PERL

Juan Julián Merelo Guervós
jmerelo@kal-el.ugr.es

¿Qué es?

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 5.001) se ha convertido en un lenguaje de propósito general, y una de las principales herramientas de un buen internetero.

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 (`lwall@netlabs.com`) es un habitual del grupo de USENET `comp.lang.perl`, y es normal que el propio Larry conteste a una pregunta de un principiante o a un error con un parche para el mismo error.

Es un lenguaje que hereda estructuras principalmente de los intérpretes de comandos de UNIX, especialmente el `csh`, y de otras utilidades estándar, como `awk` y `sed`. En realidad, puede hacer todo lo que hacen cualquiera de ellos y todos ellos juntos, la mayoría de las veces de forma más simple, comprensible y fácil de depurar.

PERL es un lenguaje interpretado, aunque internamente funciona como un compilador. 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.

La versión actual del intérprete de PERL es la 5.001 (probablemente en el momento que salga esto habrá salido ya la 5.002). Es muy habitual, sin embargo, encontrar la versión 4.036, el último *patchlevel* de la versión 4 y probablemente el más estable. Actualmente, la distribución 5 está en plena evolución (y ebullición), y viene acompañada de unos cuantos megas de parches. El consejo del autor es esperar un poco, por lo menos un par de *patchlevels*, hasta que se estabilice. 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. Los scripts presentados en este tutorial están desarrollados para la versión 4, pero deberían de funcionar, en principio, para la versión 5.

Aunque desarrollado originalmente en un entorno UNIX, actualmente hay versiones para casi todos los sistemas operativos: Windows NT, Amiga, MacOS (ver Bibliografía/Recursos). Los scripts son también compatibles, 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.

¿Para qué sirve?

Principalmente, 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, o scripts ejecutados desde páginas de la World Wide Web.

Mi primer programa en 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` tiene el mismo significado que en C; es decir, un retorno de carro[1].

Ejecutarlo es otro cantar; o mejor dicho, muchos cantares, una coral entera. El PERL siempre Hay Muchas Formas de Hacer Las Cosas (marca registrada). Al menos tres, en este caso.

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

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

2. 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

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

Se ha utilizado el símbolo de *escape* \ delante de las comillas, para que las interprete como pertenecientes a la cadena y no como final de cadena.

3. 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

---

[1] Aunque el comportamiento es ligeramente diferente en UNIX y MSDOS; mientras que este separa las líneas con un retorno de carro y *linefeed,* UNIX y el Mac usa solo un *linefeed*. En cualquier caso, `\n` se comporta de la misma forma en los dos.

```
#!/usr/local/bin/perl
```

(los símbolos iniciales se suelen denominar *pound-bang*). 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`, 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). 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
```

Más difícil todavía

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:

```
print "Valor del inmueble\n";
$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
```

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; como se ve, 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).

El programa pide además, la intervención del usuario, leyendo en una variable lo que éste teclea. Para ello se utiliza un fichero que está disponible en todos los programas, la entrada estándar o STDIN (al igual que también existe la salida estándar o STDOUT, es decir, que es lo mismo `print STDOUT` que `print`). El nombre de este fichero no está precedido por ningún símbolo para indicar que se trata de un *filehandle*, o una variable que representa un fichero (el equivalente a un `FILE*` en C). Los *angle brackets*, o paréntesis angulares, indica que se lee del fichero una línea completa, hasta que el que teclea pulsa ↵.

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

Fig. 1 Windows es un entorno ideal para trabajar con PERL, usando por ejemplo una ventana con el MS-DOS y algun editor *shareware*, como el *Programmer´s File Editor*.

en el caso de tratarse de comillas simples).

Dando vueltas sobre lo mismo

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 recuadro 1.

En este segundo programa, que llamaremos `clientes.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 el cuadro 2.

1 Programa clientes.pl

```
 1   open( MANTA, ">clientes" );
 2   while(1) {
 3       print "Cliente\n";
 4       chop( $paganini = <STDIN>);
 5       last if !$paganini;
 6       print "Valor inmueble\n";
 7       chop($valor = <STDIN>);
 8       $comision = $valor * 0.25;
 9       print "Comision = $comision\n";
10       print MANTA "$paganini $comision\n";
11   };
12   close (MANTA);
```

2 Modos de apertura de ficheros.

>	Abrir para escritura
>>	Abrir para concatenar
<	Abrir para lectura
| *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, 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). Y 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 el 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.

3 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 de 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 expresión 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.

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[2].

Ejercicios 1

1. Realizar un programa que vaya calculando cuadrados de números hasta que se introduzca una línea en blanco.

2. Realizar un programa que permita al usuario introducir un nombre, y el ordenador le escriba "Hola <mismo nombre en mayúsculas>".

Ahorrando energías

Como nuestro político corrupto 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 486DX4 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` (cuadro 3).

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]`. 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

[2] Tener en cuenta que UNIX distingue entre nombres en mayúsculas y minúculas, a diferencia de MS-DOS.

```
 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      $zipi = "$paganini $comision $hora:$min $dia/$mes\n";
11      push( @mem, $zipi );
12  };
13
14  open (MANTA,">clientes.mas");
15  foreach (sort @mem ) {
16      print MANTA;
17  }
18  close (MANTA);
```

`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.

En la línea 11 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). Si queremos eliminar el primer componente de una matriz, se hace con el operador obvio,

```
$cadena = pop(@mem);
```

En la línea 15 viene un nuevo operador 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 éste 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. En este caso la ordenará por orden alfabético, saliendo algo como esto

```
Pepe Illo 4750000 13:19 24/3
Per Illan 4749997.5 13:19 24/3
```

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 PERLística; 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.

Recordando, que es gerundio

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

5 Programa `clientes.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 siguiente (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

```
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, primer ejemplo de lectura de ficheros, empieza con una advertencia:

6 Fichero `clientes.mas`

```
zape 150000 10:40 24/3
zipi 324999.75 10:40 24/3
Mortadelo 123455 8:55 24/3
Filemon 555666 13:44 25/4
Vicente 545321 12:01 25/4
Profesor Bacterio 443322 07:00 26/4
Ofelia  23441  44453 15:00 26/4
```

"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 un argumento al programa; la matriz `@ARGV` contiene los argumentos pasados al programa; de forma que `$#ARGV` dará el número total de argumentos. Y otra forma de hacerlo sería

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

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* deltro de esos ángulos, 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.

