Universidad de Málaga

Fundación Observatorio Universidad-Empresa de Málaga

Curso de programación en Perl

Documentación completa

Versión de la documentación para Manuel Molino Milla

10 de Noviembre de 2013

Profesor:
David Santo Orcero
dsanto@uma.es
irbis@orcero.org

Índice general

Capítulo 1

Introducción al Perl

Perl ha sido denominado "lenguaje de sólo escritura" porque, según muchos, un programa en Perl una vez escrito es imposible de entender. Goza de mala fama dentro de los más adictos a la informática teórica; a pesar de soportar desde hace años programación estructurada, programación orientada a objetos, e incluso soporta programación funcional, programación de orden superior, cierres, o persistencia de objetos. Es extraño que un lenguaje que tenga tantas características que deberían hacer las delicias de los más "teóricos" tenga tan mala fama entre ellos; y, por otro lado, sea masivamente empleado por administradores de sistemas, y programadores de CGIs; en entornos donde lo importante es que el trabajo se ejecute, no que se aplique la metodología o teoría de moda. Hasta el autor de PHP reconoce que programa en Perl. ¿Qué hace que un lenguaje tenga estas características tan especiales?

1.1. Filosofía detrás del lenguaje

Perl no nace de las manos de un matemático o un físico. Nace de la mano de un lingüista, que necesitaba sacar adelante imperiosamente su trabajo de administrador de sistemas en Unisys. Larry Wall, de formación lingüista, estaba saturado administrando más máquinas de las que un humano normalmente puede tolerar. Es decir, el estado normal de un administrador de sistemas. Necesitaba una herramienta que le ayudara a llegar a su casa todos los días a una hora razonable sin volverse loco. Y se la hizo.

En lingüística, una de las cosas que se aprende es que la gente termina hablando como quiere –según algunas teorías, termina hablando de la forma más óptima–; que toma préstamos de otras lenguas, que si encuentra una expresión

que le gusta, se la apropia. Y que siempre hay muchas formas de decir lo mismo. Y eso es lo que termina siendo Perl.

Lo normal cuando alguien diseña un lenguaje es que tome su paradigma de programación favorito, y construya un lenguaje sobre dicho paradigma. Así nacen Haskell, Prolog o Pascal, por poner ejemplos. O que tome su lenguaje favorito -o el de moda-, y le añada a martillazos el paradigma de moda. Así nacen C++ o Java. O el peor de los escenarios: que lo diseñe un comité. Solo en esos casos se consiguen lenguajes como COBOL o ADA. Perl nace de la idea de "hay que sacar el trabajo adelante **como sea**", y se han añadido cosas según permitan sacar el trabajo adelante lo más rápido que sea posible, de la forma más simple que sea posible.

Hay otro elemento clave para entender Perl. Perl es un lenguaje que ha originado a su alrededor toda una filosofía de trabajo: la colaboración. Está muy relacionado con el software libre: su primer intérprete es libre y se ha usado en gran cantidad de proyectos libres. La naturaleza colaborativa de Perl se ve en principalmente el CPAN: una inmensa biblioteca de rutinas libres para Perl realizada por los programadores de Perl desde su nacimiento. CPAN tiene módulos para prácticamente cualquier cosa que se nos pase por la cabeza; de forma que las más complejas funciones que se nos ocurran probablemente ya tenga un módulo en CPAN que las permita realizar con una línea. La mayor parte de el esfuerzo de programación será buscar en CPAN el módulo correcto, leer la documentación, copiar y pegar el ejemplo, y modificar algo de la llamada para que funcione en nuestro caso.

CPAN es el más importante almacén de módulos de la red de cualquier lenguaje de programación

Perl nació para administrar centenares de máquinas Unix a la vez, sin esfuerzo. Esto hace Perl es un lenguaje completamente extraño en su estructura visual para cualquier persona fuera del mundo Unix. Perl ha tomado muchas cosas de utilidades y herramientas que existen y se usan en Unix; por lo que si conoces sed, awk, programación de shell-scripts y C, la sintaxis de Perl es absurdamente sencilla. Si no conoces estas herramientas, aparentemente Perl tiene una sintaxis "difícil de ver" hasta que te acostumbras.

Si no entiendes las "palabrotas" de este párrafo, no te preocupes: son cosas que no son necesarias para programar en Perl. Ni siquiera para hacer buenos programas, de cosas complejas, en Perl. Algo muy importante que debes saber de Perl es el hecho de que Perl pueda hacer algo no significa que tengas que saber hacer eso para poder usar Perl. En Perl **siempre hay más de una forma de hacer las cosas**.

La naturaleza colaborativa del desarrollo de Perl es responsable de una de sus características mas peculiares: su eclecticismo. Mucha gente ha metido mano, para añadirle lo que les facilitaba la vida. Esto hace que sea un lenguaje inmenso, extremadamente flexible y extensible, que soporta casi todos los paradigmas de programación existentes. Y "casi" es por no pillarme los dedos: en Perl se pueden hacer cosas como que el programa Perl cree un programa Perl, rellene el código fuente en una variable, y ejecute el código fuente contenido en la variable; lo que permite componer en caliente programas que se modifiquen a si mismos[1]. Pero no nos debemos preocupar por esto: Perl no nos obliga a saber de todo esto para programarlo. De hecho, se pueden hacer incluso programas de una línea, sin declaración de variables ni funciones. Y la mayor parte de los programas serán de una linea, sin declaraciones de variables ni funciones. En Perl podemos ir aprendiendo a usar las cosas según las necesitemos.

Su diseño ha tenido como objetivo que Perl sea lo más fácil de aprender y usar por administradores de sistemas de Unix que fuera posible, y que resuelva problemas reales de gente real. Nunca ha sido su objetivo ni ser formalmente correcto, ni satisfacer a ningún teórico de la programación. De hecho, ni siquiera podemos decir que sea visualmente elegante. Pero resuelve rápido los problemas reales, y esto es lo que importa. Los programas en Perl resuelven los problemas cotidianos de administración en unas pocas lineas.

[1] Técnicamente hablando, Perl nos permite modificar en caliente la sintaxis del propio lenguaje. Esto lo usan módulos como Moose, que permiten programar en algo que parece un híbrido de Perl, CLOS y SmallTalk. Un caso extremo de esta funcionalidad es el módulo `Lingua::Romana::Perligata`, que nos permite programar Perl el latín. Y por "programar Perl el latín" no me refiero a que los mensajes de error son en latín, sino que el programa en Perl es una parrafada en latín, identificando casos, números romanos, y todas las peculiaridades del latín, e interpretándolo como un programa Perl.

Perl viene de "Practical Extraction and Reporting Language", y realmente como herramienta de extracción de información a partir de logs del sistema es realmente excelente. Las expresiones regulares de Perl, unidas a los potentísimos mecanismos de Perl para operar con listas y con cadenas de caracteres, permiten extraer informes de logs prácticamente sin problemas. En la práctica, administrar sistemas cuando sabes Perl no tiene absolutamente nada que ver en tiempo y esfuerzo a cuando no sabes Perl.

La cebolla es el logo de Perl

Hemos hecho mucho incapié en la administración de sistemas; pero esto no es lo único que podemos hacer con Perl: sumando a todo esto el increíble archivo de rutinas libres disponible para Perl, el CPAN, Perl puede ser usado con éxito para el desarrollo de CGIs; o incluso puede ser usado en grandes proyectos o para hacer prototipos de grandes aplicaciones. Actualmente se utiliza en áreas tan diversas como bioinformática, programación web, o administración de sistemas; y permite hacer cosas tan poco comunes en un lenguaje de propósito general como álgebra y cálculo sobre expresiones simbólicas.

Perl es un lenguaje que normalmente se usa interpretado; pero se puede compilar a un código intermedio propio, o a C. Esto facilita el mecanismo de ensayo y error del proceso de depuración tradicional.

Finalmente, aunque el logo de Perl realmente es una cebolla –por lo de las capas, y que dentro de las capas hay más capas–, de forma no oficial se asocia a Perl con la imagen del dromedario. La razón de esta asociación es la serie de libros de O'Reilly que tiene en sus portadas animales dibujados a mano. El libro de O'Reilly de Perl tiene en su portada un dromedario; y la popularidad de este libro ha hecho que de forma informal la gente asocie a Perl con dicho dromedario.

1.2. Tu primer programa en Perl

Vamos a ver la pinta que tendría un programa en Perl. Se suele comenzar con un programa que imprima "Hola mundo" en la pantalla. En Perl, sería:

```
print 'Hola Mundo!';
```

Si somos tan perros que no queremos ni molestarnos en abrir un editor de textos para llamar después al interprete Perl, podemos incluso hacer el programa desde la línea de comandos que llama al intérprete:

```
perl -e "print 'Hola mundo'";
```

Podemos quedarnos con la sensación de que nos estamos haciendo trampas al solitario. Y sí, Manuel, nos estamos haciendo trampas jugando al solitario. Nadie aprende a programar un lenguaje para escribir la cadena "Hola mundo" en la pantalla. Por ello, vamos a hacer algo un poco más complicado –pero no mucho–. Vamos a hacer un programa que se lance desde `cron` cada hora; y que vuelque la hora de la máquina que lanza el programa, después se conecte con una base de datos remota, y grabe en ella dos informaciones: en una tabla, el volcado de los procesos que se ejecutan en una máquina remota, y en la otra un listado de los enlaces a imágenes de una página web remota; pero solo a los enlaces que contengan la palabra "servidorA". Pero como esto es aún sencillo, vamos a poner algo de diversión: que la página web esté detrás de un formulario, y que haya que parsearla para obtener los enlaces por separado, y así insertar una tupla por enlace. Y ya que estamos, que al final mande un correo a través de un servidor SMTP remoto a `josele@servidor.com` con copia oculta a `martin@otroservidor.com` que contenga tanto el volcado de procesos del primer servidor como el volcado de procesos del servidor que ejecuta el script. Te dan un fichero, `patroncorreo.txt.tt`, que contiene un esquema de mensaje, en el que debemos sustituir `[% pslocal %]` por el volcado local de procesos, y `[% psremoto %]` por el volcado remoto de procesos.

Sigue siendo un problema sencillo, pero es un poco más divertido. Veremos como se hace en Perl:

```
#!/usr/bin/perl -w
use WWW::Mechanize;
use Data::Dumper;
use DBI;
use MIME::Lite::TT;
use Net::SMTP::SSL;

# Nos conectamos a la base de datos
$dbh = DBI->connect(
    "DBI:mysql:nombreBD;servidorbd.dominio.com:3306",
        'usuarioBD','claveDB');

#  Imprimimos información de fecha y hora,
# (que recogerá cron)
print "*********************************************\n";
print "Arrancando el ".`date`."\n";

#  Nos conectamos al servidor remoto, para extraer los
# procesos. Suponemos que autentificamos por clave
# pública, y que la tenemos bien instalada.
$salidaps=`ssh servidordeprocesos.dominio.com "ps afx"`;

# Insertamos lo extraído en la base de datos
 $sth = $dbh->prepare(
    "INSERT INTO monitorprocesos(momento,procesos) VALUES
       (NOW(),'$salidaps');");
 $sth->execute();

# Nos conectamos al servidor
  $bot = WWW::Mechanize->new(agent =>
        'Mozilla/5.0 (Windows NT 5.1; rv:2.0.1)');
```

```
  $bot ->get("http://www.servidorweb.com/paginadeentrada")
     ;

# Rellenamos el formulario de acceso
  $bot->submit_form(form_number => 0,
    fields=> {  username => 'usuarioWeb',
                password => 'claveWeb' }
    );

# Nos traemos la página que queríamos
  $bot->get("http://www.servidorweb.com/paginaquequiero");

# Extraemos solo las imágenes
  @enlaces = $bot->find_all_links(
        tag => "a", url_regex => qr/\.(jpe?g|gif|png)$/i);

# Para cada enlace a imagen...
 foreach $enlace (@enlaces)
        {

        # ...si contiene la cadena "servidorA"...
        if( @{$enlace}[0] =~ /servidorA/)
        {
        # ...insertamos el enlace en la base de datos
        $sth = $dbh->prepare(
          "INSERT INTO monitorenlaces(momento,enlace)
             VALUES (($enlace),'@{$enlace}[0]');");
        $sth->execute();
        };
        };

# Y mandamos el correo
$opcionescorreo{INCLUDE_PATH} = '/var/
   donde_esté_el_template';
$params{pslocal}=`ps afx`;
```

```
$params{psremoto}=$salidaps;
$msg = MIME::Lite::TT->new(
 From            =>  'noreply@servidorlocal.com',
 To              =>  'josele@servidor.com',
 Bcc             =>  'martin@otroservidor.com',
 Subject         =>  "Ocurrió en ".`date`,
 Template        =>  'patroncorreo.txt.tt',
 TmplOptions     =>  \%opcionescorreo,
 TmplParams      =>  \%parametroscorreo
 );
 MIME::Lite->send('smtp','servidorcorreo.dominio.com',
        AuthUser=>'usuarioCorreo',AuthPass=>'Clavecorreo')
           ;
```

Esto se parece más a un programa Perl; y te permite hacerte una idea de la pinta que tiene un programa Perl real, en lugar de "holamundos" irreales.

No te preocupes con todo lo que no entiendas aún, que será mucho: todo lo que he puesto en este programa lo aprenderás durante el curso. En los primeros temas aprenderás a usar las variables, las sentencias de control, y los objetos, métodos y atributos que ves en el ejemplo.

A mitad del curso aprenderás a utilizar las expresiones regulares.

Y al final aprenderás las aplicaciones concretas: a usar una base de datos, a programar un spider de red, a conectarte e interactuar con los comandos de Unix desde Perl, o a mandar y recibir correo desde Perl.

1.3. Actividades

Este tema es introductorio; y no tiene ninguna actividad puntuable. Se trata apenas de transmitirte la filosofía detrás del lenguaje. Puedes mirar el código, e intentar hacer en el lenguaje de tu preferencia el ejercicio del final de este capítulo que te he resuelto yo en Perl.

Si tienes tiempo, busca a través de google lugares como CPAN, PerlMonks, PerlMongers... hay una ingente cantidad de información, documentación y bibliotecas libremente disponibles en la red. Vamos a volver a estos lugares de referencia en poco tiempo, pero es bueno que sepas ya que existen.

Si entiendes el inglés, los "State of the Onion" son interesantes y divertidos. Son charlas –transcritas– impartidas por Larry Wall, el creador de Perl, que además de darte idea de hacia adonde va Perl, te permitirá adquirir la idea intuitiva de la fuerte sensación de comunidad que hay detrás de los programadores en Perl, y del enfoque en hacer las cosas...

...o de **los enfoques**. Porque recuerda; **siempre hay más de una forma de hacerlo.**

1.4. Qué viene después

El próximo paso será la instalación de Perl.

Capítulo 2

Instalando Perl

Si utilizas cualquier sabor de Unix –sea Linux, o sea MacOS X–, estás de suerte: si no tienes Perl ya instalado, la instalación es muy fácil. Pero si utilizas Windows, no desesperes: te explicaré como instalar Perl en tu máquina, bajo Windows.

2.1. Instalando Perl en Linux

Si quieres instalar Perl y utilizas Linux, probablemente no tengas que instalarlo. Porque ya lo tendrás instalado sin haberte dado cuenta.

Casi todas las distribuciones de Linux instalan Perl sin preguntar, en su instalación por defecto[1]. Y las poquísimas que no lo hacen, lo incluyen dentro de sus sistemas de paquetería; de forma que instalarlo es tan fácil como recurrir al sistema de paquetería de tu distribución.

Puedes ver si ya tienes Perl instalado, haciendo:

```
man perl
```

Salimos de esta pantalla pulsando la tecla `q`.

Si lo tienes instalado, ya has terminado con este tema. Si no, hay un procedimiento distinto para cada distribución. En el caso de Mandriva o Mageia, será:

```
sudo urpmi -a perl
```

[1] No es por gusto: la mayor parte las utilidades de instalación y configuración automática de la mayor parte de las distribuciones están escritas en Perl.

Échale paciencia. Hay mucho que instalar. Debería instalar `perl-base`, y un montón de bibliotecas para Perl –veremos más adelante que estas bibliotecas en Perl se denominan "paquetes"–.

Las distribuciones derivadas de Debian –como puede ser, por ejemplo, Ubuntu o Mint– emplean para instalar Perl algo como:

```
sudo apt-get install perl
```

Las distribuciones derivadas de Red Hat –como puede ser, por ejemplo, Fedora o CentOS– emplearán a su vez algo como:

```
sudo yum install perl
```

Si eres usuario de Linux, probablemente conoces el mecanismo de tu distribución; sea con la herramienta de texto de gestión de paquetería, o con la herramienta gráfica que suele acompañar a las distribuciones modernas.

2.2. Instalando Perl en Mac OS X

Perl viene preinstalado en Mac OS X, por lo que no debemos preocuparnos por instalarlo. Por otro lado, `fink` está programado en Perl, por lo que si trabajamos con `fink` tenemos seguridad de disponer de un intérprete nativo Perl completamente funcional. Además, si usas `macports` con regularidad, lo más normal es que ya tengas Perl instalado; ya que muchísimos paquetes de `macports` dependen de Perl, y es fácil que en el hipotético pero improbable caso de que no tuvieras Perl instalado de entrada, lo hayas instalado ya sin darte cuenta al tirar de dependencias para instalar `fink` o cualquier paquete de `macports`.

Puedes verificar que tienes instalado Perl haciendo:

```
man perl
```

en un terminal de texto.

Si no lo tienes instalado, lo que es raro porque Perl debe venir con Mac OS X –yo me lo suelo encontrar instalado en Mac OS X; por lo que o viene ya de serie, o te lo instala de forma automática con un mínimo uso que hagas del Mac OS X como sistema operativo para programación–, puedes instalarlo a mano. Si quieres instalarlo, o utilizar un intérprete de una versión distinta, los pasos para instalarlo con `macports` serán:

```
sudo port install perl5
```

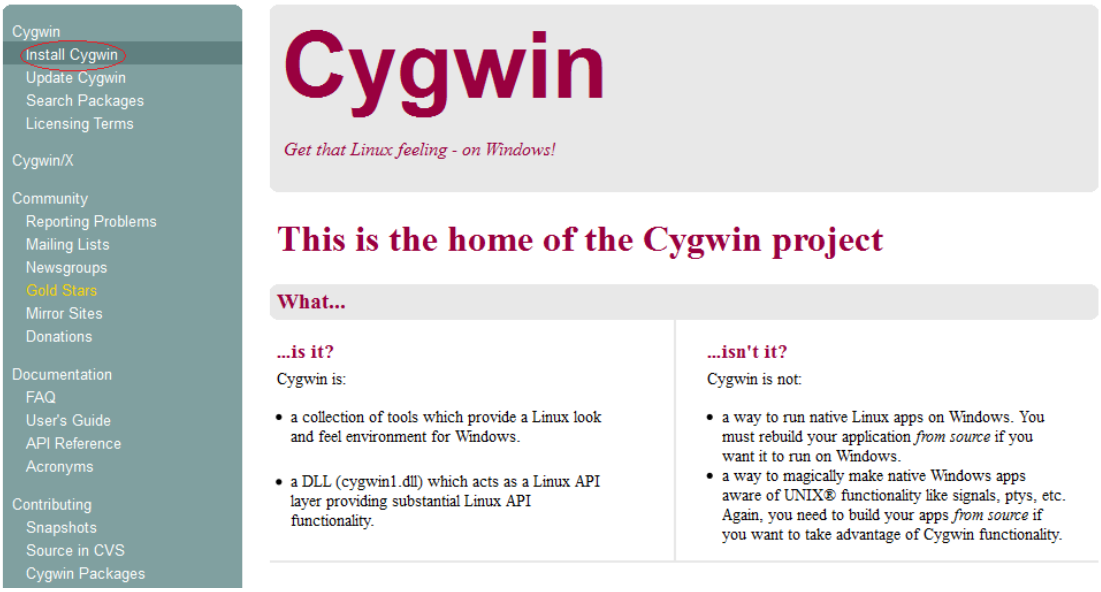
En cualquiera de los casos, te recomiendo que instales `Xcode`[2] antes de ponerte a trabajar con Perl.

