero con líneas. Con este tipo de órdenes y con `gnuplot`, y con unas pocas ganas, se pueden hacer animaciones. Sin embargo, hay que tener en cuenta que sólo se podrá hacer con programas que acepten comandos por su entrada estándar.

El cofre de la sabiduría

Como usar los módulos y librerías en programas en perl

[v5]

Como nuestro político corrupto™ ya es todo un guru de perl, la secretaría de prensa del partido decide encargarle el seguir en la Internet todos los sitios web que hablen del Señor X, para luego poder así denunciar la conspiración judeo-masónica-telefónica-mundial (o séase, de El Mundo) en contra de la democracia y de sus figuras más prominentes. Por eso, decide hacer un programa que cuente todas las páginas en las que aparece la cadena "Señor X"; así luego puede hacer un gráfico con su evolución y ver si la cosa va a más o a menos. Decide usar el buscador Altavista, ver el número de páginas que devuelve, e imprimirlo. Para ello, hace el siguiente programa (altax.pl):

Listado : Programa altax.pl

```
1  #!/usr/bin/perl
2
3  use LWP::Simple;
4  use URI::Escape;
5
6  my $codedQuery = uri_escape( '"Señor X"' );
7  my $url="http://www.altavista.com/cgi-bin/query?pg=q&sc=on&hl=on&q=$codedQuery&kl=XX&stype=stext&
8  print "Conectando a $url\n";
9  my $respuesta = get( $url );
10 my ($encontrados) = ($respuesta =~ /\S+;(\d+) pages found/gs);
11 print "Encontradas $encontrados respuestas\n";
```

Este programa imprimiría algo así como

```
Conectando a http://www.altavista.com/cgi-bin/query?pg=q&sc=on&hl=on&q=%22Se%F1or%20X%22&kl=XX&stype=stext&search.x=27&search.y=7
Encontradas 49 respuestas
```

Este programa, tal cual, seguro que no funciona en un ordenador. Usa un par de librerías (o "bibliotecas" para los puristas), que no se encuentran en la instalación estándar de PERL: la `LWP::Simple` y la `URI::Escape`. Lo primero que habrá que hacer es bajarse e instalar estas páginas.

Si está uno en Windows, es bastante fácil. Con la instalación de PERL viene un programa, `ppm`, que sirve directamente para bajarse e instalar módulos. Simplemente tecleando ppm, y luego

```
ppm> install  LWP::Simple
ppm> install  URI::Escape
```

el programa baja, compila si es necesario, y prueba los dos módulos. A veces esto que parece tan simple no lo es, porque necesita compiladores externos y otra serie de cosas, pero si hay suerte, todo irá bien. Con `ppm` se pueden usar otras órdenes, tales como

```
ppm> search WWW
```

que dará una lista de todos los módulos que incluyan WWW en su nombre. En este caso, además, las dos librerías vienen en un sólo paquete, el libwww-perl, que se puede bajar uno sólo tecleando `install libwww-perl`

En Linux y otros Unixes, es un poco más complicado, pero no mucho. Todos los módulos para PERL se crean y empaquetan de una forma estándar. Tras localizarlos, habitualmente en alguno de los sitios CPAN (http://www.cpan.org), se hace lo siguiente:

```
[yo@localhost tmp]# tar xvfz libwww-perl-5.47.tar.gz
libwww-perl-5.47/
libwww-perl-5.47/t/
(más cosas...)
[yo@localhost tmp]# cd libwww-perl-5.47
[yo@localhost libwww-perl-5.47]# perl Makefile.PL
(aquí comprobará que todas las librerías que se necesitan previamente
están instaladas, y crea un Makefile)
[yo@localhost libwww-perl-5.47]# make
[yo@localhost libwww-perl-5.47]# make install
(y también, si se quiere)
[yo@localhost libwww-perl-5.47]# make test
```

En este caso, también se acaba antes bajándose, para empezar, el módulo `CPAN.pm`, con el cual, tecleando `perl -MCPAN -e shell;` tiene uno un entorno similar al que hemos mencionado antes para Windows. Eso sí, para instalar algo hará falta, tanto en este caso como en el anterior, ser superusuario.