Sobre la variable por defecto actúa la orden `split` (una de las más usadas en PERL), dividiéndola en una serie de cadenas separas 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.

[3] Aunque hay todavía una forma Más Fácil de Hacerlo

Expresiones regulares

Una forma todavía más directa de trabajar con estos ficheros 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, y una letra o un número luego 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 caracter, en la línea 4 del programa anterior).

7 Algunas convenciones usadas en expresiones regulares.

.	Describe cualquier caracter, 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 debe de aparecer; {N} significa exactamente N veces; {N,}, como mínimo N veces.
[..]	Indica una clase de caracteres, [^...] niega la clase.
(..|..|..)	coincide con una de las alternativas.
\w	coincide con los alfanuméricos, \W con los no-alfanuméricos.
\s	con los espacios en blanco, \S con los que no lo son.
\d	con los numéricos, \D no-numéricos.

El fichero 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+)
```

que puede parecer un poco críptica (y en realidad lo es), pero cuyo significado se puede resolver mirando el cuadro 6. Con esta modificación, el bucle central del programa se queda reducido a

```
while(<>) {
    ($paganini, $pasta, $hora, $dia, $mes)
        = /(\D+) (\d+\.?\d+) (\S+) (\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 una por partes.

La primera expresión regular es `\D+`. Como se ve en el cuadro, \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;` 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 esta expresión regular como `(\S+) (\S+) (\S+) (\S+) (\S+)` funcionaría perfectamente.

Y para terminar, `(\d+)\/(\d+)` coincide con el día y el mes, 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.

Presentando lo impresentable

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 recuadro.

En este programa ya estamos hablando de palabras mayores. Para empezar, es un poco más largo. Para continuar, es sólo para MS-DOS, 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.

8 Listado del programa `dir.pl`. Está disponible en `ftp://kal-el.ugr.es/click`.

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

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) =
@_;
```

Fig. 2 Salida del programa `dir.pl`.

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.

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`.

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, y que vienen heredados de comandos similares en los

shell de UNIX, nosotros tenemos la suerte de poder usarlos en MS-DOS también. Otros operadores figuran en la Tabla I.

Decodificando

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 dejando 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).

Tabla II Algunos 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 en MS-DOS.

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`.

9 Programa `uudec.pl` para decodificar mensajes.

```
1   #!/usr/local/bin/perl
2
3   while (<>) {
4       #Buscar el nombre de fichero
5       if ( /begin \d\d\d\s+(\S+)/ ) {
6           $fileName = $1;
7           open ( OUT, ">$fileName" );
8           print "Fichero $fileName\n";
9       }
10      print OUT unpack("u", $_) if (/^M/ && !/Message/ );
11      print OUT unpack("u", $_) if ( /^F/ && !/From/ );
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. os mensajes en ciertos foros de USENET vienen divididos en varias partes, cada una de las cuales contiene un

pedazo de un fichero codificado con uudecode (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. Todas las líneas codificadas con `uudecode` comienzan con una M, es decir, que cumplen la expresión regular `/^M/` (^ corresponde a principio de línea); en la línea 10 excluimos también la línea de la cabecera que comienza por `Message`; ídem hacemos en la siguiente línea; a veces la última línea de un fichero comienza por F.

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).

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).

Bibliografía.

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. Hay varios libros publicados:

‣*Programming perl*, de Larry Wall y Randall L. Schwartz, O'Reilly and associates. Se le suelle llamar "The Camel Book" por el camello de la portada; se puede pedir directamente por e-mail a `info@ora.com`, o mediante la WWW a `http://www.ora.com`.

‣*Learning perl*, de Randall L. Schwartz, O´Reilly and associates. El "Llama book"; se puede obtener de lal misma forma.

‣*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".

‣*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).

Recursos Internet

‣Lo principal: La metaFAQ de PERL, disponible en `ftp://kal-`

el.ugr.es/click/perl-meta-faq; y en http://www.khoros.unm.edu/staff/neilb/perl/metaFAQ/metaFAQ.html. Aquí se puede encontrar toda la información posterior, y la bibliografía.

‣Hay un grupo 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*.

‣Donde conseguirlo: el sitio principal es ftp://ftp.cis.ufl.edu/pub/perl; aquí están todas las versiones, así como versiones compiladas para diversas máquinas.

‣Manual de referencia: escrito por Johan Vromans, está entre otros sitios en ftp://ftp.cs.ruu.nl/pub/DOC. Viene incluido con el libro del camello; hay además tanto para la versión 4 como la 5.

‣PERL para DOS: ftp://src.doc.ic.ac.uk/packages/ibmpc/simtel/perl; en el mismo sitio hay versiones para el Mac y el Amiga.

‣En la WWW: al menos cinco sitios, entre ellos el mencionado anteriormente; mi preferido es http://www.cis.ufl.edu/perl.

‣Camiseta PERL : http://www.emoticon.com/emoticon/perl.html. Una compañía inglesa en la que se puede comprar una camiseta que tiene por delante el célebre camello, y por detrás tu dirección de e-mail entre otras más.