2.3. Instalando Perl en Windows

Bajo cualquier sabor de Unix, o Perl ya viene instalado, o es de instalación trivial. Pero Perl bajo Windows es harina de otro costal. No es tampoco excesivamente complicado si sabemos lo que hacemos, pero no es trivial como en Unix.

La forma más práctica de usar Perl bajo Windows es instalando Cygwin. Cygwin es un paquete de herramientas que incluye las utilidades más prácticas de Unix, y permite emplearlas bajo Windows. Incorpora una biblioteca POSIX, el compilador GCC, las utilidades de desarrollo más comunes... y Perl[3].

Para descargar Cygwin entramos en su página oficial:
`http://www.cygwin.com/`
y seleccionamos la opción en "Install Cygwin":

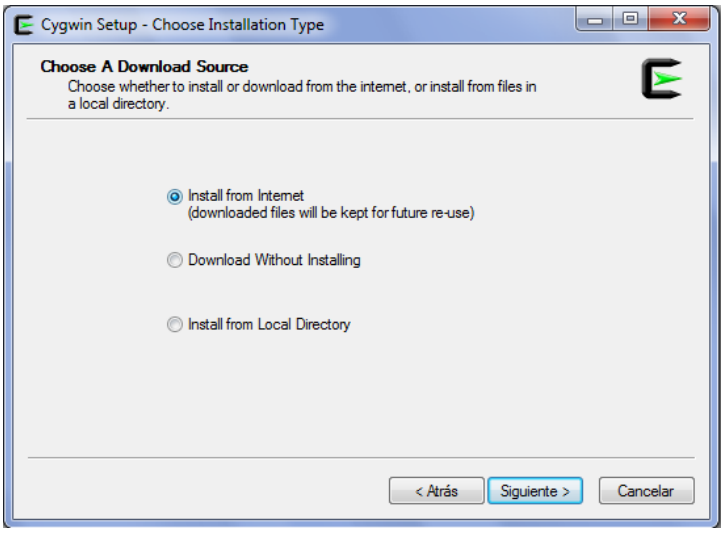


[2] Lo más común es que ya tengas instalado `Xcode` en tu MacOS X si lo empleas habitualmente para programar.

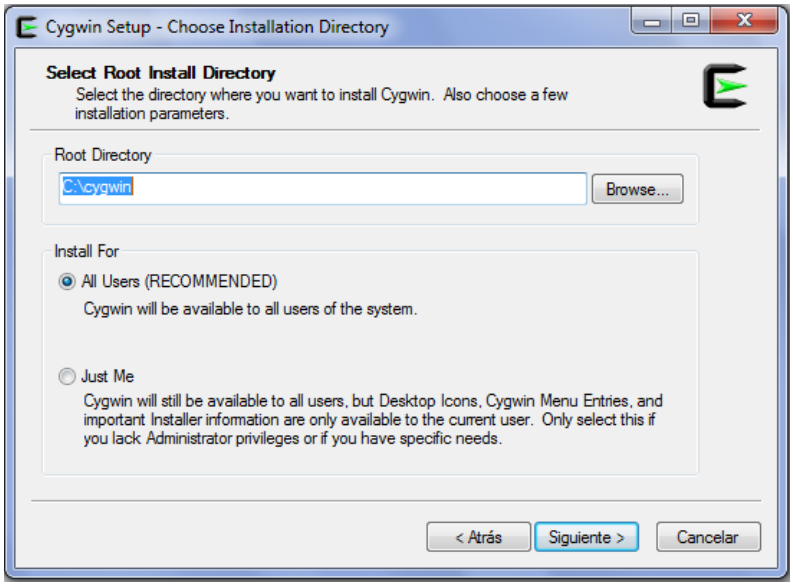
[3] Si estás haciendo el curso de LaTeX 2_ε también o tienes planificado hacerlo, Cygwin incluye LaTeX 2_ε; por lo que instalando Cygwin puedes tener ambos paquetes.

Después de descargarlo iniciamos la instalación con "Setup":

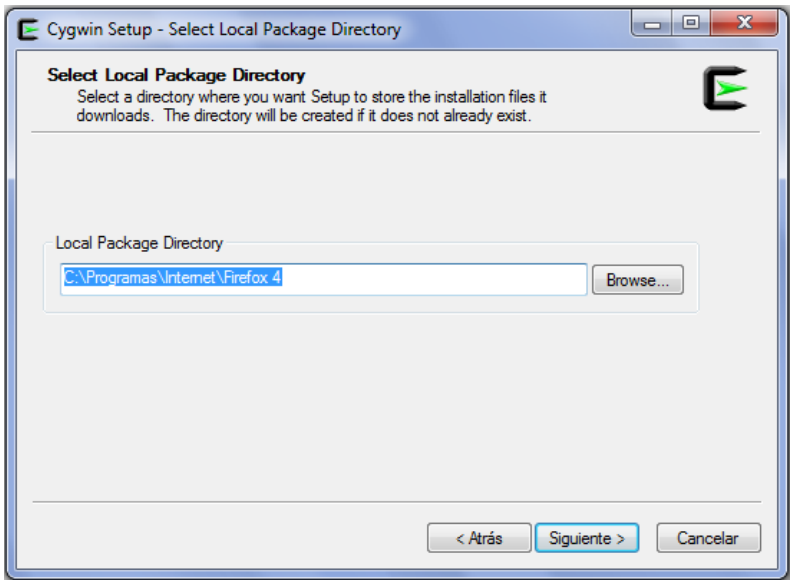
Y le decimos que lo instale desde Internet; con ello, obtendremos las últimas versiones de los paquetes, y además solo descargaremos lo que necesitamos. Esto lo hacemos en la pantalla:



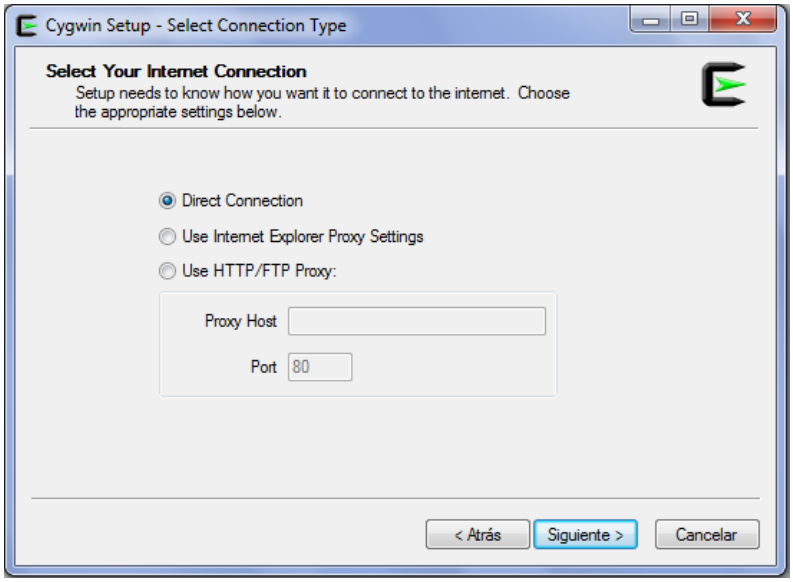
Le indicamos la ruta donde queremos que instale Cygwin, así como que lo instale para todos los usuarios:



También indicamos un directorio local para los paquetes:

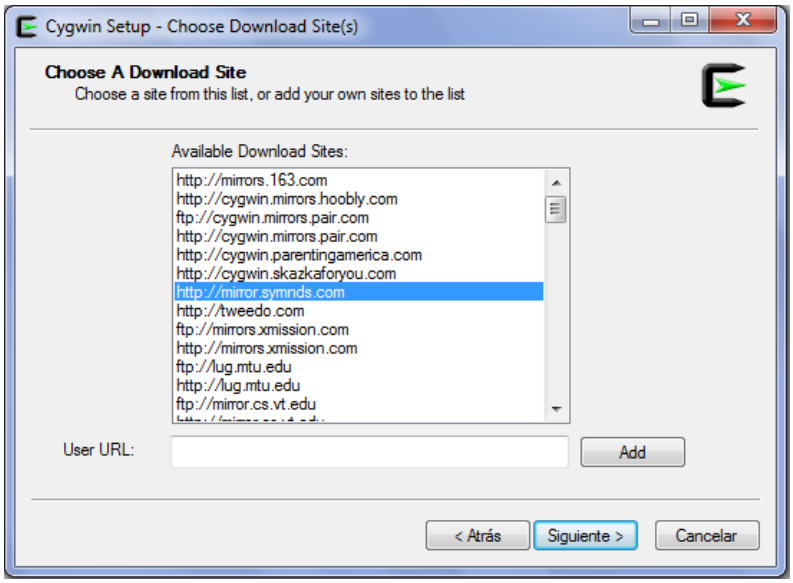


Indicamos el tipo de conexión a Internet de la que disponemos –directa, o a través de proxy; y los parámetros del proxy:



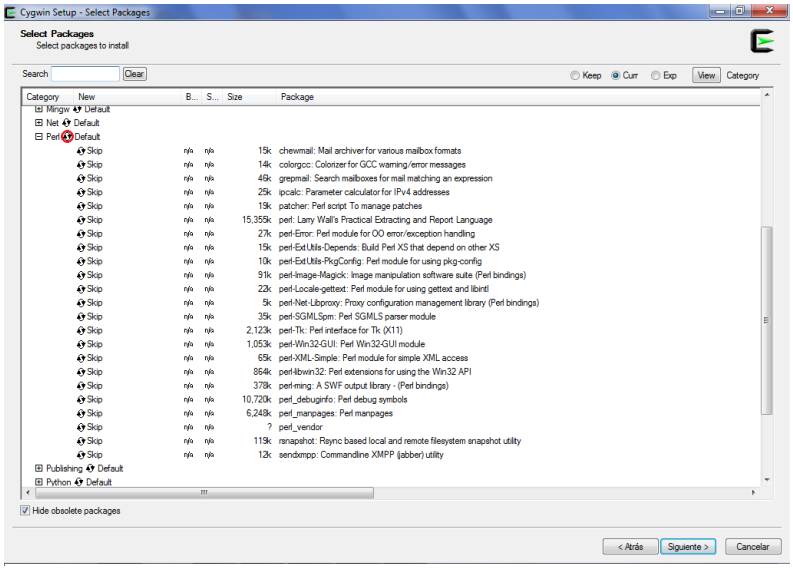
Si no sabemos que poner, indicamos "conexión directa".

El siguiente paso será indicarle la dirección de donde queremos descargar los paquetes de la lista de *mirrors*.



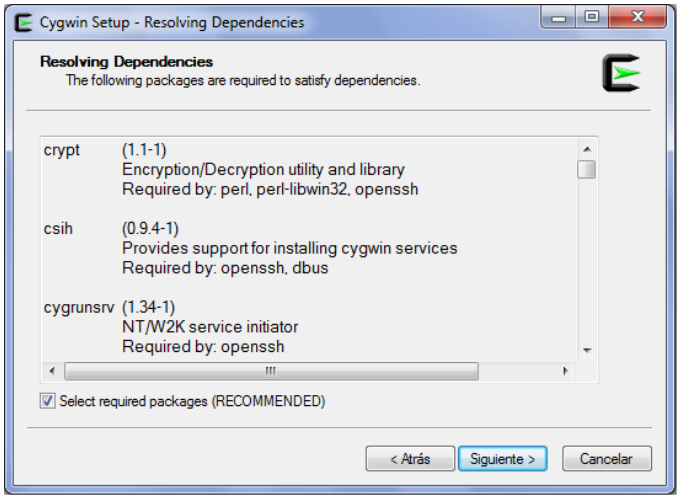
En principio debería ser indiferente el que escogamos.

Ahora el instalador entra en la pantalla de selección de paquetes. Bajamos hasta donde pone "Perl" y pulsamos solamente en el círculo indicado en rojo. Deberíamos ver una ventana como esta:



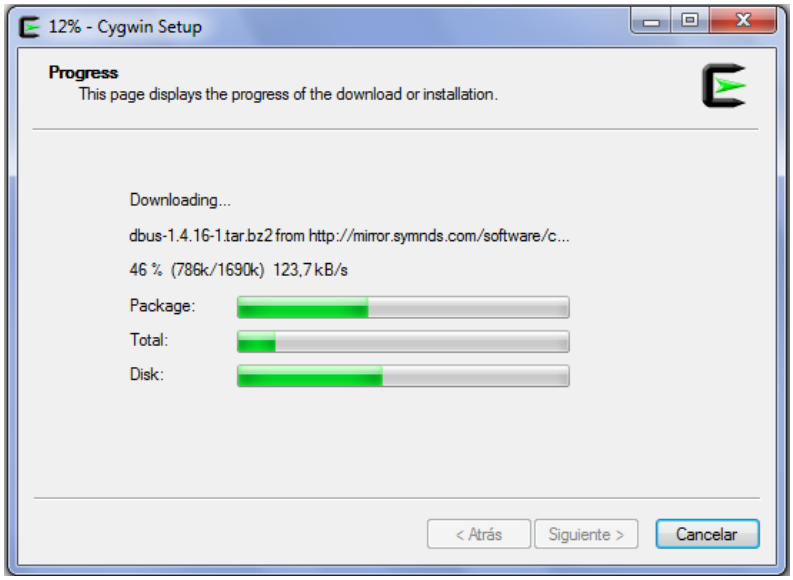
Con esto instalarás los paquetes necesarios para el funcionamiento de Perl.

Después aparecerá la pantalla de dependencias:



Seleccionamos la opción "Select required packages", y le damos al botón de "Siguiente".

Llegados a este punto es cuando el instalador empezará a descargar e instalar los paquetes necesarios para Perl.



Ahora nos lo tomamos con tranquilidad, porque esto llevará un tiempo considerable. Pero merecerá la pena: cuando el instalador termine, tendremos a Perl en marcha y listo para que programemos bajo Perl.

2.4. Actividades

El objetivo de este tema es que instales el entorno de Perl, lo que te permitirá avanzar en el curso realizando las prácticas.

La actividad puntuable de este tema es la instalación de Perl en sí.

Esta tarea la consideraré como "apta" cuando hayas instalado Perl en tu ordenador.

Localiza donde tienes instalado el intérprete de Perl en tu unidad física. La prueba de instalación la harás desde una consola de texto –`cmd` en el caso de Windows, terminal de texto en el caso de MacOS X o Linux–. Como prueba de realización de la práctica, me tienes que mandar es un pantallazo de consola, en la que teclearás:

```
perl -V
```

Pasará algo. Entonces tecleas "Manuel Molino Milla", sacas una captura de pantalla de la consola de texto donde has hecho todo esto, me lo mandas a través del campus virtual, y cierras la consola.

Si tienes tiempo, intenta compilar el "Hola mundo!" de ejemplo del artículo anterior. Si tienes dudas, ponlas en el foro. Pero recuerda, la actividad puntuable **no** es el "Hola mundo!", sino la de `perl -V` y tu nombre.

Es muy importante que recuerdes dos cosas sobre el uso del foro:

- Resolver dudas de los compañeros lo tendré en cuenta a la hora de eximiros de actividades, marcaros actividades como aptas, o poneros como aptas actividades dudosas.
- El que me ponga la solución de un ejercicio completa en el foro, le marco una actividad apta como no apta, automáticamente. Es bueno que ayudéis a los compañeros a que hagan ellos solos las actividades, no que les hagáis vosotros las actividades.

2.5. Qué viene después

El próximo paso será comenzar a ver como hacer programas sencillos en Perl. En poco tiempo estarás haciendo virguerías en Perl.

Capítulo 3

Introducción a la programación en Perl

Vamos a comenzar a ver la programación en Perl desde el principio. Para ello veremos las bases que forman cualquier lenguaje, y que nos permitirán pasar a las sentencias de control.

3.1. Los tipos en Perl

El concepto de "tipo" en Perl no coincide con el concepto de "'tipo" de otros lenguajes de programación. En los lenguajes de programación tipados comunes, tenemos como tipos los enteros, los caracteres, o los números en coma flotante, entre otros ejemplos de tipos. Por otro lado, en los lenguajes no tipados en una variable se puede meter de todo.

En Perl, los tipos básicos son algo peculiares en su filosofía. Estos son:

- **Escalar:** escalar es todo aquello que se compone de solo un miembro: un entero, un flotante, un carácter, un objeto, o una cadena de caracteres, por poner algunos ejemplos[1]. Todo escalar está precedido por el carácter `$` .
- **Lista:** lista es todo aquello que se compone de varios elementos, organizados en orden. En Perl, una variable asociada a este tipo se puede utilizar

[1] O una lista, o una matriz asociativa. Más adelante volveremos sobre los tipos básicos, porque he sobresimplificado mucho la explicación de los tipos para una primera aproximación. Como todo en Perl, hay capas; nos podemos quedar aquí, o podemos profundizar más. Lo bueno de Perl es que nos podemos quedar trabajando en cualquier capa, sobresimplificando, y seguir haciendo programas útiles.

como si fuera un vector, una lista, una pila, una cola o un conjunto; el lenguaje nos da instrucciones para utilizar los mecanismos de cualquiera de estos tipos sobre una variable de tipo lista. Las listas pueden contener elementos que sean a su vez listas; por lo que las listas también pueden ser utilizadas como árboles, como bosques, o como grafos; y también el lenguaje nos da instrucciones para utilizar cualquiera de los mecanismos de estos tipos sobre una lista. Realmente, una lista es todo aquello que tiene más de un elemento. Dada la flexibilidad conceptual de las listas en Perl, en el texto de este curso emplearemos varios términos para referirnos a lista; matriz, vector, lista, pila, cola o árbol. Cuando utilicemos un término concreto de estos en el texto, nos estamos refiriendo a una lista sobre la que accedemos utilizando los comandos de Perl para dicho tipo. Todos los nombres de lista están precedido por el carácter `@`.