Las librerías no sólo instalan el código, instalan también los manuales correspondiente. En este caso, se puede consultar el manual escribiendo `perldoc LWP`; y merece la pena hacerlo, porque nos encontraremos muchas cosas interesantes. Como indicamos en la sección sobre el zen del PERL, merece la pena conocer bien las librerías antes de ponerse a hacer cualquier cosa en PERL, porque en la mayoría de los casos se encontrará con la mitad del trabajo hecho.

Además, en este caso las librerías son dos palabras separadas por `::`; esto es porque los módulos están organizados en una especie de jerarquía. Cada módulo puede estar dentro de una jerarquía o no, dependiendo de lo que quiera el autor y dependiendo de su funcionalidad. En este caso, `Simple` está dentro de LWP (lib-www-perl), porque va de eso; pero hay muchas más de la misma familia (por ejemplo, `LWP::UserAgent` un módulo más flexible para bajarse páginas). Lo mismo ocurre con el otro módulo.

Ya que tenemos las librerías instaladas, podemos ir al programa. Las dos líneas que empiezan por `use` importan las librerías. En realidad, hacen algo más que eso: traen todos los identificadores de la librería al espacio global de nombres, de forma que podemos usar las subrutinas y variables de la librería como si hubieran sido declaradas en el mismo programa. En realidad, en este caso usamos sólo una función de cada librería, así que, para no contaminar el espacio de nombres, podríamos haber escrito

```
use LWP::Simple qw(get);
use URI::Escape qw(uri_escape);
```

con exactamente el mismo resultado. Esta expresión, aunque pueda parecer un poco rara, usa el mecanismo quote del PERL, que sirve para definir matrices, cadenas y otras cosas sin necesidad de escribir demasiado; o séase, que se ahorra uno comillas, comas y cosas por el estilo. `qw( a b c)` equivaldría a `('a','b','c')`. Por supuesto, el operador `qw` y todos los demás por el estilo (ver la página de manual perlop) pueden usarse en cualquier punto del programa PERL.

Hasta ahora, en realidad, no hemos hecho nada, salvo importar las librerías; el programa de verdad empieza en la línea 6, que requiere un poco de explicación. Usa la subrutina `uri_escape` del módulo `URI::Escape`, que sirve para codificar cadenas que incluyan caracteres no alfanuméricos de forma que se pueda incluir dentro de un URL o dirección Web, tal como la que usan los buscadores. En este caso, la cadena contiene espacios, comillas y la letra ñ, que no puede ir tal cual en un URL. Por eso se codifica, quedando así: `%22Se%F1or%20X%22`. Generalmente, cuando se vayan a hacer peticiones a buscadores desde un programa, habrá que hacer este tipo de codificación.

El URL que se va a solicitar se compone en la siguiente línea. El único truco aquí es ver que la petición que se hace a Altavista va dentro de la variable `q` (no tienes más que hacer una búsqueda de cualquier palabra, y ver dónde aparece en el URL de la respuesta); eso se puede ver mirando al código HTML de la página, si no es demasiado complicado. En este caso, encontramos una línea HTML que dice:

```
<input type=text name=q size=35 maxlength=800 value="">
```

con lo cual, se ve que la variable `q` es la que contiene la petición del usuario. Algo similar se puede hacer en casi todos los demás buscadores.

La dos líneas siguientes imprimen un mensaje, y se bajan efectivamente el URL. La función `get` está sacada del módulo `LWP::Simple`, y simplemente se baja una página y la mete en una variable. También se puede imprimir usando `getprint` o meterse en un fichero usando `getstore`. Como siempre, más información escribiendo `perldoc LWP::Simple`.

Finalmente, usando expresiones regulares (¿recuerdas las expresiones regulares? Mira en Regularizando la situación) se extrae el número de páginas que se han encontrado, y se imprime. Y nuestro político corrupto, finalment, comprueba que últimamente ya casi ni se habla del tema.