- *Matriz asociativa -Hash-:* una matriz asociativa es como una lista, solo que los elementos se pueden referenciar también través de una palabra clave, además de a través de su posición. Como los elementos pueden ser listas y matrices asociativas también, las matrices asociativas también implementan los tipos de datos "árbol", "'bosque" y "grafo", cuando necesitamos seleccionar la rama también por un atributo de la rama. Toda matriz asociativa comienza con la variable `%`.

En Perl no es necesario declarar las variables antes de su uso. Esto es muy práctico cuando estamos haciendo un script Perl rápido, de media docena de líneas, para sacar adelante un trabajo de urgencia –algo que haremos cuando sepamos Perl con más frecuencia de la que podemos estimar ahora–. Sin embargo, cuando ya el código no te cabe en la pantalla, es una muy mala idea que las variables no sean declaradas; ya que un error muy frecuente y difícil de depurar es equivocarse al teclear el nombre de la variable en algún punto del programa, y volverse loco buscando un error de lógica del código cuando se trata de un error sintáctico fácil de corregir. La forma de evitar esto, es diciéndole a Perl que la declaración de variables es obligatoria. Esto lo haremos añadiendo la línea:

```
use strict;
```

Esta linea debería estar entre las tres primeras lineas de todo programa Perl que hagas durante el próximo año al menos. Y ojo, que no es la "L" del conductor novel; sino el cinturón de seguridad de tu coche en un derby de demolición. **Creeme, Manuel: lo vas a necesitar.**

3.2. Las variables escalares en Perl

Declaramos una variable escalar con:

```
my $mivariable;
```

El uso es como en cualquier lenguaje, recordando que siempre que la usemos tenemos que poner el caracter `$` antes del nombre de la variable, sin separación:

```
my $uno=1;
my $dos=7;
my $resultado;

$resultado=$uno+$dos+17;

print $resultado;
```

el resultado de esta operación es `25`.

Por otro lado, también podemos operar con números no enteros:

```
my $uno=1.2;
my $dos=7.3;
my $resultado;

$resultado=$uno+$dos+17.1;

print $resultado;
```

el resultado es `25.6`.

Finalmente, también podemos trabajar con cadenas. Por ejemplo, podemos hacer:

```
my $uno="cadena";
my $dos="otracadena";
my $resultado;

$resultado=$uno."␣" .$dos."␣y␣esta␣última";


print $resultado;
```

siendo el resultado `cadena otracadena y estaúltima`. Observamos que concatenamos cadenas con el punto –`.`–. Realmente es una de las formas de concatenar; en Perl siempre hay más de una forma de hacer lo mismo. La otra forma es:

```
my $uno="cadena";
my $dos="otracadena";
my $resultado;

$resultado="$uno␣$dos␣y␣esta␣última";


print $resultado;
```

Esto se debe a que cuando Perl se encuentra una cadena entre comillas dobles, interpreta y expande las variables que encuentra dentro –es decir, las sustituye por su valor–. Esto no funciona solo con cadenas, sino con cualquier variable escalar: por ejemplo, si hacemos:

```
my $x=2.3;
my $y=-1.8;
my $z=0.2;

my $cadena="la␣coordenada␣del␣átomo␣es␣($x,$y,$z)\n";

print $cadena
```

Obtendremos como salida:

```
la coordenada del átomo es (2.3,-1.8,0.2)
```

Si lo que queríamos era escribir en `$resultado` la cadena "$uno $dos y esta última", que contiene dos signos de dólar, deberíamos hacer:

```
my $uno="cadena";
my $dos="otracadena";
my $resultado;

$resultado='$uno␣$dos␣y␣esta␣última';


print $resultado;
```

Así Perl no expande los nombres de las variables. Realmente también hay otra forma de que no expanda los nombres de las variables, ya que siempre hay más de una forma de hacer lo mismo en Perl:

```
my $uno="cadena";
my $dos="otracadena";
my $resultado;

$resultado="\$uno␣\$dos␣y␣esta␣última";


print $resultado;
```

Como vemos, lo de que "siempre hay más de una forma de hacer lo mismo" es el día a día de Perl. En este caso, lo que estamos haciendo es "escapar" los signos de dólar, para quitarles sentido como carácter especial. En Perl tenemos caracteres especiales dentro de cadena que permiten introducir caracteres más delicados, como pueda ser el retorno del carro. Estos caracteres especiales son normalmente los mismos que en C; las más comunes son:

- `\n`: retorno de carro a una nueva línea.
- `\b`: retorno a la posición anterior.
- `\a`: pitido.
- `\t`: tabulación (8 espacios).

- \\: el caracter \
- ": las comillas dobles, dentro de cadenas delimitadas por comillas dobles.
- ': las comillas simples, dentro de cadenas delimitadas por comillas dobles.

En cadenas delimitadas por comillas simples, las comillas simples deben ir precedidas de\ , y las comillas dobles no. Por otro lado, en cadenas delimitadas por comillas dobles, las comillas dobles deben ir precedidas de \ , y las comillas simples no.

Veamos un ejemplo de caracteres especiales escapados: podemos emplear en una cadena el \n –como en C– para obtener retornos de carro. Por ejemplo, podemos hacer:

```
my $minariz;

$minariz="\n Érase un hombre a una nariz pegado,\nérase
   una nariz superlativa,\nérase una nariz sayón y escriba
   ,\nérase un peje espada muy barbado.\n\n Era un reloj
   de sol mal encarado,\nérase una alquitara pensativa,\
   nérase un elefante boca arriba,\nera Ovidio Nasón más
   narizado.\n";

print $minariz;
```

Para obtener un retorno de carro al final de cada verso –la cadena no tiene saltos de línea internas, va entera en la misma línea–. Observamos que no hay un espacio después del carácter especial para separarlo de la siguiente palabra: Perl ya sabe que el carácter especial \n Tiene solo un carácter. De cualquier forma, siempre hay más de una forma de hacer lo mismo, y unas son más cómodas que otras: a veces tendremos textos largos, y como vemos esto es poco operativo. Perl permite utilizar cadenas que ocupen más de una línea, por lo que el código anterior sería equivalente a:

```
my $minariz;

$minariz="
```

```
␣Érase␣un␣hombre␣a␣una␣nariz␣pegado,
érase␣una␣nariz␣superlativa,
érase␣una␣nariz␣sayón␣y␣escriba,
érase␣un␣peje␣espada␣muy␣barbado.
␣Era␣un␣reloj␣de␣sol␣mal␣encarado,
érase␣una␣alquitara␣pensativa,
érase␣un␣elefante␣boca␣arriba,
era␣Ovidio␣Nasón␣más␣narizado.

";

print $minariz;
```

En Perl podemos declarar un delimitador de cadena; y dentro de dicho delimitador, cualquier cosa vale, mientras que no aparezca dicho delimitador: valen las comillas simples, las comillas dobles...Perl expandirá las variables, y sustituirá los caracteres especiales.

La sintaxis de declaración del delimitador será:

```
<<NUEVODELIMITADOR
cadena multilinea
NUEVODELIMITADOR
```

Esto puede ir en cualquier lugar en el que puede ir cualquier tipo de cadena. Por ejemplo, podemos hacer:

```
my $minariz;

$minariz=<<FINPOEMA

 Érase un hombre a una nariz pegado,
érase una nariz superlativa,
érase una nariz sayón y escriba,
érase un peje espada muy barbado.
 Era un reloj de sol mal encarado,
érase una alquitara pensativa,
érase un elefante boca arriba,
```

```
era Ovidio Nasón más narizado.

FINPOEMA
;
print $minariz;
```

Cuando trabajemos con bases de datos veremos como esto nos permitirá hacer legible el código con *queries* de SQL inmensas.

3.3. Listas en Perl

Las listas en Perl son realmente fáciles de usar, y realmente flexibles. Pueden ser usadas con semántica de matriz -accediendo por índice-, con semántica de pila -accediendo sólo secuencialmente, desmontando la estructura en el acceso- o con semántica de lista -accediendo secuencialmente sin desmontar la estructura en el acceso-, según más nos convenga, entre otros usos y semánticas de acceso –hay muchos más–.

Definimos la matriz/lista/pila/cola vacía con:

```
@mimatriz=();
```

Y una lista con un elemento por cada palo de la baraja:

```
@palos=('oros','bastos','copas','espadas');
```

podemos montar también una lista con variables:

```
my $escalar1='i';
my $escalar2='j';
my $escalar3='k';

my @lista=($escalar1,$escalar2,$escalar3);
```

lo que crea una nueva lista cuyo contenido son tres elementos: las letras 'i', 'j' y 'k'.

los elementos de una matriz pueden ser de cualquier tipo; podemos también tener listas de enteros:

```
@misnument=(3,4,5,2);
```

o de flotantes:

```
@misnumfloat=(3.4,2.3,71.2,5.2);
```

y podemos mezclar cualquier tipo dentro de una lista:

```
@tiposvarios=('hola',34,4.53,'adios',17.3,44);
```

Los elementos dentro de una lista no tienen por que ser necesariamente escalares. Podemos montar listas/matrices/pilas/colas de cualquier tipo de elementos:

```
@listaanidada=('hola',('esto␣es','una',('estructura')
   ,17,89,2.3),3,4,"adios");
```

O a través de variables:

```
my @listaanidadaC=('a',('s',2),17);
my $escalar='j';
my $escalar2='k';
my $escalar3='l';

my  %hash;

my @otralistanidada=($escalar,23,($escalar,$escalar3,
   @listaanidadaC),$escalar2,%hash,$escalar3);
```

pudiendo utilizarse cualquier variable o constante de cualquier tipo, tantas veces como deseemos. Esto nos permitirá definir casi cualquier estructura de datos acíclica. Hemos sobresimplificado las estructuras no lineales en esta primera aproximación, ya que realmente almacenamos referencias en estos casos; lo que tendrá algunos efectos importantes. Este será un tema que revisitaremos más adelante, cuando aprendamos a usar las referencias, lo que nos permitirá definir estructuras de datos cíclicas con las listas de Perl.

3.4. Métodos de acceso a una matriz por índice

Podemos acceder los elementos como en las matrices[2] en C, es decir, por un índice. Por ejemplo, la instrucción:

```
@palos=('oros','bastos','copas','espadas');
print @palos[1];
```

imprimiría en pantalla la palabra `bastos`.

Y aquí es donde podemos meter la pata si no entendemos lo que estamos haciendo. Cuidado con esto, que ahora damos el "salto al mundo Perl".

¿Por qué carácter comienza `@palos[1]`? Por `@`. Eso significa que `@palos[1]` es una lista. Corresponde con lista con un único elemento, que es el contenido de la segunda posición de la matriz `palos`.

Ojo, Manuel; que esto es una mina del lenguaje. En el sentido militar, no en el geológico. Si este es tu primer contacto con Perl, rebobina al inicio de la sección y lee de nuevo el ejemplo para asegurarte que lo entiendes. La diferencia entre entender bien esto y no entenderlo es la diferencia entre Perl como herramienta de programación y Perl como herramienta para flagelarte.

En Perl, la variable no es `@palos`. La variable es `palos`. Que tenga una arroba antes lo que nos dice es cómo tenemos que intrepretarla. No todas las interpretaciones tendrán sentido. Y a veces, tendremos que encadenar correctamente varios símbolos previos para que Perl haga lo que queramos. Por eso es importante entender desde un principio esto, que es la principal complejidad de Perl.

Si queremos devolver un escalar, recordando lo que hemos indicado sobre los tipos en Perl, debemos usar:

```
@palos=('oros','bastos','copas','espadas');
print $palos[1];
```

es decir, hemos cambiado el `@` del nombre de la matriz por `$`, consiguiendo así un escalar en lugar de una matriz.

[2]En la universidad probablemente te hallan enseñado esto con otro nombre: `array`. `Array` no deja de ser el término inglés para denominar lo que en español es una matriz. Peor es lo de `library`, que a veces te lo traducen como "librería", cuando realmente significa "biblioteca". La traducción correcta mantiene además la analogía con la realidad: en una biblioteca entras, y tomas prestado lo que necesitas de lo disponible. Si haces eso en una librería, probablemente el guarda de seguridad de la puerta tenga algo que opinar respecto a tu acción.

Resumiendo: si en el acceso al elemento ponemos un `@` antes, el valor devuelto será una lista. Si ponemos un `$` antes, el valor devuelto será un escalar.

Debemos tener cuidado al usar la matriz porque, como en C, comenzamos a contar los índices por `0`. Por ejemplo, recorremos la matriz anterior con:

```
@palos=('oros','bastos','copas','espadas');
print $palos[0];
print $palos[1];
print $palos[2];
print $palos[3];
```

Podemos obtener submatrices de una matriz seleccionando los elementos que queramos a través de los índices. Para ello, indicamos los indices que queremos obtener uno a uno. Por ejemplo, si tenemos la matriz:

```
my @numeros=('cero','uno','dos','tres',
             'cuatro','cinco','seis','siete');
```

Obtenemos una lista con los números impares con:

```
@numeros[1,3,5,7]
```

Ahora sí usamos la arroba, porque lo que queremos es una lista.

También podemos acceder a una matriz indicando un rango de elementos, en lugar de los elementos uno a uno. Por ejemplo, para indicar los elementos contenidos entre el segundo y el quinto hacemos:

```
@numeros[1..4]
```

definimos el rango con el operador `..`, que devuelve los elementos comprendidos entre el operando izquierdo y el operando derecho, ambos inclusive. Sí, entre el segundo y el quinto. Recordemos que las matrices comienzan por cero, por lo que el primer elemento de `@numeros` se indexará con `@numeros[0]`.

Podemos mezclar las nomenclaturas anteriores, incluso superponiendo elementos en la selección, seleccionando así un elemento más de una vez. Por ejemplo, podemos hacer:

```
@numeros[1..4,1,3,5,7]
```

obteniendo así una lista que contiene todos los elementos entre el segundo y el quinto, así como los elementos impares de la lista `@numeros`.

3.5. Accediendo a listas, colas y pilas

Perl permite otros métodos de acceso a una matriz más acordes con una semántica de lista, de pila o de cola. Hay muchos, por lo que no estudiaremos todos, solo os que son más útiles.

Recordemos que listas, pilas, colas y matrices son lo mismo, secuencias ordenadas de datos. Una variable la podemos tratar como una lista, para tratarla poco después como una pila, para tratarla después como una matriz. Sin conversión de tipos de datos. Por dentro, a nivel de como se estructuran los datos, son exactamente lo mismo: la diferencia es en qué operaciones usamos. Esto suele ser común en Perl.

La primera función que destacamos es la función `pop`. Esta función tiene un único parámetro, que es una lista –que se comportará como una pila–. Su sintaxis es:

```
pop @lista
```

esta función devuelve el último elemento de la pila; eliminando este elemento ya extraído de la pila.

También podemos extraer el primer elemento de la lista –con lo que se comportará como una cola–, con el comando:

```
shift @lista
```

que devuelve el primer elemento de la cola, eliminando dicho elemento ya extraído de la cola.

Podemos insertar elementos en la lista; donde se inserte depende de la instrucción que empleemos. Se comportará la variable de tipo lista como se comportaría el tipo de datos correspondiente.

Si deseamos insertar elementos al final de la lista, hacemos:

```
push @lista, $elemento;
```

Podemos insertar más de un elemento al final de la lista, con la sintaxis:

```
push @lista, $elemento1, $elemento2, $elemento3;
```

También podemos insertar elementos al principio de la lista, con el comando:

```
unshift @lista, $elemento;
```

Podemos insertar más de un elemento al principio de la lista, con la sintaxis:

```
unshift  @lista, $elemento1, $elemento2, $elemento3;
```

Es importante destacar que podemos insertar datos con la orden de cola, y extraerlos con la de lista. No es que el dato que insertamos de una forma solo lo podemos sacar como lo hemos insertado –idea más propia de un lenguaje perpetrado por alguien demasiado adepto a la teoría–; sino que tenemos mecanismos para hacer lo que nos apetezca con la estructura de datos. Las combinaciones de estos comandos nos permiten implementar los tipos básicos de estructuras lineales:

- Estructura LIFO –*Last In, First Out*–. Último en entrar, primero en salir. Es la estructura pila tradicional. Corresponderá con cualquier lista a la que accedamos insertando los datos con `push` y extrayéndolos con `pop`.
- Estructura FIFO –*First In, First Out*—. Primero en entrar, primero en salir. Es la estructura cola tradicional. Corresponderá con cualquier lista a la que accedamos insertando los datos con `unshift` y extrayéndolos con `pop`.

En principio, estas dos formas de acceder a la estructura son las más comunes. Sin embargo, como también podemos acceder secuencialmente a la estructura, es importante el orden interno en el que estén los datos, por los que en la práctica podemos incluir dos tipos adicionales de estructura:

- Estructura FILO –*First In, Last Out*—. Primero en entrar, último en salir. Es la estructura pila tradicional, y se comporta igual que LIFO si sólo extraemos datos con esta semántica; pero los datos se almacenan internamente en el orden inverso al de la estructura LIFO, lo que debemos tener en cuenta al acceder a la estructura a través de sus índices. Corresponderá con cualquier lista a la que accedamos insertando los datos con `unshift` y extrayéndolos con `shift`.

- Estructura LILO –*Last In, Last Out*—. Último en entrar, último en salir. Es la estructura cola tradicional, y se comporta igual que FIFO si sólo extraemos datos con esta semántica; pero los datos se almacenan internamente en el orden inverso al de la estructura FIFO, lo que debemos tener en cuenta al acceder a la estructura a través de sus índices. Corresponderá con cualquier lista a la que accedamos insertando los datos con `push` y extrayéndolos con `shift`.

3.6. Revisitando el programa "Hola Mundo!"

Un programa en Perl que imprima en pantalla la cadena "Hola Mundo!" es algo tan sencillo como:

```
print "Hola Mundo";
```

Esto ya es un programa en Perl que imprime en pantalla "Hola Mundo!". Sin embargo, estamos haciendo trampas: un programa en Perl suele necesitar más código que apenas esto. Una segunda aproximación al "Hola Mundo!" en Perl es:

```
#!/usr/bin/perl -w

# Esto debería ser obligatorio...
use strict;
use warnings;
use English;

print "Hola Mundo";
```

Esto se parece más a un programa en Perl; aparecen cuatro lineas nuevas. Una realmente ya ha aparecido antes, aunque no hayamos hablado de ella; de las otras tres no.

La primera linea de todo programa en Perl suele ser:

```
#!/usr/bin/perl -w
```

el hecho de incluir esta linea permite lanzar en Unix un programa en Perl como si se tratase de un programa compilado, sin necesitar utilizar explícitamente

el intérprete, si le activamos los privilegios de ejecución con `chmod`. La opción `-w` hace que los errores sean un poco más verbosos, facilitando la programación. Es una línea que, aunque no es obligatoria, deberíamos incluir siempre para facilitarnos la vida.

Después vemos un comentario en Perl. Ya han aparecido comentarios en algún ejemplo, En Perl, todos los comentarios deben comenzar por el carácter `#`, y terminan con el fin de línea. Perl es un lenguaje en el que se pueden escribir las cosas de forma legible, de difícil compresión, o hacer incluso código de solo escritura[3]. Por ello, es muy recomendable que comentes las cosas un poco más extrañas que hagas, o dentro de unos meses ni tú entenderás tu propio código.

Después hay unos `use`. De momento debemos saber que `use` permite incluir bibliotecas en Perl, y que estos tres `use` que vemos en el ejemplo deberíamos ponerlos siempre, ya que nos ayudarán en la programación. El `use warnings` es autoexplicativo, con que sepamos algo de inglés: nos dará avisos de cosas que sean "feas". Ya hemos hablado del `use strict`, y el `use english` será de mucha utilidad, ya que nos permitirá emplear nombres un poco más descriptivos con variables de uso común al trabajar con expresiones regulares. Volveremos a `use english` cuando hablemos de expresiones regulares.

Recordamos, al llegar en este punto por si no lo has intuido ya: en Perl, como en C, separamos las instrucciones por el carácter `;`.

3.7. Utilizando paquetes en Perl

Los paquetes son la característica más potente en Perl[4]. Un paquete es un conjunto de rutinas y su mecanismo de inicialización. Es similar al concepto de biblioteca[5] en otros lenguajes de programación, solo que además incluye un

[3] Frente a C que mantiene su concurso de código ofuscado, y otros lenguajes que tienen concursos florecientes de código ofuscado, el concurso de código ofuscado de Perl duró poco, y dejó de celebrarse hace más de diez años. Viendo la perfección en la ofuscación alcanzada desde las primeras entregas, sospecho que se debe a que un concurso de código ofuscado en Perl es como correr los cien metros vallas con un coche de F1. Por su filosofía de que hay más de una forma de hacerlo todo, y por determinadas cosas que no terminaremos de explicar en el curso, en Perl puede hacerse código realmente difícil de leer. Si es lo que queremos, podemos siempre usar el módulo `Lingua::Romana::Perligata`. Pero por favor, Manuel: pon comentarios donde hagas algo raro. Aunque uses el módulo `Lingua::Romana::Perligata`. Te gustará ser capaz de leer dentro de seis meses el código que programastes ahora.

[4] No el concepto de paquete en sí, que tienen todos los lenguajes; sino por los millares de paquetes libres que podemos importar de CPAN.

[5] Mal llamada librería en muchas clases de informática, y no pocos textos de informática.

código de inicialización; que es algo específico de Perl y aprenderemos como se hace y para qué sirve cuando hablemos sobre como hacernos nuestros paquetes.

Usar un paquete es tan sencillo como hacer:

```
use nombredepaquete;
```

Y podemos usar todas las funciones del paquete indicado.

3.8. Definiendo características con `use`

Algunas características específicas de Perl se activan con la misma sintaxis de el uso de paquetes, es decir, con `use`. También se desactivan con la opción que elimina la importación de un módulo, el comando no `no` –en Perl, del mismo modo que se puede cargar un paquete, se puede descargar un paquete. Y se puede hacer en cualquier parte del código. Esto nos permite intuir por donde irán los tiros en el código de inicialización de paquetes, pero no nos preocupemos: lo veremos más adelante–. Los ejemplos más importantes de características controlables con `use` son:

```
use locale;
```

Que usa los parámetros de configuración local, definidos según el estándar POSIX.

```
use integer;
```

Que fuerza a que las operaciones entre enteros siempre darán enteros.

```
use diagnosis;
```

Lo que aumenta el número de comprobaciones del código de Perl al ejecutarse, y la verbosidad de estas comprobaciones.

```
use sigtrap;
```

Este `use` permite realizar tratamiento de señales. Podemos indicar con `qw` las señales que queremos capturar. Lo del `qw` lo comentaré con detalle en un tema posterior.

```
use strict;
```

Prohíbe utilizar las construcciones de Perl que potencialmente pueden ser errores. Podemos indicar con `qw` sobre que aspecto queremos un análisis estricto de la sintaxis.Lo del `qw` lo comentaré con detalle en un tema posterior.

```
use vars;
```

Con este `use` podremos indicar entre paréntesis qué variables serán exportadas a otros módulos desde el nuestro.

Finalmente así aumentaremos el número de avisos generados por Perl sobre cosas "sospechosas" de ser errores, en el código:

```
use warnings;
```

3.9. Operadores en Perl sobre números

Tal y como ya hemos comenzado a estudiar, en Perl asignamos idénticamente a como lo hacemos en C; es decir, con el operador =. También tenemos casi los mismos operadores aritméticos que en C:

- +: Operador de suma.
- −: Operador de resta.
- ⋆: Operador de multiplicación.
- /: Operador de división.

- `%`: Aritmética modular. Devuelve el resto de dividir el primer número entre el segundo.
- `**`: Potencia –el primer número elevado al segundo–. Este operador no existe en C.
- `..`: Operador de rango. Devuelve el rango contenido entre el primer valor y el segundo. Los elementos deben ser escalares del mismo tipo, y además deben ser forzosamente enteros o caracteres. No funciona con números en coma flotante ni con cadenas. Este operador no existe en C.

Estos operadores son autoexplicativos, y no entraremos en mucho detalle.

En Perl también tenemos el operador de autoincremento y el de autodecremento:

```
$a++;
```

como en C, `$a++;` incrementa el valor de `$a`; y:

```
$a--;
```

también como en C, `$a--;` decrementa el valor de `$a`.

3.10. Operadores y funciones en Perl para cadenas

Los operadores y las funciones de cadenas son interesantes: tenemos, además del operador de concatenación del que ya hemos hablado, varios operadores de uso común. No vamos a verlos todos porque son muchísimos; pero sí comentaré los más importantes:

```
chomp(cadena)
```

Elimina la terminación de la cadena que pasamos como parámetro. Por ejemplo:

```
my $cadena="Hola mundo\n";

chomp($cadena);
```

```
print $cadena." y a los otros planetas.\n";
```

Por otro lado, si queremos eliminar el último carácter de la cadena y recuperar el carácter eliminado emplearemos:

```
chop(cadena)
```

```
my $cadena = "Esto es una prueba";
my $c = chop($cadena);
print "$c";
```

Para convertir de número a ASCII lo podemos hacer con:

```
chr(número)
```

Por ejemplo:

```
 my $cod = 67;
print chr($cod);
```

Lo que imprimirá la letra "C".

Podemos convertir a minúsculas con:

```
lc(cadena)
```

Por ejemplo:

```
print lc("HOLA");
```

Imprimirá "hola" en pantalla.

También podemos convertir a mayúsculas con:

```
uc(cadena)
```

Por ejemplo:

```
print uc("hola");
```

Imprimirá "HOLA" en pantalla.

Son también posibles las conversiones de solo la primera letra; por ejemplo, pasamos la primera letra a mayúscula con:

```
ucfirst(cadena)
```

Por ejemplo:

```
print ucfirst("pepe");
```

Imprimirá "Pepe" en pantalla.

Finalmente, el equivalente en minúsculas –convertir solo la primera letra–, sería:

```
lcfirst(cadena)
```

Por ejemplo:

```
print lcfirst("TODO");
```

Imprimirá "tODO" en pantalla.

Siguiendo con las funciones útiles de cadenas, podemos saber el número de caracteres de una cadena con:

```
length(cadena)
```

Por ejemplo:

```
my $cadena = "Esto es una prueba";
print length($cadena);
```

imprime el número de caracteres de la cadena `$cadena`. Es importante que recordemos que si trabajamos en UTF-8, el tamaño de una cadena y su número de caracteres son dos datos distintos, y entre los que no existe proporción lineal, ya que UTF-8 emplea un número distinto de bytes para almacenar caracteres distintos, según la página de la tabla UNICODE en la que estén.

Podemos buscar subcadenas dentro de otras cadenas mediante la función:

```
index(cadena, subcadena, posición)
```

Esta función devuelve la posición de la primera aparición de `subcadena` dentro de `cadena`, a partir de la posición de inicio `posición`. Por ejemplo:

```
my $cadena = "Hola, mundo, hola a todo el mundo.";
my $posicion = index($cadena,"mundo",12);
print $posicion;
```

Si queremos buscar desde el inicio de la cadena indicada la subcadena buscada, basta con no indicar el parámetro `posición`:

```
index(cadena, subcadena)
```

Por ejemplo:

```
my $cadena = "Hola, mundo, hola a todo el mundo.";
my $posicion = index($cadena,"mundo");
print $posicion;
```

Ahora localizará la primera aparición de la palabra "mundo", cuando el ejemplo anterior localizaba la segunda aparición.

Estas dos operaciones se pueden hacer también con expresiones regulares –en Perl siempre hay más de una forma de hacer lo mismo–; las veremos en un tema próximo.

```
rindex(cadena)
```

Si lo que queremos es buscar de atrás adelante, lo podemos hacer con:

```
rindex(cadena, subcadena, posición)
```

Esta función devuelve la posición de la última aparición de `subcadena` dentro de `cadena`, a partir de la posición de inicio `posición`, de dicha posición hacia atrás. Por ejemplo:

```
my $cadena = "Hola, mundo, hola a todo el mundo.";
my $posicion = rindex($cadena,"mundo",18);
print $posicion;
```

Si queremos buscar desde el final de la cadena indicada la subcadena buscada, basta con no indicar el parámetro `posición`:

```
rindex(cadena, subcadena)
```

Por ejemplo:

```
my $cadena = "Hola, mundo, hola a todo el mundo.";
my $posicion = rindex($cadena,"mundo");
print $posicion;
```

Ahora localizará la última aparición de la palabra "mundo", cuando el ejemplo anterior localizaba la segunda aparición, contando desde el final de la cadena.

Estas dos operaciones también se pueden hacer también con expresiones regulares –en Perl siempre hay más de una forma de hacer lo mismo, como estamos repitiendo constantemente–.

Podemos extraer subcadenas mediante el comando:

```
substr(cadena, desplazamiento,longitud)
```

Función que devolverá la subcadena de `cadena` que, comenzando por `desplazamiento`, tiene exactamente `longitud` caracteres. Por ejemplo:

```
my $cadena = "Hola, mundo, hola a todo el mundo.";
my $posicion = substr($cadena, 6,5);
print $posicion;
```

Imprimirá la cadena "mundo".

Por otro lado, si utilizamos:

```
substr(cadena, desplazamiento)
```

Es decir, si no especificamos la longitud, devolverá la subcadena de `cadena` que comience por `desplazamiento`, llegando hasta el final de `cadena`. Por ejemplo:

```
my $cadena = "Hola, mundo, hola a todo el mundo.";
my $posicion = substr($cadena, 6);
print $posicion;
```

Imprimirá la cadena "mundo, hola a todo el mundo.".

Una función muy útil es:

```
join(cadena,lista)
```

Que devolverá una cadena resultante de concatenar los elementos de la lista pasada como segundo parámetro, introduciendo entre ellos la cadena indicada como primer parámetro. Por ejemplo:

```
 my @lista = ("Hola","a","todo","el","mundo");
my $lista = join(" ",@lista);
print $lista;
```

Lo que imprimirá la cadena "Hola a todo el mundo". Existe el complementario, `split`, que convierte una cadena en una lista empleando expresiones regulares para decidir el criterio para "partir" en trozos dicha cadena. Veremos dicho operador cuando estudiemos las expresiones regulares, y te aviso, Manuel, que usarás mucho el `split` de Perl si administras sistemas.

Terminaremos este tema estudiando el operador de despliege. Este operador es el `x`, y permite repetir una cadena un número determinado de veces. Por ejemplo, si Bart quiere poner en la variable `$pizarra` mil veces el valor "No le meteré petardos al director en el bocata", le basta con hacer:

```
$pizarra = "No le meteré petardos al director en el bocata
   \n" x 1000;
```

cumpliendo con celeridad, sin trabajos ni bucles, el odiado castigo. También se puede hacer esto con bucles; el Perl, siempre hay más de una forma de resolver el mismo problema. Pero eso queda para el próximo tema.

3.11. Actividades

Este tema no tiene ninguna actividad puntuable; las prácticas son demasiado sencillas, y si lo fusiono con el siguiente queda un tema demasiado espeso. Por lo que para hacerte perder el tiempo, no te obligaré a hacer una actividad puntuable. Lo que no quita que sea recomendable para que puedas aprovechar el tema que teclees algunos ejemplos, los pruebes, y te familiarices con el entorno de trabajo y el lenguaje.

El objetivo de este tema es que entiendas una serie de conceptos clave. Es muy importante que leas el tema hasta entenderlo, y que preguntes las dudas. Si tienes interés en probar cosas concretas o hacer cosas, puedes hacerlo: hay ejemplos en este artículo que puedes probar, y ver el resultado. Insisto: quizás sea un buen momento para aprovechar ese entorno de Perl que tienes recién instalado, jugar un poco con el entorno, y ver si consigues echar a andar los ejemplos que incluyo en este tema.

Capítulo 4

Las sentencias de control en Perl

En el tema anterior aprendimos los conceptos básicos de Perl. En este tema vamos a aprender sus estructuras de control básicas, es decir, cómo realizar que no se ejecuten de forma lineal; y vamos a aprender a estructurar nuestros programas en Perl.

Las estructuras de control en Perl son muy fáciles de aprender para los programadores de Unix de la vieja escuela, así como a los programadores de C. Esto es así porque sintácticamente son muy similares a las que tiene C y algunos shells.

En Perl contamos con casi todas las estructuras de C; a excepción de `switch`, que no forma parte del Perl estándar –aunque hay una biblioteca que permite utilizarlo–. Contamos con alguna estructura de control más que no existe en C, como `foreach`; y también con estructuras que, aunque se pueden hacer equivalentes a otras de C, facilitan la legibilidad; como son `unless`, `until` y `do...until`. También el `if` de una sentencia, después de la sentencia guardada.

Vamos a estudiar en este tema una a una todas las estructuras de control de Perl. Pero comenzaremos por los operadores relacionales, que son los que nos permitirán delimitar condiciones de control complejas.

4.1. Operadores relacionales

Los operadores relacionales que soporta Perl son:

- `==`: Operador de igualdad. Cierto si los dos lados de la igualdad devuelven el mismo valor, falso en otro caso.
- `!=`: Operador de distinto. Cierto si los elementos a los lados del operador son distintos, falso en otro caso.
- `<`: Menor que. Cierto si el elemento de la izquierda es menor que el de la derecha, falso en otro caso.
- `>`: Mayor que. Cierto si el elemento de la derecha es menor que el de la izquierda, falso en otro caso.
- `<=`: Menor o igual. Cierto si el elemento de la izquierda es menor o igual que el de la derecha, falso en otro caso.
- `>=`: Mayor o igual. Cierto si el elemento de la izquierda es mayor o igual que el de la derecha, falso en otro caso.
- `<=>`: Relación. Devuelve `-1` si el elemento de la izquierda es menor que el de la derecha, `0` si los dos elementos son iguales, y `1` si el elemento de la izquierda es mayor que el de la derecha.

Estos operadores no funcionan con cadenas de caracteres. Para cadenas de caracteres, los operadores relacionales son:

- `eq`: Operador de igualdad. Cierto si los dos lados de la igualdad devuelven cadenas idénticas, falso en otro caso.
- `ne`: Operador de distinto. Cierto si las cadenas a los lados del operador son distintas, falso en otro caso.
- `lt`: Menor que. Cierto si la cadena de la izquierda es menor que la de la derecha, falso en otro caso.
- `gt`: Mayor que. Cierto si la cadena de la derecha es menor que la de la izquierda, falso en otro caso.

- `le`: Menor o igual. Cierto si la cadena de la izquierda es menor o igual que la de la derecha, falso en otro caso.
- `ge`: Mayor o igual. Cierto si la cadena de la izquierda es mayor o igual que la de la derecha, falso en otro caso.
- `cmp`: Relación. Devuelve `-1` si la cadena de la izquierda es menor que la de la derecha, `0` si las dos cadenas son iguales, y `1` si la cadena de la izquierda es mayor que la de la derecha.

4.2. Operadores lógicos

Los operadores lógicos de Perl son:

- `&&`: Y lógico. Cierto si los elementos en los dos lados son ciertos, falso en otro caso.
- `||`: O lógico. Falso si los elementos en los dos lados son falsos, cierto en otro caso.
- `!`: No lógico. Cierto si el elemento es falso, falso en otro caso.

4.3. Operadores de bit

Los operadores a nivel de bit de Perl son:

- `&`: Operador and a nivel de bit.
- `|`: Operador or a nivel de bit.
- `^`: Operador xor a nivel de bit.
- `~`: Operador not a nivel de bit.
- `<<`: Rotación a la izquierda, a nivel de bit.
- `>>`: Rotación a la derecha, a nivel de bit.