Ejercicios

- Usando el módulo `Benchmark`, medir cuanto tarda en ejecutarse el código anterior. Programar también una petición similar para otro buscador, y comparar ambos resultados.
- Localizar una librería que sirva para hacer gráficos bidimensionales en **PERL** (pista: es parte de la jerarquía `Chart`), y usarla para hacer el histograma que se propuso en el tercer ejercicio del bloque segundo.

[/v5]

Algunos consejos

Algunos trucos para programar en PERL

Ya que hemos aprendido todo lo que debíamos aprender sobre PERL, no está de más dar unos cuantos consejos para realizar buenos programas.

Mucho cuidado con los espacios, con los puntos y comas y los demás caracteres que no están ahí de adorno

Un espacio entre el `$` y el nombre de una variable, dará un error de sintaxis. O un espacio entre el `#` y el `!`, que dará un error extraño, ya que el shell interpretará el resto de la línea como un comentario. Y no olvidemos nunca el `;` al final de las sentencias, ni las llaves alrededor de cualquier bloque, como por ejemplo en los bucles.

Sigue siempre la via PERL

Aunque nos permita recurrir a lo que conocemos de otros lenguajes, PERL permite hacer las cosas de forma mucho más compacta, elegante y a veces rápida. Por ejemplo, en vez de

```
$zipi= $matriz[0];
$zape = $matriz[1];
```

en PERL se puede hacer:

```
($zipi,$zape) = @matriz;
```

O para dividir algo como "González McKenzie, Pepeillo", lo más rápido es `($Ap1, $Ap2, $Nombre) = /(\S+) (\S+), (\S+)/;` (previa asignación a `$_`); en vez de utilizar `split` dos veces. O `@fichero= <FILE>;` en vez de usar un bucle para leer el fichero línea a línea. Los operadores de asignación de matrices, las variables asociativas y las expresiones regulares son fundamentales en casi cualquier programa PERL; dominarlas es alcanzar el Zen del PERL

Aprovéchate del depurador

Tiene órdenes bastante simples, y permite hacer muchas cosas; de hecho, todo lo que se puede hacer con PERL. Aunque parezca un poco espartano, es la mejor opción para empezar a crear un programa.

Conoce tus bibliotecas

Aparte de las muchas librerías estándar que trae PERL, hay otras, disponibles sobre todo en el CPAN, para hacer casi todo, desde acceso a base de datos hasta realizar tablas astrológicas. El conocer qué librerías hay disponibles, o mirar antes ahí, ahorrará mucho trabajo.

Preguntas frecuentemente preguntadas

Preguntas que suelen enviar por correo electrónico, y que suelo contestar.

¿Cómo se pueden hacer CGIs en PERL?¿Qué diferencia hay entre CGI y PERL?
CGI no es un lenguaje, es una forma de escribir programas que se pueden escribir en cualquier lenguaje; en especial, se pueden escribir en PERL; que está especialmente diseñado para ello. Escribir un CGI en PERL es tan simple como esto:

```
print "Content-type: text/html\n\n";
print "Hola";
```

. Por supuesto, habrá que colocarlo en la zona correspondiente del servidor para que sea accesible como CGI. Otra forma, a partir de la versión 5, es:

```
use CGI;
print header, start_html, title('titulo'),end_html;
```

que usa el módulo estándar CGI.pm, que sirve precisamente para escribir CGIs. Más información, escribiendo `perldoc CGI`

¿Cómo se puede acceder a una base de datos en PERL?¿Qué bases de datos puedo usar?
Lo más fácil es usar el módulo DBI (que, como todos los demás, está en el CPAN), con un driver específico para el gestor de bases de datos, o si hay ninguno específico, y la BD está accesible usando ODBC, pues con ODBC. En WinXX hay un módulo alternativo, específico para ODBC, denominado `Win32::ODBC`. Con cualquiera de estos dos módulos, se puede usar con cualquier SGBD; en Linux/Unix, se puede usar con MySQL o MiniSQL, o Postgres; y en Windows se puede usar con Microsoft SQL server combinado con ODBC. Eso sí, como es inherente a las conexiones ODBC, es más bien tirando a lento.