4.4. La sentencia selectiva

La sentencia selectiva en Perl es similar a la de C. A diferencia de la de C, siempre son necesarias las llaves en el bloque de instrucciones, aunque tengamos una única instrucción. La sintaxis del `if` básico en Perl es:

```
if(condición)
    {
    instrucción1;
    instrucción2;
    ...
    instrucciónN;
    };
```

donde condición puede ser una función o una expresión condicional. Por ejemplo, podemos hacer:

```
if($i<17)
    {
     $j=$i+17;
     print "Valor␣demasiado␣pequeño.\n";
    };
```

Podemos incluir también una cláusula `else` en el `if`, quedando como:

```
if(condición)
    {
    instrucción1;
    instrucción2;
    ...
    instrucciónN;
    }
else
    {
    instrucción1;
    instrucción2;
    ...
    instrucciónN;
```

```
    };
```

Por ejemplo, podemos hacer:

```
if($i<17)
    {
     $j=$i+17;
     print "Valor demasiado pequeño.\n";
    }
else
   {
    $j=$i;
    print "El valor es el adecuado.\n";
    };
```

También podemos anidar las cláusulas `if` con la sentencia `elsif`, que supone lo mismo que una contracción de `else` e `if`, lo que facilita la legibilidad. La sintaxis es:

```
if(condición)
    {
    instrucción1;
    instrucción2;
    ...
    instrucciónN;
    }
elsif
    {
       instrucción1;
       instrucción2;
       ...
       instrucciónN;
     }
     else
     {
       instrucción1;
       instrucción2;
```

```
        ...
        instrucciónN;
     };
```

Por ejemplo, podemos hacer:

```
if($i<17)
    {
     $j=$i+17;
     print "Valor demasiado pequeño.\n";
    }
elsif($i>32)
    {
     $j=$i-32;
     print "Valor demasiado grande.\n";
     }
    else
   {
        $j=$i;
        print "El valor es el adecuado.\n";
    };
```

4.5. Selectivas `unless`

Tenemos también una forma de simplificar la notación y hacer más legible una sentencia selectiva: usando la sentencia `unless`. Por ejemplo, el bucle:

```
if(!condición)
    {
    instrucción1;
    instrucción2;
    ...
    instrucciónN;
    }
else
    {
```

```
    instrucción1;
    instrucción2;
    ...
    instrucciónN;
    };
```

–observamos que la condición está negada–, puede ser sustituido por:

```
unless(condición)
    {
    instrucción1;
    instrucción2;
    ...
    instrucciónN;
    }
else
    {
    instrucción1;
    instrucción2;
    ...
    instrucciónN;
    };
```

donde la cláusula no está negada. Por ejemplo, podemos hacer:

```
unless(estatodobien())
    {
     print "Algo␣está␣mal";
     };
```

4.6. Selectivas compactas

Podemos poner una selectiva que afecte apenas a una sentencia, poniendo la selectiva inmediatamente después de la sentencia. Son las selectivas compactas. Su sintaxis es:

```
sentencia if condición;
```

Observamos que no necesitamos ponerle paréntesis a la condición. Por ejemplo, podemos hacer:

```
print "$i es mayor" if  $i>$j;
```

lo que es lo mismo que hacer:

```
if ($i>$j)
   {
    print "$i es mayor";
   };
```

Por otro lado, podemos hacer lo mismo con la selectiva `unless`. La sintaxis del `unless` compacto es:

```
sentencia unless condición;
```

No es necesario poner entre paréntesis la condición. Por ejemplo, podemos hacer:

```
print "No es correcto" unless  correcto();
```

Lo que es lo mismo que hacer:

```
unless (correcto())
   {
    print "No es correcto";
    };
```

Lo que a su vez es lo mismo que hacer:

```
if  (!correcto())
   {
    print "No es correcto";
    };
```

4.7. Las sentencias iterativas

Como en el caso de la sentencia selectiva, las sentencias iterativas en Perl son similares a las de C, contando con que en Perl siempre el bloque tiene que estar

entre llaves.

El bucle `while` en Perl tiene como sintaxis:

```
while(condición)
  {
   instrucción1;
   instrucción2;
   ...
   instrucción;
  };
```

Se ejecutan las instrucciones entre llaves -el bloque- mientras que `condición` sea cierto. Antes de cada ejecución de las sentencias entre llaves se comprueba si la condición es cierta; si lo es, ejecuta de nuevo las sentencias. Si no, sale del `while`. Y se repite iterativamente el proceso. Un ejemplo sería:

```
while($a<$b)
  {
  print "$a\n";
  print "$b\n";
  $a=$b/2;
  $b=$b-1;
   };
```

4.8. El bucle `until`

Como en el caso del `if`, `while` tiene una forma simplificada cuando la condición está negada. Es `until`, y tiene como sintaxis:

```
until(condición)
  {
   instrucción1;
   instrucción2;
   ...
   instrucción;
  };
```

Por ejemplo, si tenemos algo como:

```
while(!correcto())
 {
  hazcosa();
  hazotracosa();
 };
```

podemos reescribirlo como:

```
until(correcto())
 {
  hazcosa();
  hazotracosa();
 };
```

Podemos también compactar `while` con una sentencia, de forma que se ejecuta iterativamente una única sentencia mientras que se cumpla la condición. La sintaxis es:

```
sentencia while condición;
```

Ojo, que la condición no necesita estar entre paréntesis. Por ejemplo, podemos hacer:

```
lee_dato() while hay_datos();
```

lo que es equivalente a:

```
while (hay_datos())
{
lee_dato();
};
```

también tenemos la compactación de la cláusula `until`. La sintaxis es:

```
sentencia until condición;
```

otra vez no necesitamos los paréntesis con la condición.

Por ejemplo, podemos hacer:

```
lee_dato() until fin_fichero();
```

Lo que es lo mismo que hacer:

```
lee_dato() while (!fin_fichero());
```

O, lo que es lo mismo, hacer:

```
until (fin_fichero())
{
lee_dato();
};
```

Tenemos también el bucle `do...while`. Su sintaxis es:

```
do
  {
   instrucción1;
   instrucción2;
   ...
   instrucción;
  }
  while (condición);
```

Esta sentencia ejecuta las instrucciones entre llaves mientras que se cumpla la condición. A diferencia del bucle `while`, aquí se comprueba la condición después de ejecutar las sentencias, en lugar de comprobar la condición antes, como hace el bucle `while`.

Por ello, las instrucciones del bucle `while` pueden no ejecutarse si antes de entrar en el `while` no se cumple la condición, mientras que el `do...while` siempre se ejecuta al menos una vez.

Por ejemplo, podemos hacer:

```
do
{
$i++;
}while($i<$j);
```

4.9. El bucle `do..until`

Así como tenemos el par `while` - `until`, también tenemos el par `do...while` - `do...until`. La sintaxis del bucle `do..until` es:

```
do
  {
   instrucción1;
   instrucción2;
   ...
   instrucción;
  }
  until (condición);
```

El `do...until` ejecuta las instrucciones entre llaves mientras que no se cumpla la condición. A diferencia del bucle `until`, en el bucle `do...until` se comprueba la condición después de ejecutar el bloque de sentencias, en lugar de comprobar la condición antes, como hace el bucle `until`. Esto significa que las instrucciones del bucle `do...until` siempre se ejecutan al menos una vez, mientras que el bloque de instrucciones del bucle `until` pueden no ejecutarse nunca.

Un ejemplo de bucle `do...until` es:

```
do
{
   $j--;
   $i++;
}until($i == $j);
```

lo que es lo mismo que:

```
do
{
    $j--;
    $i++;
}while(!($i==$j));
```

4.10. Los bucles

Los bucles son un caso particular de sentencia iterativa, donde tenemos una variable –denominada contador– que incrementamos o decrementamos; y cuando llegamos a un valor particular salimos del bucle. Los bucles en Perl también son similares a los de C, solo que siempre debemos poner las sentencias del bloque entre llaves. La sintaxis de un bucle en Perl es:

```
for(inicialización; condición; incremento)
  {
   instrucción1;
   instrucción2;
   ...
   instrucción;
  };
```

Como en el caso de los bucles en C, el comportamiento es un pelín esotérico para los no iniciados y tiene diferencias de lo que es un `bucle intuitivo` en otro lenguaje[1]; por lo que vamos a ver paso a paso lo que se hace realmente con un bucle `for`. Los tres elementos del bucle:

- `inicialización`: corresponde a un conjunto de instrucciones o expresiones, separadas por comas. Se evalúan una única vez, antes de comenzar la ejecución del bucle. El ámbito léxico de cualquier variable declarada aquí es apenas el del bucle, lo que hace este campo perfecto para declarar como variables temporales los iteradores e inicializarlos.

- `condición`: corresponde a un conjunto de instrucciones o expresiones, separadas por comas, que se evalúan para decidir si se sale del bucle o no. A pesar de que se evalúan todas las condiciones, el valor que cuenta para comprobar si se itera de nuevo el bucle o no es la última condición. Esto es fundamental entenderlo, ya que en `condición` pueden haber funciones que generen efectos laterales.

- `incremento`: corresponde a un conjunto de instrucciones o expresiones, separadas por comas, que se ejecutan siempre antes de la condición.

[1] Aunque si conocemos C, nos basta con saber que los bucles en Perl operan como los de C.

Esto significa que se ejecuta primero la inicialización, y después se ejecutan, por orden:

1. Ejecuta incremento.
2. Ejecuta la condición.
3. Si la última de las condiciones especificadas no es cierta, sale del bucle.
4. Ejecuta las instrucciones del bucle.
5. Repite de nuevo.

Un bucle simple, intuitivo, es:

```
for(my $i=0; $i<4;$i++)
{
   print $i;
};
```

Un bucle más complicado es:

```
for(my $i=0,my $j=0;$i<=4,$j<3;$i=$i+4,$j=$j+1)
{
  print "i:($i),j:($j)\n";
};
```

Da un aviso, porque el resultado de `$i<=4` no se utiliza. En este caso si lo quitamos no se modifica la lógica del programa; pero si tenemos una función con efectos laterales debemos tener cuidado, porque se evalúan todos los condicionales.

4.11. El bucle `foreach`

Por último, hablaremos de una estructura propia de Perl que extremadamente útil: `foreach`. Su sintaxis es:

```
foreach variable (lista)
{
   instrucción1;
```

```
    instrucción2;
    ...
    instrucciónN;
};
```

`foreach` itera la lista, ejecutando el bloque de instrucciones con cada elemento de la lista asignado a la variable. Por ejemplo, si tenemos la lista:

```
my @lista=('uno','dos','tres','cuatro');
```

Podemos hacer:

```
foreach my $elemento (@lista)
{
print "elemento: $elemento \n";
};
```

Y veremos el resultado, comprendiendo al momento el funcionamiento de `foreach`. Si declaramos el iterador en la cabecera del `foreach`, su ámbito léxico es el bucle `foreach`.

4.12. Perl y Goto

Perl tiene `goto`. Pero no sólo el `goto` tradicional de salto a una etiqueta, sino también el `goto` que tiene como parámetro una expresión que, al ser evaluada, genera la cadena de caracteres que será el nombre de la etiqueta a la que se salta. Como los `gotos` del Fortran, para los que hayan trabajado en este lenguaje. También tiene un tercer `goto` realmente esotérico , el `goto &`, del que no he encontrado aún ningún uso real.

Dicho en pocas palabras: si `goto` no, en Perl menos. Un programa en Perl con `gotos` puede ser realmente complicado de entender. Hasta para expertos en Perl.

4.13. La sentencia `next`

Dentro de un bucle, `next` pasa a la siguiente iteración del bucle. Es similar al `continue` de C y otros lenguajes derivados. Por ejemplo, podemos hacer:

```
while(a<b)
 {
     a--;
    if(7==a) {next;};
    c=c+a;
}
```

`next` comprueba otra vez la condición; si se cumple, vuelve a iterar el bucle `while`; si no se cumple, sale del bucle. Esto hace que `next` sea útil para establecer excepciones dentro del bucle, o para pasar a la iteración siguiente cuando sabemos que hemos operado todo lo que debíamos en la iteración en curso.

`next` puede llevar una etiqueta de bucle, saltando a la próxima iteración del bucle especificado. Por ejemplo, podemos hacer:

```
my $i;
my $j=0;

BUCLE: while($j<4)
{
     print "------\n";
     print "------\n";
     print "j=$j\n";
    $i=8;
    while($i>2)
    {
        print "-----\n";
        print "a:$i\n";
        $i--;
        $j++;
        print "b:$i\n";
        if($i==5){next BUCLE;};
        print "c:$i\n";
    };
};
```

`next` ejecuta la siguiente iteración de `BUCLE`, evaluando antes la condición de salida. Por ello, en este ejemplo nunca llega a ejecutarse entero el bucle más

interno, y el programa termina[2].

4.14. La sentencia `redo`

Para algunas cosas puede ser una buena idea no comprobar de nuevo la condición, sólo repetir la iteración en curso. Por ejemplo, si la condición tiene como efecto lateral realizar alguna operación -como extraer un dato del disco-, o si realmente queremos repetir la ejecución del cuerpo del bucle con el dato actual de disco, sin extraer otro. Otro escenario de uso es si queremos "engañar" al cuerpo de bucle a la entrada del bucle. Para todos estos escenarios nos será útil `redo`.

Un ejemplo de uso del bucle `redo` es:

```
my $i=7;
my $j=1;
while($i>3)
{
    print "-----\n";
    print "a:$i\n";
    $i--;
    print "b:$i\n";
    if($i==5){$i=$j;redo;};
    print "c:$i\n";
};
```

Este bucle también nos permite estudiar la diferencia entre `next` y `redo`. Con `my $j=1;` vemos que `redo` realiza una iteración más que `next`; una iteración que está fuera de la condición del bucle. Lo comprobamos ejecutando este código y ejecutándolo cambiando el `next` por `redo`.

Por otro lado, si cambiamos `my $j=1;` por `my $j=5;` la ejecución de `next` y de `redo` será la misma.

`redo` también puede llevar una etiqueta de bucle, repitiendo de nuevo la iteración en curso del bucle especificado. Por ejemplo, podemos hacer:

```
my $i;
```

[2]Más adelante, al estudiar `redo`, veremos que sustituyendo `next` por `redo`, el programa no termina.

```
my $j=0;

BUCLE: while($j<2)
{
    print "------\n";
    print "------\n";
    print "j=$j\n";
    $i=8;
    while($i>2)
    {
        print "-----\n";
        print "a:$i\n";
        $i--;
        $j++;
        print "b:$i\n";
        if($i==5){redo BUCLE;};
        print "c:$i\n";
    };
};
```

en este ejemplo `redo` ejecuta otra vez desde el principio la iteración en curso de `BUCLE`, pero sin evaluar la condición de salida, con lo que repetirá el bucle principal hasta que abortemos el script. Pero observamos que crece `$j`, y que supera el límite máximo del bucle; esto es así porque no estamos evaluando la comprobación de salida del bucle principal, lo que es la gran diferencia respecto a `next`.

4.15. La sentencia `last`

Por último, la sentencia `last` nos permite salir del bucle al instante sin realizar ninguna iteración más. Por ejemplo, si hacemos:

```
my $i=8;
while($i>2)
{
    print "-----\n";
```

```
    print "a:$i\n";
    $i--;
    print "b:$i\n";
    if($i==5){last;};
    print "c:$i\n";
};
```

haremos solo dos iteraciones y media.

`last` también puede llevar una etiqueta de bucle, saltando al final del bucle especificado. Por ejemplo, podemos hacer:

```
my $i;
my $j=0;

BUCLE: while($j<100)
{
    print "------\n";
    print "------\n";
    print "j=$j\n";
    $i=8;
    while($i>2)
    {
        print "-----\n";
        print "a:$i\n";
        $i--;
        print "b:$i\n";
        if($i==5){last BUCLE;};
        print "c:$i\n";
    };
};
```

aquí `last` sale al final del bucle `BUCLE`, en lugar de al final de su bucle local.

4.16. `next`, `redo` y `last` en sentencias selectivas

En principio, `next`, `redo` y `last` se suelen utilizar sólo con bucles. Sin embargo, se pueden utilizar dentro de cualquier entorno. Por ejemplo, podemos

hacer:

```
{
    # Operaciones
    # ...
    #
    $n--;
     if($n<3)
                    {
                            print "Menor que tres\n";
                            last;
                    };
    # Operaciones
    # ...
    #
    $n*=17;
    print "Mayor que tres.";
};
```

Saltando si `n` es menor que 3 a donde cierran las llaves del entorno, es decir, después el `print`. Esto nos da un mecanismo para "saltar hacia adelante" cuando cumplimos determinada condición; eliminando el único uso posible de `goto` para optimizar código.

Si nos fijamos en el ejemplo, el fin de entorno no es el cierre de llaves de la selectiva, sino el cierre de llaves del último entorno abierto. Esto no significa que no podamos utilizar esto dentro de una selectiva; sino que si queremos usarlo dentro de una selectiva, debemos crear un entorno adicional. Por ejemplo:

```
if($n<3)
 {{
        if ($n==1){
                        $n=+2;
                last;
                        };
        $n+=2;
        $n*=3;
  };};
```

Observamos las dobles llaves en el bucle: son lo que nos permitirán que al usar `last` salte al final de la selectiva, y no al final del entorno que contiene el `if`. También podemos usar anidaciones múltiples, entornos con nombres y cualquiera de los elementos ya comentados, por lo que tenemos mucha flexibilidad de salto si queremos optimizar código.

4.17. Actividades

La actividad de este tema será un programa que calcule el factorial de un número varias veces, e imprima el resultado.

El programa debe hacer lo siguiente:

- Almacena tu DNI en una variable tipo hash.
- Almacena los dígitos de tu DNI en un vector.
- Para cada una de las sentencias de control iterativo estudiadas, salvo `goto`:
  - Suma los dígitos almacenados en el vector, recorriéndolo con dicha sentencia de control iterativo.
  - Imprime el resultado.
  - Calcula el factorial del número resultante en el paso anterior
  - Imprime el resultado.
  - Calcula el factorial del número resultante en el paso anterior, eliminando de la cuenta los múltiplos de 2 y de 3.
  - Imprime el resultado.

El programa, por lo tanto, imprimirá tres resultados por cada una de las sentencias de control iterativo estudiadas en este tema.

Cuando necesites control no iterativo, puedes hacerlo mediante cualquiera de las sentencias de control no iterativo estudiadas; pero tienes que terminar usando todas en el programa. Como tendrás que utilizar varias en lugares distintos, no tendrás problemas en utilizarlas todas.

La actividad será calificada como APTA si el programa funciona, imprimiendo el resultado correcto y utilizando todas las sentencias de control iterativos y no iterativo estudiadas, salvo `goto`.

Capítulo 5

Subrutinas y paso de parámetros en Perl

En un lenguaje como Perl donde tenemos toda la flexibilidad que queramos, hacer un programa modular es fundamental. En este artículo vamos a aprender los mecanismos que el lenguaje nos da para hacer nuestros programas modulares.

La función es el mecanismo básico de la programación estructurada. En Perl, como en C, tenemos funciones; y no existe una distinción entre el concepto de "función" y "procedimiento". Los parámetros se pasan a una función en Perl por valor, aunque también podemos pasar referencias como parámetros.

El uso de funciones en Perl es extremadamente potente y flexible, mucho más que en cualquier otro lenguaje. Podemos definir una función prácticamente en cualquier lugar del código. Podemos declarar funciones dinámicamente con `eval`; lo que permitirá que los programas en Perl puedan modificarse a sí mismos. Podemos, mediante el mecanismo de funciones anónimas, declarar funciones sin nombre, que referenciamos a través de una variable. Podemos programar funciones de orden superior, o incluso hacer programas basados en el paradigma de programación funcional. En este curso no vamos a cubrir todo esto por razones de tiempo, solo comentando que existen, y que son posibles. Sí vamos a ver el mecanismo de declaración y uso de funciones más antiguo y empleado, el de la programación modular; y más adelante estudiaremos el orientado a objetos.

El mecanismo de funciones de Perl que está más relacionado con lo que pue-

de ser una programación estructurada más convencional es sencillo de usar; pero si no lo usamos sabiamente, podemos hacer programas realmente poco legibles y que tienen errores muy difíciles de localizar.

Por otro lado, en algunos casos específicos –por ejemplo, que según el valor de una variable o de una entrada se puedan hacer gran cantidad de operaciones completamente distintas– el mecanismo de funciones dinámicas nos permite realizar un código limpio y elegante; pero, como ya hemos dicho, nos centraremos en el uso y definición convencional de funciones.

5.1. Declarando funciones en Perl

A diferencia de otros lenguajes, Perl no necesita una definición previa de funciones. Sin embargo, sí es una buena idea realizarla para eliminar errores muy molestos, ya que damos más información al interprete y le permitimos que nos localice más errores de programación.

El primer mecanismo que disponemos en Perl para declarar una función es declarar sólo su nombre, lo que hacemos con:

```
sub nombredefunción;
```

donde `nombredefunción` es el nombre de la función. Aquí lo que hacemos es exclusivamente decirle a Perl: "ojo, existe una función que declararemos y usaremos en el programa, y que se llama `nombredefunción`". Con este mecanismo podemos hacer que los errores de colisiones en el espacio de nombres se generen al principio del código, en lugar de después de mucho tiempo de ejecución; sobre todo, en rutinas generadas dinámicamente. Por ello, este mecanismo se suele utilizar inmediatamente después de la importación de paquetes en Perl, o para definir todos los nombres de todas las rutinas que esperamos desarrollar con objeto de verificar que nuestro convenio de nombres no nos genera una colisión en el espacio de nombres. No debemos preocuparnos por dicha colisión en exceso ahora, ya que en Perl hay un mecanismo muy simple para evitar colisión de espacio de nombre entre un paquete y una función local, que podemos emplear una vez que la hemos detectado; pero sí tenemos que identificar la colisión, y repararla. Hablaremos de este mecanismo más adelante, cuando entremos en los paquetes y la programación orientada a objetos.

5.2. Declarando prototipos en Perl

El segundo mecanismo que tenemos en Perl para declarar funciones es la declaración de prototipos. En este mecanismo damos un poco más de información al intérprete; ya que le estamos informando de cuantos parámetros pasamos, y de qué tipo, además del nombre de función que indicábamos mediante el mecanismo anterior. El concepto es similar a las cabeceras de C; indicamos el nombre de la función y qué parámetros pasamos. La sintaxis es:

```
sub nombredefunción (parámetros);
```

Donde `parámetros` son tanto el tipo de los parámetros que tiene la función, como su número; representando cada tipo por el carácter que antecede una variable de dicho tipo. Por ejemplo, una declaración válida de prototipo de función sería:

```
sub mifunc ($$@);
```

Donde estamos declarando una función `mifunc` cuyo primer parámetro es un escalar, el segundo parámetro es un escalar y el tercero es una matriz. Por ello, la rutina puede ser llamada con:

```
mifunc($a,$b,@c);
```

Definiendo los prototipos de las funciones nos aseguramos que los errores se generarán al cargar el fichero en Perl, en lugar de que al ejecutarlo, hagamos la llamada errónea, en el momento que dicha llamada acontezca. Esto nos ahorrará mucho tiempo de depuración de código[1].

5.3. Declarando las funciones

Declarar o no la cabecera de una función, para poder ejecutar una función debemos forzosamente definir su cuerpo, es decir, decir que hace la función.

Hay muchas formas de declarar una función en Perl, tanto de forma estática como de forma dinámica. Algunas se deberían estudiar en asignaturas de

[1] Ojo con esto, Manuel: cuando digo que te ahorrará mucho tiempo de depuración, me refiero a **horas** por cada metedura de pata que podrías haber corregido en segundos. Y por la naturaleza de Perl, tendrás muchas meteduras de pata de este estilo en el desarrollo de cualquier programa en Perl. Por ello, salvo que quieras presentar código a un torneo de código ofuscado, siempre debes usar como mínimo este nivel de declaración de funciones.

programación declarativa o funcional. Pero nos centraremos en las dos formas de definir una función acordes con el paradigma imperativo[2] y la programación modular: indicando su prototipo, o no indicándolo. La sintaxis de la definición del cuerpo de una función sin prototipo es:

```
sub nombredefunción
{
    cuerpo
};
```

Por ejemplo[3], podemos definir la función:

```
sub incrementa
{
     (my $temp)=@_;
    $temp++;
    return $temp;
};
```

La definición de una función con prototipo es:

```
sub nombredefunción(prototipo)
{
    cuerpo
};
```

Por ejemplo, podemos definir la función anterior con su prototipo con:

```
sub incrementa($)
{
     (my $temp)=@_;
    $temp++;
    return $temp;
};
```

[2]Si no sabes qué es "paradigma imperativo" interprétalo como "programacíón de toda la vida".

[3]En este momento voy a utilizar dos cosas que no te he explicado; la primera línea y la tercera línea de la función. Pero por favor, Manuel, dame algo de cancha. La tercera línea puedes deducir lo que hace; y aunque la primera línea sea aún un criptograma indescifrable, antes de que termine el tema sabrás lo que significa. Pero por algún lado tengo que comenzar a explicártelo, por lo que de momento te crees que tiene sentido, y ya te explicaré cual es.

Si queremos ignorar el prototipo de una función al llamarla[4], lo podemos hacer poniendo `&` antes de la llamada a la función. Por ejemplo, podemos llamar la función del ejemplo anterior con:

```
&incrementa(3,4);
```

Y no dará error en la llamada. De hecho, en este caso particular va a funcionar correctamente la rutina, ya que tiene todos los parámetros que necesita para funcionar. Si sobran parámetros, lo puede que no ocurra nada negativo. Si faltan parámetros, lo que pase dependerá fuertemente de la implementación de la rutina. La forma que vamos a estudiar en este curso para utilizar en una rutina los parámetros de llamada que le han mandado es la más robusta posible en Perl; en caso de que llamemos a la función con menos parámetros de los debidos, por no haber declarado la función o por haberla llamado con `&`, lo que ocurrirá es que no se inicializará la variable que va a contener el valor no definido; de hecho, no se definirán los parámetros del final al principio de la lista de parámetros. Dicho de otra forma, si definimos cuatro parámetros en el prototipo y lo llamamos solo con dos parámetros, los dos últimos parámetros no se definirán. Según usemos `use strict` o no, el efecto será distinto. Pero ya volveremos a esto más adelante.

5.4. Acceso a los parámetros de una función

Ahora es cuando entramos en el paso de parámetros, y en una línea que he empleado varias veces y aún no he explicado.

Perl se caracteriza porque tiene muchas variables "especiales". Estas son variables que valen cosas; y que el valor depende del momento en el que el programa se ejecuta. Tenemos variables especiales que contendrán los argumentos con los que se llamó al script en Perl; variables especiales que contendrán los resultados de la última búsqueda en una expresión regular, variables especia-

[4]Esto es importante, Manuel, porque te permite entender la filosofía detrás de Perl. Perl no es un lenguaje megaestructurado que supone que eres imbécil, y que el lenguaje te debe protegerte de ti mismo. Perl te da toda la potencia y la flexibilidad. Puedes incluir `use strict`, declarar las funciones y las variables. O no declararlas. O incluso tienes una forma de forzar al intérprete a ignorar que una función está declarada, y que la estás llamando de forma errónea. Perl te da herramientas: si utilizas el martillo para clavar clavos, o para golpearte repetidas veces la cabeza, es algo en lo que el lenguaje no entra. Sé, especialmente si has trabajado demasiado con Java, que no estás acostumbrado a esta mentalidad; pero en determinados entornos es más importante sacar adelante el trabajo de forma limpia, rápida y eficiente, que la pureza formal.

les que contendrán el último error de Perl, el último error de llamada, el último resultado de `eval`, o el PID del script de Perl en ejecución.Este mecanismo es potentísimo; ya que ahorra un montón de llamadas a funciones para obtener cosas, y de variables temporales de un solo uso. Sin embargo, tiene el handicap de que los nombres de estas variables especiales no siempre son legibles. Por ello, en Perl podemos hacer uso al principio de nuestro programa de:

```
use English;
```

Básicamente este módulo lo que permite es llamar a las variables especiales con nombres menos crípticos. Personalmente considero que cuando ya llevas algunos programas en Perl –y no muchos, precisamente– dejan de ser crípticos estos nombres; pero ahí está `use English`, por si quieres utilizarlo en tu código. Sí tienes que tener cuidado con `use English` en expresiones regulares muy complicadas, ya que castiga el rendimiento si la expresión regular no es de las "sencillas".

Explicado esto, vamos a ver la primera variable especial de las que hablaremos en nuestro curso.

Cuando llamamos a una función, a la función le llegan los parámetros en una variable especial de tipo lista que se llama `@_`. Si utilizamos `use English`, también podemos referirnos a esta lista como `@ARG`[5].

Hay muchas formas de extraer parámetros de la lista de parámetros que recibes en una subrutina. Hay tres que son comunes: extraerlos de uno en uno con `shift` o `pop`, emplearlos directamente, o asignación directa de parámetros. Personalmente encuentro el primer método poco robusto e inoperativo; aunque lo he visto en programas de otros, yo no lo empleo. Los métodos que explicaré serán emplear el parámetro directamente, o su asignación directa.

5.5. Uso directo de los parámetros de llamada

Emplear directamente el parámetro es algo que se hace sobre todo en funciones pequeñas, y que tienen pocos parámetros. En funciones grandes, o que tienen muchos parámetros, este sistema supone un importante castigo a la legibilidad. Si nuestra función tiene un único parámetro de tipo lista, es tan sencillo como usar directamente `@_`. Por ejemplo, calculamos el módulo de un vector en

[5] Y puedes tú mismo juzgar cuanto hemos "ganado" por haber empleado el `use English`.

Perl con la función:

```
sub modulo
{
        my $ac=0;
        foreach $i (@_)
                {
                $ac+=$i**2;
                };
        return sqrt($ac);
}
```

Ahora es un buen momento para explicar otra variable especial: `$_`. Esta variable podemos interpretarla como "lo que me traigo entre manos", y significa cosas distintas en lugares distintos, pero siempre con ese sentido. En el caso especial del `foreach`, el código anterior podemos sustituirlo por:

```
sub modulo
{
        my $ac=0;
        foreach (@_)
                {
                $ac+=$_**2;
                };
        return sqrt($ac);
}
```

Esta función hace exactamente lo mismo que la anterior. Si nos damos cuenta, realmente `$_` nos quitará decenas de variables temporales de un solo uso en la mayor parte de los programas. Tendremos más apariciones de esta variable temporal `$_` más adelante cuando lleguemos a las expresiones regulares.

He explicado en este punto esta variable especial para no confundirla con otra. Y ojo, Manuel, porque aquí viene una de las cosas de Perl que se le atragantan a la gente. Debemos recordar que en Perl existen tres espacios distintos de nombres de variables, cada uno por tipo. Esto significa que podemos tener dos variables con distinto tipo, que se llamen de la misma forma; por ejemplo:

```
my $a=3;
```

```
my @a=(4,5);

print "$a␣@a\n";
```

imprimirá 3,4,5 en pantalla.

Todo esto significa que Perl interpretará en base al uso cual de las dos variables nos referimos. Pero tenemos que interpretar correctamente lo que Perl entiende. Podemos preguntarnos, para entender esto, ¿Como accedemos al primer elemento de `a`? Haciendo `a[0]`[6]. Y, ¿Como decimos que lo interpretaremos como escalar? Con `$a[0]`. Es decir, que `$a[0]` no significa que tomamos la variable `$a` y le extraemos el primer elemento añadiéndole el `[0]`; sino significa que tomamos la variable `a`, identificamos que es de tipo matriz porque le extraemos un elemento con `[0]`, quedando `a[0]`; y que, como lo que sale lo queremos interpretar como escalar, lo precedemos por `$`; escribiendo el conjunto con `$a[0]`.

Sé que esto es lioso, Manuel. Léelo de nuevo, mira el ejemplo que comento ahora, prueba en un ordenador lo que digo, y, si no lo entiendes aún, pregunta lo que necesites preguntar en el foro.

Por ello, el programa anterior hace lo mismo que:

```
my $a=3;

my @a=(4,5);

print "$a␣$a[0]␣$a[1]\n";
```

que también imprimirá 3,4,5 en pantalla.

Y ahora viene la clave: si tenemos varios parámetros de llamadas a una función, y no son una única lista, ¿Podemos utilizar estos parámetros por separado? Sí, accediendo al vector `@_` con semántica de matriz. Su primer elemento sería `$_[0]`, su segundo elemento sería `$_[1]`, su tercer elemento sería `$_[2]`, y así para todos los parámetros. Por ejemplo, una función que calcule el módulo de

[6] Lo que no funciona en Perl, porque nos falta por incluir $, % o según qué tipo queramos que el intérprete suponga que tiene el contenido de `a[0]`.

vectores de exactamente tres escalares sería:

```
sub modulo3
{
 return sqrt($_[0]**2+$_[1]**2+$_[2]**2);
}
```

Es importante que repitamos esto, que es muy importante: `$_` es una variable que no está relacionada con `@_`; ya que están en espacios de nombres distintos; `$_` en el de escalares, y `@_` en el de vectores. Y cuando accedemos a `$_[1]`, estamos accediendo al segundo elemento de la variable de tipo vector _, y este elemento extraído lo estamos interpretando como un escalar.

5.6. Asignación directa de los parámetros de llamada

Asignar de forma directa los parámetros de llamada es mi forma preferida de asignar los parámetros en Perl[7]. Por ello, lo primero que debemos hacer es obtener los parámetros de la lista. Por ejemplo, podemos hacer:

```
sub seisparametros($$$$$$)
{
    (my $parametro1,
     my $parametro2,
     my $parametro3,
     my $parametro4,
     my $parametro5,
     my $parametro6)=@_;
};
```

Recordamos del apartado anterior que `@_` corresponde a la lista de los parámetros pasados a la función concatenados. Por ejemplo, si llamamos a la función:

```
sub prueba
{
    print @_;
};
```

[7] También es la forma preferida de Perl Critic, http://www.perlcritic.org/, un evaluador de código en Perl.

Con:

```
prueba;
```

No imprimirá nada, ya que `@_` contiene la lista vacía.
Si la llamamos con:

```
prueba "Hola";
```

Imprimirá `Hola`, ya que `@_` contiene un único elemento, la cadena `Hola`.
Si la llamamos con:

```
prueba 2,"hola";
```

Imprimirá `2hola`, ya que `@_` contiene una lista con dos elementos, uno con el entero `2` y otro con la cadena `hola`.
Pero ojo; si hacemos:

```
my @lista=('uno','dos','tres');

prueba 2,"hola",@lista;
```

Los programadores en C supondrán que imprimiremos `2hola` y después una referencia, pero esto no es así. En Perl el paso de parámetros es muy distinto internamente a C; ya que realmente `@_` contendrá una lista con cinco elementos: 2, `hola`, `uno`, `dos` y `tres`.
Llegados a este punto, ya sabemos pasar escalares a una función o subrutina en Perl. Es una buena idea que pares ahora la lectura, Manuel; y te sientes a probar los ejemplos en un ordenador.
Esto nos lleva al segundo punto que debemos destacar respecto al paso de parámetros en Perl: el paso de listas como parámetros.

5.7. El paso de listas a funciones

Con lo que has aprendido hasta ahora, puedes pasar a una rutina una única lista, pero no dos listas. Por ejemplo, si hacemos:

```
sub doslistas(@@)
{
```

```
    (my @a,my @b)=@_;

    print "Primera lista: @a";
    print "\n";
    print "Segunda lista: @b";
    print "\n";
};

my @m1=('a','b','c','d');
my @m2=('1','2','3','4');

doslistas(@m1,@m2);
```

El resultado será:

```
Primera lista: a b c d 1 2 3 4
Segunda lista:
```

Es decir, realmente tenemos toda la información en la primera cadena, cuando deberíamos haber recibido cada una de las cadenas que se pasaron en una cadena distinta.

Si te detienes a pensarlo, hasta tiene su lógica, podemos pensar: la función recibe un único parámetro, que es una lista; por lo que si mandamos dos listas, realmente la función recibe una, que es la concatenación de ambas. De hecho, no es que hayamos concatenado las listas en recepción; sino que la coma es el operador de concatenado. Y que realmente en Perl todas las funciones reciben un único parámetro, y devuelven un único parámetro. Por poner otro ejemplo, a la función:

```
sub modulo3
{
 return sqrt($_[0]**2+$_[1]**2+$_[2]**2);
}
```

Da lo mismo llamarla con:

```
modulo3(3,9,2)
```

que con:

```
modulo3((3,9),2)
```

o que con:

```
modulo3(3,(9,2))
```

En el primer caso, estamos concatenando tres escalares de golpe y pasando la lista resultante a la función `modulo3`. En el segundo, estamos concatenando dos escalares, contatenado después un tercer escalar, y pasando la lista resultante a la función `modulo3`, y el tercero hace operaciones análogas en orden distinto.

¿Sorprendente? Pues ahora sí entiendes el paso de parámetros en Perl.

Ahora puede que estés comenzando a preguntarte: ¿Y como paso una estructura de datos compleja?

Para poder resolver esto, te falta un detalle: la referencia que apunta a una variable se obtiene con el operador `\`; por lo que si queremos obtener una referencia a la variable `@m1`, basta con emplear `\@m1`. `\@m1` es un valor escalar; y si concatenamos `\@m1` con `\@m2`, no tendremos una lista que contenga los valores de `@m1` y los de `@m2`; sino que tendremos una lista que contiene dos listas: una con la lista `@m1`, y otra con la lista `@m2`.

Resumiendo, si hacemos:

```
(@m1,@m2)
```

el resultado será una lista que contiene la concatenación de los valores de `@m1` y los de `@m2`; pero si hacemos:

```
(\@m1,\@m2)
```

El resultado será una lista con solamente dos elementos; el primero será la lista `@m1`, y el segundo la lista `@m2`. Tendremos, por lo tanto, una lista de listas.