Tengo este error en un programa en PERL. Por favor, ¿puedes echarle un vistazo a ver qué le pasa? O, una variación de la anterior: por favor, ¿puedes mandarme un programa que haga tal cosa?
Pues generalmente, no. Bastante tengo con hallar los errores de mi código, para ponerme a hallar los errores ajenos. Y en cuanto a la segunda, si lo tuviera y quisiera darlo, estaría entre los ejemplos, y si no está, en fin, hay gente que se gana la vida programando el PERL, echadles una mano... Ojo, esto no significa que no podáis preguntarme dudas, estoy encantado de contestarlas. Pero no corregir programas ajenos, o hacer programas para alguien (aunque por un módico precio, podemos llegar a un acuerdo). En estos casos, lo mejor es que escribáis a alguna lista de correo que trate de PERL, tal como PERL-ES; seguro que habrá gente encantada de ayudaros.

¿Funcionan exactamente igual los programas PERL en Windows y en Unix?
En general, sí. En particular, hay algunas cosas que no funcionan de la misma forma: por ejemplo, la orden `fork` no funciona (salvo en la nueva versión 5.6), el acceso a base de datos no se hace igual (se usa el módulo DBD/DBI en Linux y en Windows, generalmente, el Win32::ODBC, aunque también se puede usar el mismo); para ejecutar un programa en PERL hay que definir una serie de cosas. Pero en general, cualquier programa que no use ninguna característica específica de un sistema operativo funciona exactamente igual en los dos sistemas operativos. Y si hay alguna duda, se puede insertar código particular detectando el valor de la variable `$OSNAME`, `$^O` o `$Config{osname}` . Por ejemplo, se puede hacer algo así:

```
if ( $^O eq 'linux' ) {
    [código específico para Linux];
} else {
    [código para otros SOs menos afortunados];
}
```

¿Existe algún sitio web, preferentemente gratuito, donde se puedan poner scripts en PERL?
Existen varios: Lycos, por ejemplo, deja poner CGIs y PHP; si tienes algún proyecto en SourceForge, puedes poner también CGIs, aunque la versión de Perl está anticuada. La mayoría de los hosting comerciales permiten poner Perl sin problemas.

Oye, qué tutorial más chachipiruli tienes. ¿Puedo copiarlo a mi sitio web, o incluirlo en un CD que voy a repartir, o que voy a vender? En resumen: temas de licencia y copyright
La respuesta corta es vale, si y no; la respuesta larga es que, en general, este tutorial se puede copiar sin problemas, siempre que lo dejes como está y pongas un enlace a su sitio original, este; lo mismo sirve para cachos del tutorial, o los ejemplos. En cuanto a los CDs, si los CDs son de algún modo comerciales, es decir, se venden a un precio que supone ganancia para el que los vende, no, no hay permiso; si los CDs son sin ánimo de lucro, gratuitos, parte de un curso, o incluidos junto con otras muchas cosas en una revista, sí, si hay permiso, siempre que se incluya tal cual, y se me mande una copia (inclusive revista, si fuese así) a

```
JJ Merelo
Depto. Arquitectura y Tecnología de Computadores
ETS Ingeniería Informática
C/ Daniel Saucedo Aranda, s/n
18071 Granada (España)
```

Todo esto, por supuesto, afecta también a cualquier parte del tutorial o a los ejemplos que se incluyen con el mismo.

¿XML? ¡Pero si es muy simple!