Sabiendo esto, la solución pasa a ser tan sencilla como, en lugar de pasar dos listas, pasar una lista con dos referencias. La rutina anterior funciona perfectamente una vez que está escrita como:

```
sub doslistas($$)
{
      (my $a,my $b)=@_;
    print "Primera␣lista:␣@$a";
    print "\n";
    print "Segunda␣lista:␣@$b";
    print "\n";
};

my @m1=('a','b','c','d');
my @m2=('1','2','3','4');

doslistas(\@m1,\@m2);
```

Es muy importante que nos fijemos un poco más en cómo estamos declarando la función. Ahora no pasamos dos parámetros vectoriales, sino dos escalares; ya que hemos pasado referencias a listas, que al fin y al cabo serán dos listas. En el momento en el que pasamos dos escalares, tenemos que recibirlos como escalares; por eso hacemos:

```
      (my $a,my $b)=@_;
```

Y ahora viene la magia de Perl. ¿Qué tenemos en `$a`? Una referencia a una lista. Hay que forzar a que lo interprete como lista. ¿Y cómo lo hacemos?

```
    print "Primera␣lista:␣@$a";
```

Exactamente. Como `$a` contiene un escalar que realmente es una referencia a una lista, con `@$a` decimos que "vea" la lista que hay detrás de ese escalar. Cuando veamos un "chorizo" de arrobas, porcentajes, y símbolos de dólar antes de un identificador, realmente lo que estamos haciendo es recorrer la estructura referenciada por ese identificador, hasta llegar al dato que buscamos. Es como cuando en C tenemos varios asteriscos a la izquierda del identificador: cada asterisco correspondía con un "es un puntero; mira a donde apunta". Y podíamos encadenar varios. En Perl, no solamente recorremos estructuras simples, lineales, como hacemos con las secuencias de asteriscos: podemos recorrer estructuras realmente complicadas, los signos a la izquierda nos dicen como debemos interpretar en cada momento a donde estamos apuntando, y los de la derecha,

donde tenemos que saltar dentro de la estructura. Si además sabemos que con las llaves podemos agrupar el símbolo de la izquierda –que indica el tipo de lo que leemos– y el de la derecha –donde saltamos– para facilitar legibilidad, algo como:

```
@{@{$subtree}[4]}[2]
```

Deja de ser un ataque de ácido en mal estado y cobra sentido si lo leemos de dentro hacia afuera: `$subtree` será un escalar, referencia a un vector; cuyo cuarto elemento será a su vez una referencia a otro vector, del que recuperaremos la segunda posición.

Este ha sido un salto conceptual muy complicado, y te estás enfrentando en este tema a las cosas más extrañas que estudiaremos de Perl en todo el curso. Es una buena idea que pares ahora la lectura, Manuel; y te sientes a probar los ejemplos que hemos comentado hasta ahora en un ordenador, y que hagas tus pruebas.

Finalmente, pasar varias matrices asociativa en Perl tiene exactamente el mismo problema, que solucionamos exactamente de la misma forma:

```
sub doshashs($$)
{
      (my $a,my $b)=@_;

 # Aquí las operaciones con hashes


};

my %m1=('a','1','b','2','c','3','d','4');
my %m2=('e','5','f','112');

doshashs(\%m1,\%m2);
```

Es decir, pasamos dos parámetros en cualquiera de las dos rutinas que hemos estudiado: serán dos escalares, que corresponden a referencias a los parámetros "reales".

5.8. Devolviendo valores desde funciones

Lo último que debemos saber del uso de funciones es como devolver un valor al llamante. Esto lo hacemos con:

```
return valor;
```

Si llegamos al final de una función sin haber devuelto un valor con `return`, la función devuelve el último valor evaluado. Por ejemplo, podemos escribir la rutina `incrementa` como:

```
sub incrementa($)
{
      (my $temp)=@_;
    $temp++;
    return $temp;
};
```

Que es lo mismo que hacer:

```
sub incrementa($)
{
      (my $temp)=@_;
    $temp++;
};
```

o incluso que hacer:

```
sub incrementa($)
{
    $_[0]++;
};
```

Si queremos devolver más de un valor en una función, tampoco es problema. Podemos devolver listas; por ejemplo:

```
sub unoydos
{

        return (1,2);

}
```

Devuelve una lista que contiene los valores `1` y `2`. Si queremos devolver estructuras más complicadas, siempre podemos devolver referencias. O listas de referencias a estructuras de datos arbitrariamente complicadas.

5.9. Actividades

La actividad de este tema será un programa que tenga implementadas las siguientes funciones:

- Que reciba dos números, los sume y devuelva el resultado.
- Que reciba una lista de números, los sume si cumplen una condición, y devuelva un resultado. La condición para que no sean excluídos estos números de la suma es que sean distintos de los dígitos de tu DNI.
- Que reciba dos listas de números, calcule el producto vectorial de ambas listas, realice el producto vectorial del resultado con un vector formado por las cifras tu DNI, y devuelva el producto vectorial de ambas listas, y el resultado del producto vectorial de ambas listas por tu DNI –es decir, debe devolver dos listas–.
- Que reciba una lista y una matriz asociativa. Dicha matriz asociativa tendrá al menos un campo, cuya clave sea tu DNI, y que contenga una lista. Esta función debe llamar a la tercera de las funciones que te he pedido, pasando como parámetros la lista recibida y la lista referenciada por la matriz asociativa a través de tu DNI; recuperar las dos listas de resultados, y llamar sobre ambas a la segunda de las funciones que te he pedido. El resultado numérico lo debes pasar como primer parámetro a la primera función que te he pedido, y como segundo parámetro pasarás el número 18.

Si en algún momento te "faltan vectores" para alguna operación –lo que te pasará al calcular el producto vectorial–, los n vectores que te faltan a partir del i serán:

$$\begin{aligned}
\vec{v}_i &= (1, i, 1, i, 1, i, 1, i, 1, i, 1, i, 1, i, 1, i, \quad \cdots) \\
\vec{v}_{i+1} &= (1, (i+1), 1, (i+1), 1, (i+1), 1, (i+1), \quad \cdots) \\
\vec{v}_{i+2} &= (1, (i+2), 1, (i+2), 1, (i+2), 1, (i+2), \quad \cdots) \\
&\vdots \qquad\qquad\qquad \ddots \qquad\qquad\qquad \vdots \\
\vec{v}_{i+n} &= (1, (i+n), 1, (i+n), 1, (i+n), 1, (i+2), \quad \cdots)
\end{aligned}$$

Además de esto, el programa debe:

- Declarar una variable de tipo matriz asociativa, que tenga un campo, cuya clave sea tu DNI, y que contenga una lista con los dígitos de tu DNI al revés.
- Declarar una variable de tipo lista, que contenga los dígitos de tu DNI.
- Llamar con la matriz asociativa y las listas antes declaradas a la última de las funciones solicitadas.
- Finalmente, imprimir en pantalla en resultado.

Te habrás dado cuenta, Manuel, que todas las funciones se terminan llamado entre sí. Realmente lo importante es que practiques el "musculo" de la declaración de parámetros y el paso de parámetros.

Para que la tarea sea valorada como apta, el programa debe funcionar bien, hacer lo que te pido, y además **tienen que estar bien declaradas las funciones**. Esto incluye declarar el número y el tipo de los parámetros de cada una de ellas.

Te recomiendo antes de que te pongas en faena que pruebes los ejemplos. Este tema he comentado las rarezas más importantes de Perl, algunas no son triviales, y mientras que hasta ahora han sido muy fáciles las prácticas, esta es la práctica más compleja a la que te enfrentarás en el curso.

Después de haber probado los ejemplos de los apuntes, y de haber intentado el ejercicio, no dudes en utilizar el foro para preguntar todo lo que no entiendas.

5.10. Actividad alternativa

Te recomiendo que intentes la práctica anterior. Aprenderás más; y te es especialmente interesante aprender lo más posible del curso.

Sin embargo, mucha gente se está atrancando en esta práctica. Es cierto que esta actividad está pensada para que emplees la mitad del tiempo que emplearás en el curso; pero tampoco quiero que nadie se quede atrancado aquí.

Por ello, y si ves que la anterior se te atranca, te propongo una serie de prácticas que son la "actividad alternativa"; si me haces esta de forma correcta también te daré por buena la actividad, y podrás pasar al tema siguiente.

En esta actividad me tienes que mandar un programa que haga lo siguiente:

- Funciones que realicen las siguientes operaciones sobre vectores de longitud variable e indeterminada –una función por operación–:
  - Suma de vectores.
  - Producto de un escalar por un vector.
  - Producto de un vector sobre una matriz. Esta matriz también debe ser de tamaño variable e indeterminado.
  - Suma de componentes de un vector. Me debes implementar dos funciones que implementen versiones distintas del cálculo de la suma de componentes de un vector: una iterativa, y otra recursiva.
- Además, una función de suma de matrices, que opere sobre dos matrices de tamaño variable e indeterminado.

El programa debe llamar a todas estas funciones comentadas, e imprimir los valores resultantes. Como necesitarás parámetros para llamar a las funciones, emplearás los siguientes parámetros:

- Donde necesites un vector como parámetro, emplearás un vector cuyas componentes sean los dígitos de tu DNI.
- Donde necesites un escalar, emplearás la suma de los dígitos de tu DNI; que calcularás empleando la función recursiva de suma de componentes de un vector de la que ya hemos hablado.
- Donde necesites una matriz, usa matriz cuyas filas estén definidas de la forma:

$$\begin{aligned}
\vec{v}_i &= (1, i, 1, i, 1, i, 1, i, 1, i, 1, i, 1, i, 1, i, 1, i, \quad \cdots) \\
\vec{v}_{i+1} &= (1, (i+1), 1, (i+1), 1, (i+1), 1, (i+1), 1, (i+1), \quad \cdots) \\
\vec{v}_{i+2} &= (1, (i+2), 1, (i+2), 1, (i+2), 1, (i+2), 1, (i+2), \quad \cdots) \\
&\vdots \qquad\qquad\qquad \ddots \qquad\qquad\qquad \vdots \\
\vec{v}_{i+n} &= (1, (i+n), 1, (i+n), 1, (i+n), 1, (i+2), \quad \cdots)
\end{aligned}$$

Además de esto, el programa debe antes de llamar a las funciones:

- Declarar una variable de tipo matriz asociativa, que tenga un campo, cuya clave sea tu DNI, y que contenga una lista con los dígitos de tu DNI al revés.
- Declarar una variable de tipo lista, que contenga los dígitos de tu DNI.
- Llamar con la matriz asociativa y las listas antes declaradas a la última de las funciones solicitadas.
- Finalmente, imprimir en pantalla en resultado.

Capítulo 6

Expresiones regulares en Perl

Una de las cosas más potentes en Perl son sus expresiones regulares. También son una de las cosas que estéticamente menos se parecen a lo que un neófito puede estar acostumbrado a utilizar. En este tema vamos a analizar qué son, para qué sirven y cómo se usan las expresiones regulares en Perl.

6.1. Qué es una expresión regular

Para entender qué es una expresión regular, vamos a comenzar viendo las expresiones regulares que son más sencillas de encontrar y que todos hemos usado alguna vez.

Supongamos que queremos listar todos los archivos que terminan en `.c` en un directorio. Para conseguir esto, podemos hacer:

```
ls *.c
```

si, por otro lado, queremos listar todos los archivos que contengan un carácter cualquiera, la cadena `imagen`, una secuencia de caracteres, y la terminación `.png`, hacemos:

```
ls ?imagen*.png
```

Tanto `*.c` como `?imagen*.png` son expresiones regulares simples.

6.2. Qué es una expresión regular y para qué sirven

Una expresión regular es una cadena que describe un patrón de cadenas. Este patrón puede definir ninguna cadena razonable, una cadena, varias cadenas o incluso infinitas cadenas. Por ejemplo, en línea de comandos el patrón:

```
ab
```

Significa "`archivos que se llamen exactamente ab`"; mientras que el patrón:

```
a?b
```

Significa: "`archivos que comiencen por a, tengan un carácter después indeterminado y terminen por b`". Esto da unos centenares de posibles cadenas definidas por un único patrón. Por último, la expresión regular definida por:

```
a*
```

Significa "`cualquier archivo que comience por a`". Este patrón define infinitas cadenas, ya que podemos poner detrás de `a` lo que queramos[1].

Al igual que definimos cadenas con una expresión regular, también podemos usar las expresiones regulares para definir subcadenas dentro de una cadena mayor. Por ejemplo, todos estamos acostumbrados a algo como buscar la cadena "`es`" en "`esto es una prueba`"; lo que nos daría dos apariciones. Pero, por ejemplo, la subcadena "`e?`" dentro de la cadena "`esto es una prueba`" nos daría:

- `es` a partir de la posición 0
- `es` a partir de la posición 5
- `eb` a partir de la posición 15

Esto corresponde con la idea de *"la primera y sucesivas apariciones de la expresión regular* `e?` *en la cadena"*.

La subcadena "`pr*`" dentro de la cadena "`esto es una prueba`" daría:

[1] En la realidad física del ordenador tenemos límites del `a*`, por lo que no es realmente infinito; pero desde el punto de vista formal es infinito, y podemos entender la idea.

- `pr` a partir de la posición 12
- `pru` a partir de la posición 12
- `prue` a partir de la posición 12
- `prueb` a partir de la posición 12
- `prueba` a partir de la posición 12

Esto corresponde a la idea de cuan "glotona" sea la expresión regular. Estos dos conceptos son importantes, ya que significan que una misma expresión regular buscada en una subcadena puede dar lugar a muchísimos más emparejamientos válidos de los que podemos suponer; y será el principal escoyo que tendremos a la hora de trabajar con expresiones regulares: que emparejemos, pero no emparejemos con lo que buscamos, sino con una subcadena, o una supercadena de lo que buscamos; o que emparejemos, pero con una subcadena ubicada en un lugar distinto del que pensamos.

Ahora es un buen momento para que pares, Manuel; e intentes buscar las subcadenas `e*` dentro de la cadena "`esto es una prueba`". Con un papel y un lápiz. Te tienen que salir exactamente 34 subcadenas. Si te salen menos, es que hay algún concepto que no ves. Es muy importante que te pelees con esto hasta verlo, y que si no las encuentras todas, preguntes en el foro. No avances hasta que tengas claro esto, o te arriesgas a no aprovechar el resto del tema en una primera lectura; y, lo que es peor, perder muchísimo tiempo peleándote con las prácticas sin dar en la tecla, buscando un error en tu sintaxis de Perl cuando se trata de un error conceptual en las expresiones regulares.

Otro concepto importante es el "consumo de carácter". Supongamos que buscamos el patrón:

```
*a*
```

Está claro que este patrón empareja con `cccabbb`. El primer asterisco empareja con la cadena `ccc`. Y ahora viene la pregunta interesante: el segundo asterisco, ¿Con qué cadena empareja? Parece claro que con `bbb`. Si nos fijamos, al emparejar el carácter `a` nos lo hemos "comido". Aunque empareja con el segundo

asterisco, no lo incluímos porque ya lo hemos emparejado en la expresión regular: hemos consumido el carácter al emparejarlo con `a`. Pero en algunos casos nos interesará que el carácter emparejado sea más un "elemento de sincronía", para determinar qué emparejamos con el primer asterisco, y qué emparejamos con el segundo; es decir, algo similar a "de la `a` hacia atrás emparéjamelo con el primer asterisco, `a` incluida; y de la `a` en adelante, `a` incluida, me lo emparejas en el segundo asterisco". La `a`, por lo tanto, estaría emparejada en el segundo asterisco. ¿Y esto es importante? Sí, mucho; ya que Perl meterá en variables especiales lo que ha ido emparejando; por lo que analizar un texto en Perl es pasarle una expresión regular que lo modele, y luego leer las variables especiales asignadas a la expresión regular. Es por esto que nos interesa tanto que consuma o no caracteres; ya que lo interesante de las expresiones regulares en Perl no es apenas el emparejamiento en sí, sino la información que extraemos al haberla emparejado con una entrada –que puede ser un histórico, o la respuesta a un comando Unix, por poner un par de ejemplos–.

Una vez que tenemos todo esto claro, lo normal es que comencemos a sospechar que lo de las expresiones regulares se puede desmadrar fácilmente tanto en memoria como en tiempo de cálculo. Así es. Por eso es muy importante hacer bien la expresión regular, y que el motor de cálculo de las expresiones regulares sea eficiente. Lo segundo sí te lo da Perl; lo primero lo tendrás que hacer tú como programador.

Las expresiones regulares tienen innumerables usos, y no están limitadas apenas a la búsqueda de conjuntos de ficheros dentro de un directorio. Principalmente las usaremos en la programación diaria para buscar subcadenas dentro de una cadena mayor para realizar operaciones específicas. Esto, por ejemplo, permitiría recorrer el archivo de históricos del Apache y recorrer sus lineas campo por campo.

En Perl las expresiones regulares tienen un mecanismo distinto en algunos detalles de como funcionan en la expansión de ficheros del shell y en las expresiones regulares descritas hasta aquí. Vamos a ver cómo son realmente las expresiones regulares en Perl.

6.3. Expresiones regulares el Perl

El potente mecanismo de trabajo con expresiones regulares en Perl es una de las razones por las que Perl se ha hecho tan popular. Pero también es una de las razones de la –mala– fama de ilegibilidad de Perl.

Es cierto que las expresiones regulares tienen una sintaxis un tanto extraña cuando no se les ha cogido el truco. También es cierto que están tan mal explicadas en casi todos los lugares que solo unos pocos se atreven con ellas; y para la mayoría es la parte jeroglífica de Perl. De cualquier forma, son más sencillas de lo que en principio podemos sospechar, y veremos como no entrañan demasiados secretos.

En primer lugar, en una expresión regular en Perl empleamos cadenas de texto para definir expresiones regulares sobre cadenas de texto. Esto es un problema, ya que necesitamos algunos caracteres para "informar" de cosas especiales. Es el caso en los ejemplos anteriores del carácter `*`, que no significa "asterisco", sino "una secuencia de cero o más caracteres".

Los metacaracteres, pues, serán todos aquellos caracteres que tienen algún significado especial. Dominar las expresiones regulares es apenas entender bien para que sirve cada metacarácter, y cómo se aplica. Si necesitamos emplear el carácter ASCII[2] del metacarácter debemos `escapar` el metacarácter, es decir, poner un `\` antes. Por ejemplo, mientras que `*` significa `cero o más caracteres`, `\*` significa `*`, es decir, apenas un asterisco.

6.4. Metacaracteres de expresiones regulares

Como a la hora de imprimir con `print`, en las expresiones regulares existe un conjunto de metacaracteres. Los más comúnmente usados son:

- Todos los códigos de escape de `print`. Todos los códigos de escape de `print` significan lo mismo como metaracacteres de emparejamiento. Por ejemplo, `\t` empareja con el carácter de tabulación, y `\n` con el salto de linea.

- `[]`: emparejamos con uno y solo uno de los caracteres dentro de los corchetes. Por ejemplo, `[abc]hola` empareja con `ahola`, `bhola` y `chola`.

[2] Para ser estrictos, el carácter UTF-8; ya que Perl internamente trabaja en UTF-8. Pero te haces una idea de a lo que me refiero.

- `caracterA-caracterB`: emparejamos con todos los caracteres comprendidos entre `caracterA` y `caracterB`. Ojo, que este mecanismo depende fuertemente de la codificación de caracteres que empleemos en la máquina. Por ejemplo, en ASCII `[a-m]` empareja con cualquier carácter comprendido entre `a` –incluido– y `m` –incluido–.

- `[^]`: no emparejamos con ninguno de los caracteres contenidos dentro de los corchetes. Por ejemplo, `[^abc]` empareja con cualquier carácter que no sea ni `a`, ni `b`, ni `c`. En ASCII, `[^a-m]` empareja con cualquier carácter que no esté comprendido entre `a` –incluido– y `m` –incluido–.

- `\d`: empareja con cualquier dígito. `0` y `9`, por ejemplo, emparejan con `\d`.