Usando XML desde PERL de la forma más simple posible

Como nuestro político corrupto está en la oposición, ya no tiene tanto tiempo de corromperse, y sí mucho más tiempo de aprender PERL, así que decide poner orden en sus cuentas, y ha oido hablar de una cosa que se llama XML, así que, ya puestos, lo hace en XML, para poder trabajar fácilmente con bases de datos, y para añadir un poco de metainformación a todo lo que tiene. Su fichero de clientes, en XML, podría tener esta pinta:

```
<clientela>
  <cliente>
    <nombre>Filemon Pi</nombre>
```

```
    <pasta> 404041.5</pasta>
    <fecha>13:50 30/1/2000</fecha> </cliente>
<cliente>
    <nombre>Pantuflo Zapatilla</nombre>
    <pasta>3750 </pasta>
    <fecha>13:50 23/2/2000</fecha></cliente>
<cliente>
    <nombre>Superintendente Vicente</nombre>
    <pasta>64041.5</pasta>
    <fecha> 13:50 25/2/2000</fecha></cliente>
<cliente>
    <nombre>Zape Zapatilla</nombre>
    <pasta>50000</pasta>
    <fecha> 13:50 22/3/2000</fecha></cliente>
<cliente>
    <nombre>Zipi Zapatilla</nombre>
    <pasta>5000</pasta>
    <fecha> 13:50 29/3/2000</fecha></cliente>
</clientela>
```

.

Por supuesto, hizo un programa en PERL para pasarlo del formato anterior a este. En realidad, visto así, el XML no es tan complicado: basta con poner:

- Una etiqueta raíz, que en este caso es `clientela`.
- Todas las etiquetas tienen que estar bien emparejadas, es decir, deben de "terminar" en orden contrario al que "empezaron".

Esto, y algún detalle más, es lo que se denomina XML bien formado. Ya es más fácil procesarlo que el texto anterior, sobre todo si la información que hay en cada "campo" no siempre tiene el mismo formato, y se puede hacer simplemente con expresiones regulares. Sin embargo, es más simple hacerlo usando el módulo que se llama apropiadamente `XML::Simple` de esta forma (`xml.pl`):

```
1  #!/usr/bin/perl
2  use XML::Simple;
3  my $clientes = XMLin("/my/home/txt/tutoperl2000/clientes.xml");
4  for( @{$clientes->{cliente}} ) {
5    print "$_->{nombre} => $_->{pasta} ($_->{fecha})\n";
6  }
```

Cuando se dice que una cosa es simple, es que es simple: 5 líneas, que podrían ser 3. La función `XMLin`, que se importa automáticamente del módulo `XML::Simple`, toma un fichero XML, se lo traga entero, y lo mete en una estructura de datos compuesta de hashes de hashes de hashes. En nuestro caso, en `$clientes->{cliente}` habrá un array con todos los clientes, es decir, todos los contenidos de todas las etiquetas clientes; pero esos contenidos estarán también analizados, y, por tanto, dentro de `$clientes->{cliente}[0]{nombre}` estará el contenido de la etiqueta `nombre` del primer cliente. En el programa, en la línea 4, se hace un bucle sobre el array principal, y se van imprimiendo los contenidos de las etiquetas del "segundo nivel".

Una referencia que trata exclusivamente de PERL y XML es *Perl & XML de Erik T. Ray, Jason McIntosh* . Cubre todos los módulos y entornos de **PERL** y XML, tales como el AxKit, y diferentes módulos que se pueden usar, aparte del que se explica aquí.

Si cada cliente tuviera varios nombres, el contenido de la etiqueta `nombre`, en vez de ser simplemente el nombre, sería un array. En resumen, lo que se hace es traducir el árbol en el que se convierte el fichero XML a un árbol de matrices asociativas: cada etiqueta se convierte en una clave de una matriz asociativa, y si hay varias etiquetas en un nivel el contenido de esa clave es un array con todos los interiores de las etiquetas

Algunas notas sobre la instalación

Instalar el módulo `XML::Simple` es relativamente fácil; basta con dar la orden correspondiente desde el Perl Package Manager o desde el módulo CPAN. Sin embargo, previamente tiene que estar instalado el programa y la librería de `expat`, que es un parser de XML. Expat se lo puede uno bajar desde `http://expat.sourceforge.net/`, tanto en fuentes como en RPMs (para RedHat).

Para saber más

Si quieres saber más sobre XML, hay un curso completo dentro de GeNeura:Formación, incluyendo una lección sobre parsers de XML en PERL, el único que hay en castellano en la red.

Bibliografía.

Libros para ahondar en el mundo del Perl

Perl viene con todo el material necesario para trabajar; en el caso de la versión 4, con una macro-página de manual en formato UNIX, y en el caso de la versión 5, con varias páginas de manual, e incluso páginas de manual en HTML. Hay, además varios libros publicados:

- Programming Perl, de Larry Wall y Randall L. Schwartz, O'Reilly and associates. Se le suele llamar "The Camel Book" por el camello de la portada. Escrito por el gran guru, incluye manual de referencia, muchos ejemplos, y una explicación de cada una de las estructuras.

- Professional Perl Programming, de Peter Wainwright, Shelley Powers, Aldo Calpini, Arthur Corliss, Simon Cozens, JJ Merelo-Guervós, Aalhad Saraf, Chris Nandor, Wrox Press. A este se le podría llamar el "libro de las cabezas". Un libro muy completo, que trata todos los aspectos de Perl en profundidad, incluyendo algunos que otros libros se dejan fuera: programación usando `locale`, por ejemplo, módulos, procesos y programación dirigida a objetos.

- Learning Perl, de Randall L. Schwartz y otros, O´Reilly and associates. El "Llama book"; se puede obtener de la misma forma, por ejemplo, de Amazon. Si quieres aprender, después de leer este tutorial, puedes seguir con este libro. Más educativo que el anterior.

- Perl cookbook, de Larry Wall y otros, un libro con un montón de ejemplos, que muestra todo lo que puede hacer Perl, desde CGIs hasta conectar con otras máquinas usando telnet. A este se le debe conocer por el libro del cabrito, o algo así.

- The Perl Cd Bookshelf : 6 Bestselling Books on Cd-Rom, todos los libros anteriores, y además en CD-ROM, para poder buscar lo que a uno le interesa. Imprescincible.

- Programming the Perl DBI por Alligator Descartes, Tim Bunce , que explica claramente cómo usar este módulo que sirve para acceder a bases de datos, y una introducción general sobre cómo trabajar con bases de datos y Perl.

- Teach yourself Perl in 21 Days, David Till, SAMS Publishing; un tocho considerable que me encontré en una librería al lado de la Facultad. Muy completo, pero no tan divertido como el "Camel Book". Por cierto, que me lo han chorizado, por favor, si alguien lo ve, que me lo diga.
- Developing Applications in Perl, de Tom Christiansen, Byte, April 1994, p. 231-236.

Además, se han publicado tutoriales en diversas publicaciones, como UNIX World (Mayo-Julio 1990), y en la fenecida revista Click!, de marzo a junio de 1995 (una versión anterior de este mismo tutorial).

Recursos Internet

Enlaces a sitios en internet que hablan sobre Perl

Sitios generalistas

- Alguna cosilla hay en `http://www.perl.com`, sitio central dedicado al Perl. Este sitio contiene noticias, listas de correo, enlaces a todos los sitios de donde puede uno bajarse las diferentes versiones de Perl.
- Los sitios CPAN (Comprehensive Perl Archive Network), o red de sitios con archivos, módulos, y librerías relacionados con el `Perl`.
- Los sitios PerlMongers (`http://www.pm.org`) están dedicados a grupos de usuarios de perl; y también hay un `www.perl.org`.
- La documentación de `Perl` se puede encontrar en PerlDoc.
- Hay varios sitios que incluyen noticias periódicas sobre `Perl`: Use Perl y Perl News, ambos incluyen noticias sobre nuevos módulos, nuevas versiones, eventos relacionados con `Perl` y cotilleos en general.
- La versión para WindowsNT/98/95/2K se puede bajar de la empresa ActiveState `http://www.activestate.com/`.
- Hay unos cuantos grupos de usenet dedicado al Perl; `comp.lang.perl`, tiene un tráfico de unos 100 mensajes al día, y normalmente los propios desarrolladores de Perl, Larry, Tom Christiansen (que, por cierto, habla español) y Randy Schwartz contestan a tus dudas. Mucho tráfico, pero nada que no pueda salvar un buen killfile; otros grupos se dedican a anuncios `comp.lang.perl.announce`, y a una version de Perl combinada con tcl/tk, el tkPerl, `comp.lang.perl.tk`.
- Indigo Star ha creado una versión de `Perl`, IndigoPerl, que incluye un servidor Apache; es ideal para depurar CGIs en sistemas WinXX (todo lo ideal que pueda ser en estos sistemas).

Sitios en castellano: tutoriales, noticias

Además, hay una serie de sitios sobre Perl que están en castellano

- En el Yahoo español hay bastantes punteros a recursos en castellano, varios cursos, sitios con código y demás.
- Otro sitio en castellano, Código Perl, con programas de ejemplo y tutoriales. Enfocado sobre todo a programar CGIs en Perl.
- El sitio de Perl en español, creado por el grupo de usuarios de Perl en Tijuana, México, con enlaces a varios tutoriales y FAQs traducidas al español.
- Hay diversas listas de correo que tratan de Perl en español, muchas de ellas alojadas en YahooGroups . La principal es perl-es. Tiene mensajes de todo tipo, desde el básico hasta el avanzado, y viene a tener 1-2 mensajes al día.
- Un tutorial bastante extenso, y en postscript, es el escrito por Diego Sevilla, que es además uno de los habituales en la lista de correo anterior. Es muy completo, y llega incluso a explicar cómo acceder a bases de datos por ODBC en Windows (el enlace al tutorial de Perl está al final de la página.
- El que tiene la portada más cachonda es la guía del programador, con lo de Perl lava más blanco y demás.

Sirve para principiantes, pero también explica como programar server side includes y CGIs.

- Uno de los clásicos, la introducción al lenguaje Perl, de Víctor Osuna y Eleuterio Rodríguez, en la Universidad de Córdoba (España), en la que todavía parpadean los fallos HTML y la fecha de última modificación, hace 4 años.Incluye ejercicios, una pequeña guía de referencia, y llega hasta las expresiones regulares. Bastante básico, pero está bien si quieres aprender expresiones regulares. Este tutorial desapareció de su sitio original y apareció en Colombia, lo cual demuestra que los buenos tutoriales de Perl nunca mueren.
- Desde Colombia, nos llega El Evangelio de Perl (en .doc), un tutorial bastante avanzado, que incluye cómo usar referencias y otras características de la versión 5. El HTML es un poco espartano, eso sí. También he tenido que buscarlo por el mundo; ha aparecido en Argentina.
- Un excelente tutorial en castellano sobre Perl DBI, de Javier García Castellano, enfocado principalmente al comercio electrónico, que explica cómo programar carritos de la compra usando Perl, DBI y MySQL.
- Otro excelente tutorial en castellano sobre expresiones regulares, CGIs y subrutinas en Perl, de Pedro Castillo Valdivieso.
- Aparte de todo, este tutorial es uno de los más copiados en Internet, aparte de las réplicas oficiales en LuCAS; no hay más que buscar en google. Por favor, si queréis poner un tutorial de Perl en vuestras páginas, poned un enlace a este tutorial; o por lo menos reconocer quién ha sido el que lo ha escrito (el menda).
- Hay algunos sitios web sobre perl "regionales", como Perl en Español (la carpeta spanish contiene un montón de tutoriales), el grupo Perl Mongers de Madrid (con enlaces a cursos), de Barcelona, La Rioja (con enlaces a diferentes tutoriales)
- El sitio original de este tutorial de Perl es este o este, en caso de que lo estés mirando desde otro sitio, ahí te podrás encontrar la última versión. También está replicado en TLDP-es, en Aula Digital, en Colombia; aquí, alguien se ha molestado en pasarlo a PDF, otra copia, en formato Wordy otra copia.
- Barrapunto de perl en castellano, colección de noticias relacionadas con el Perl, el español.