- `\D`: empareja con cualquier carácter que no sea dígito. `A` y `)`, por ejemplo, emparejan con `\D`.

- `\w`: empareja con cualquier carácter que sea letra no acentuada –mayúscula o minúscula– salvo la `ñ`; con cualquier dígito y con _. Dicho de otra forma, empareja con las letras del alfabeto anglosajón, con los dígitos, y con _. Por ejemplo, `\w` empareja con `3`, con `d` y con `M`. No empareja con `,`, ni con `í`, ni con `%`, ni con `ç`.

- `\W`: empareja con cualquier carácter que no sea ni letra, ni dígito, ni _. Empareja con todas las letras acentuadas y con diacríticos. Empareja con la eñe mayúscula y minúscula. Dicho de otra forma, empareja con cualquier cosa que no sean las letras del alfabeto anglosajón, con los dígitos, y con _. Por ejemplo, empareja con `ï`, con `@` y con `ú`. No empareja con `w` o con `4`.

- `\s`: empareja con cualquier carácter separador, incluyendo tabulaciones, espacios y retornos de línea. Es, pues, lo mismo que `[\ \t\r\n\f]`.

- `\S`: empareja con cualquier carácter que no sea separador. Es, pues, lo mismo que `[^\ \t\r\n\f]`.

- `.` (el punto): empareja con cualquier carácter, salvo con `\n`.

- `\b`: empareja con un límite entre carácter que empareje con `\w`, y uno que empareje con `\W`; o entre un carácter que empareje con `\W`, y uno que empareje con `\w`. Ojo, que no es lo mismo que `((\w\W)|(\w\W))`. Mientras

que `\b` no consume ningún carácter, `((\w\W)|(\w\W))` consume dos caracteres. Lo del consumo de carácter lo entenderemos en los ejemplos. `\b` se usa para ayudar a parsear tokens.

- `^`: empareja con un inicio de línea. No consume carácter. Por ejemplo, `^ab` significa: `ab aparece al principio de la línea`.
- `$`: empareja con un final de línea. No consume carácter. Por ejemplo, `df$` significa: `df aparece al final de la línea`.
- `|`: empareja con uno o con otro. Por ejemplo, `ab|c` empareja con `ab` y con `ac`. No empareja ni con `a`, ni con `abc`, ni con `ab`, ni con `c`.
- `+`: empareja una o más veces. Por ejemplo, `ab+` empareja con `ab`, con `abb` y con `abbb`. `ab+` no empareja con `a` ni con `abab`.
- `*`: empareja cero o más veces. Por ejemplo, `ab*` empareja con `a`, con `abb` y con `abbb`. `ab+` no empareja con `ba` ni con `abab`.
- `?`: Empareja ninguna o una vez. Por ejemplo, `ab?` empareja con `a` o con `ab`. No empareja con `abb` ni con `abab`.
- `{n}`: Empareja exactamente n veces. Por ejemplo, `ab{3}` empareja con `abbb`. No empareja con `abb`, ni con `abbbb`, ni con `ababab`.
- `{n,}`: Empareja al menos n veces. Por ejemplo, `ab{3,}` empareja con `abbb` y con `abbbb`. No empareja con `abb`, ni con `ab`, ni con `ababab`.
- `{n,m}`: Empareja entre n y m veces. Por ejemplo, `ab{3,5}` empareja con `abbb` y con `abbbb`. No empareja con `abb`, ni con `abbbbbb`, ni con `ab`, ni con `ababab`.
- `()`: agrupa emparejamientos. Por ejemplo, `c(ab)+` empareja con `cab`, y con `cabab`. Finalmente, `(ab)|(cd)` empareja con `ab` o con `cd`. `a(bc)?` empareja con `a` y empareja con `abc`.

6.5. Comprobando un emparejamiento

Comprobar el emparejamiento de un patrón con una cadena es muy fácil en Perl. Basta con indicar la expresión regular que vamos a emparejar entre barras, con la sintaxis:

```
variable =~ m/expresiónregular/opciones
```

lo que devuelve cierto si ha habido algún emparejamiento, y falso en otro caso. Por ejemplo:

```
my $cadena="Tengo una moto";

if ($cadena =~ m/moto/)
{
   print "Tiene una moto\n"
}
   else
{
   print "No tiene una moto\n"
};

if ($cadena =~ m/coche/)
{
   print "Tiene un coche\n"
}
   else
{
   print "No tiene un coche\n"
};
```

Importante, Manuel: de aquí al final del tema, cada ejemplo pruébalo en el ordenador antes de seguir con la lectura. Es fácil perderse con las expresiones regulares; pero si lo haces paso a paso, verás que no son tan complicadas como aparentan.

También tenemos la negación de este operador:

```
variable !~ m/expresiónregular/opciones
```

Lo que devuelve cierto si no ha habido ningún emparejamiento, y falso en otro caso. Por ejemplo:

```
my $cadena="Tengo una moto";

if ($cadena !~ m/moto/)
{
  print "No tiene una moto\n"
}
  else
{
  print "Tiene una moto\n"
};

if ($cadena !~ m/coche/)
{
   print "No tiene un coche\n"
}
   else
{
   print "Tiene un coche\n"
};
```

6.6. Las opciones de emparejamiento

Las opciones de emparejamiento indicadas en `opciones` nos permiten especificar algunos aspectos de como se realizará el emparejamiento. De entre las opciones disponibles, las más importantes son:

- `i`: Es quizás la opción más utilizada. Perl por defecto realiza el emparejamiento por igualdad de carácter a carácter. Esto significa que las palabras `hola` y `Hola` son palabras distintas. Si incluimos la opción `i`, sin embargo, realiza un emparejamiento insensible a la capitalización de letras, lo que significa que `Hola` emparejará con `hola`, con `hOLA` y con `HOLA`. Por ejemplo:

```
my $cadena="HoLa";

if ($cadena =~ m/hola/)
{
   print "Coincide␣exactamente\n";
}
   else
{
   print "No␣coincide␣exactamente\n"
};

if ($cadena =~ m/hola/i)
{
   print "Coincide␣salvo␣en␣mayusculas/minusculas\n";
}
   else
{
   print "No␣coincide\n"
};
```

- g: Empareja globalmente, buscando todas las apariciones de la expresión regular. En búsquedas fijas tiene poco sentido, pero si realizamos algún efecto lateral –como una asignación en el emparejamiento–, sí tiene sentido usar este parámetro. Veremos las asignaciones en el emparejamiento más adelante.

- o: Esta es una opción muy interesante. Le dice a Perl que compile el patrón de búsqueda de la expresión regular en código intermedio, y que no lo evalúe más para esa línea. En la práctica significa que si tenemos un patrón de búsqueda que evaluamos muchas veces obtendremos una mejora de rendimiento considerable. A cambio, el patrón de búsqueda debe ser fijo: si cambia alguna vez, perl ni se dará cuenta del cambio. Por ello, si tenemos variables en la expresión regular no debemos usarlo nunca –hay excepciones, pero es muy fácil pillarse los dedos–, y si no las tenemos debemos usarlo siempre –los programas en Perl se pueden modificar a si mismos, por lo que esto puede no cumplirse; pero si estamos haciendo programas

que se modifican a si mismo probablemente no estaremos preocupándonos por la corrección del `o`–.

- `x`: Usa expresiones extendidas de Perl, que permiten definir expresiones regulares aún más potentes que las vistas en este tema, y que no trataremos en nuestro curso.

6.7. Substituciones de expresiones regulares

Perl nos permite realizar substituciones dentro de una cadena de texto. Esto significa que podemos buscar cualquier expresión regular, y cambiarla por una cadena de texto u otra expresión regular. La sintaxis de la substitución es:

```
s/expresiónregular/sustituto/opciones
```

Donde `expresiónregular` es la expresión regular que buscamos, y el contenido de `sustituto` corresponde a aquello que pondremos en lugar de la expresión regular. Esto devuelve la cadena vacía –es decir, falso– si no ha realizado ninguna sustitución, o el número de sustituciones si ha realizado alguna sustitución. Por ejemplo, podemos hacer:

```
my $texto="Tengo una moto";
print "Antes:($texto)";
$texto=~ s/moto/coche/;
print ",Después:($texto)\n";
```

Sin embargo, esto tiene un problema: sólo sustituimos la primera aparición de `moto` por `coche`. Si, por ejemplo, tenemos:

```
my $texto="Tengo una moto. Ahora estoy contento con mi
   moto.";
print "Antes:($texto)";
$texto=~ s/moto/coche/;
print ",Después:($texto)\n";
```

Sólo sustituiremos la primera aparición de `moto`. Si esto es lo que queríamos, perfecto. Pero muchas veces queremos sustituir todas las apariciones de una expresión regular, y no solo la primera. Esto se hace con la opción `g`. Reescribimos en ese caso el ejemplo anterior como:

```
my $texto="Tengo una moto. Ahora estoy contento con mi
   moto.";
print "Antes:($texto)";
$texto=~ s/moto/coche/g;
print ",Después:($texto)\n";
```

6.8. Opciones de sustitución

Como en el caso del emparejamiento, podemos indicar opciones en la sustitución, que cambian la semántica de dicha operación.

- `i`: realiza la sustitución buscando independientemente de la capitalización. Por ejemplo, mientras que:

  ```
  my $texto="Tengo una Moto";
  print "Antes:($texto)";
  $texto=~ s/moto/coche/;
  print ",Después:($texto)\n";
  ```

  no empareja, por lo que no sustituye "Moto" por "coche", la sentencia:

  ```
  my $texto="Tengo una Moto";
  print "Antes:($texto)";
  $texto=~ s/moto/coche/i;
  print ",Después:($texto)\n";
  ```

  empareja y sustituye "Moto" por "coche".

- `e`: Evalúa también la expresión de sustitución como una expresión regular. Usar esto es un poco complicado, y no lo estudiaremos en principio en este curso.

- `o`: Esta es una opción muy interesante. Como en el caso de la búsqueda, le dice a Perl que compile el patrón de sustitución en código intermedio, lo que incluye tanto de la expresión regular de búsqueda, como la expresión regular o texto de sustitución. Ninguna de las expresiones reguladas involucradas en la sustitución para dicha sustitución será evaluada más. En la práctica significa que si tenemos un patrón de sustitución que evaluamos

muchas veces obtendremos una mejora de rendimiento considerable. A cambio, el patrón de sustitución debe ser fijo: si cambia alguna vez, perl ni se dará cuenta del cambio. Por ello, como en el emparejamiento, si tenemos variables en la expresión regular no debemos usarlo nunca –hay excepciones, pero es muy fácil pillarse los dedos–, y si no las tenemos debemos usarlo siempre –los programas en Perl se pueden modificar a si mismos por lo que esto puede no cumplirse, pero si estamos haciendo programas que se modifican a si mismos probablemente no estaremos preocupándonos por la corrección del `o`–.

- `x`: Usa expresiones extendidas de Perl, que permiten definir expresiones regulares aún más potentes. No lo estudiaremos en este curso.

6.9. Eliminando texto

Un caso particular de sustitución es cuando queremos eliminar una expresión regular dentro de un texto. Esto lo hacemos con el mismo mecanismo de sustitución, solo que dejamos el sustituto vacío. Por ejemplo, eliminamos la primera aparición de la palabra `moto` de la cadena con:

```
my $texto="Tengo una moto. Ahora estoy contento con mi
   moto.";
print "Antes:($texto)";
$texto=~ s/moto//;
print ",Después:($texto)\n";
```

Y eliminamos todas las apariciones de la palabra `moto` en la cadena con:

```
my $texto="Tengo una moto. Ahora estoy contento con mi
   moto.";
print "Antes:($texto)";
$texto=~ s/moto//g;
print ",Después:($texto)\n";
```

Un ejemplo práctico del mundo real: eliminar los espacios antes y después de un campo. Esto es muy común hacerlo cuando se trabaja con bases de datos, después de la extracción y antes de la inserción, si no hicimos el `trim` al extraer de la base de datos. Lo hacemos con las líneas:

```
my $campo="␣␣␣␣Esto␣es␣una␣prueba.␣␣␣␣␣␣␣";
print "Antes:($campo)";
$campo=~ s/^\s+//g;
$campo=~ s/\s+$//g;
print ",Después:($campo)\n";
```

6.10. Trabajando con / en expresiones regulares

En algunos casos, tenemos gran cantidad de símbolos / dentro de la expresión que queremos casar o de la expresión que queremos sustituir. Esto es extremadamente pesado de programar, ya que debemos escapar uno a uno cada uno de los símbolos /. Es el caso, por ejemplo, de cuando trabajamos con rutas de archivos.

La solución que nos propone Perl es sustituir / por cualquier símbolo no alfanumérico. Ojo, que esto sólo lo podemos hacer cuando no omitimos el operador de operación con expresiones regulares. Por ejemplo, podemos hacer:

```
my $cadena="Tengo␣una␣moto";

if ($cadena =~ m*moto*)
{
   print "Tiene␣una␣moto\n"
}
   else
{
   print "No␣tiene␣una␣moto\n"
};

if ($cadena =~ m*coche*)
{
   print "Tiene␣un␣coche\n"
}
   else
{
  print "No␣tiene␣un␣coche\n"
```

```
};
```

Obteniendo el mismo resultado que en el ejemplo del texto. De cualquier forma, es común en Perl mantener el / por motivos de legibilidad, y porque cambiar el símbolo despista al `syntax hilighting` de muchos editores de texto editando código Perl.

6.11. Emparejamiento rápido en Perl

Una versión extremadamente rápida del emparejamiento es:

```
?expresiónregular?
```

Lo que busca la expresión regular indicada una única vez. Es más rápido que `m/expresiónregular/`. Por ejemplo, podemos hacer:

```
my $cadena="Tengo una moto";

if ($cadena =~ ?moto?)
{
   print "Tiene una moto\n"
}
   else
{
   print "No tiene una moto\n"
};

if ($cadena =~ ?coche?)
{
   print "Tiene un coche\n"
}
   else
{
   print "No tiene un coche\n"
};
```

6.12. Revisitando los operadores de cadena

Ahora estamos en el punto de revisitar los operadores de cadena, para un operador que no llegamos a tratar: el operador `split`.

El operador `split` Parte una cadena en subcadenas, empleando para ello una expresión regular. La expresión regular identificará la subcadena que separa los elementos de la cadena principal. Su sintaxis es:

```
split(expresiónregular, cadena, límite);
```

Donde `expresionregular` será la ya comentada expresión regular, `cadena` la cadena que deseamos partir, y `límite` el número máximo de elementos que queremos extraer. Esta función, lanzada en un entorno de vector –es decir, lanzada donde Perl espera un vector–, devolverá una lista con dichos elementos identificados; y en entorno escalar –es decir, lanzada donde Perl espera un escalar– devuelve el número de elementos que pueden ser extraídos en base a dicho criterio[3]. Es importante que recordemos que si omitimos `límite` extraerá todos los elementos disponibles, y si omitimos `cadena` operará con `$_`; que recordamos que es una variable comodín que significa "lo que nos traemos entre manos".

Por ejemplo:

```
my $cadena="Hola␣a␣todos!";
my @palabras_de_cadena=split(/\W/,$cadena);
my $numero_palabras=split(/\W/,$cadena);
# Veamos los resultados:
print "Ha␣habido␣$numero_palabras␣palabras;␣que␣son␣".join
   (',',@palabras_de_cadena)."\n";
```

Vamos a poner un ejemplo más interesante, poniéndolo todo junto. Podemos parsear tanto todas las líneas como todos los campos de el archivo de claves de Unix con:

```
 open(PASSWD, '/etc/passwd');
    while (<PASSWD>) {
```

[3]Intencionalmente he omitido los conceptos de "entorno escalar" y "entorno vectorial" para no hacer más confuso el curso; pero ha llegado el momento de comentarlo. Muchas funciones de lista se comportan de forma distinta si lo que devuelven se lo asignamos a una lista, o si se lo asignamos a un escalar. Es un buen momento para que pruebes las funciones de las que hablamos hace dos capítulos, y veas que hacen en ambos entornos.

```
        chomp;
        ($login, $passwd, $uid, $gid,
           $gcos, $home, $shell) = split(/:/);
         #Aquí ponemos el código que queramos que
         #procese los distintos campos
         print "Nueva␣linea␣de␣password:␣usuario␣$login␣
            con␣home␣en␣$home\n";
    }
```

Aquí estamos utilizando gran parte de lo que ya hemos visto. Como novedad, abrimos un fichero con `open`, asignando como nombre de manejador la palabra `PASSWD`; e iteramos dicho fichero poniendo el manejador entre símbolos `<>` dentro de un `while`, con lo que se ejecutará una iteración de `while` por cada línea del fichero abierto con el manejador `PASSWD`. También es importante destacar que hemos omitido en el `split` qué es lo que estamos partiendo; por lo que partirá `$_`, es decir, cada linea del fichero.

6.13. Asignando valores a variables a través de expresiones regulares

Queda el último elemento clave de las expresiones regulares en Perl: asignar lo emparejado a variables. Y para eso haremos uso de las "variables comodines" o variables especiales. El caso más simple es cuando estamos buscando una cosa muy concreta. En ese caso, dispondremos de las variables `$`, `$&`, `$'` donde:

- `$`: Valdrá lo que Perl haya encontrado antes del último emparejamiento.
- `$&`: Valdrá lo que Perl haya encontrado como último emparejamiento
- `$'`: Valdrá lo que Perl haya encontrado después del último emparejamiento.

A veces queremos cosas más complejas; por ejemplo, podemos querer parsear los números de una IP. Aunque se puede hacer con `split`, siempre hay más de una forma de hacerlo en Perl. Podemos extraer una IP de una línea de texto mediante la expresión regular:

```
/(\d+)\.(\d+)\.(\d+)\.(\d+)/
```

Luego podemos emparejar con:

```
$linea=~ /(\d+)\.(\d+)\.(\d+)\.(\d+)/;
```

Pero ¿Como extraemos los dígitos analizados? Perl nos va a meter en una variable especial cada una de las cadenas emparejadas que hemos puesto entre paréntesis; será `$1` para la primera, `$2` para la segunda, y así para todos los subpatrones emparejados, donde identificamos un subpatrón como la parte del patrón dentro del paréntesis. Por ejemplo, podemos hacer:

```
open(FICHERO, '/etc/hosts');
   while (<FICHERO>) {
       /(\d+)\.(\d+)\.(\d+)\.(\d+)/;
       print "Los␣dígitos␣de␣las␣IPs␣del␣hosts␣son:␣
          primero␣$1,␣después␣$2,␣después␣$3␣y␣finalmente
          ␣$4\n";
   }
```

Aquí hemos hecho "trampa". Recordemos que `$_` es “lo que me traigo entre manos”; luego el programa anterior es idéntico a:

```
open(FICHERO, '/etc/hosts');
   while (<FICHERO>) {
       $_ =~ /(\d+)\.(\d+)\.(\d+)\.(\d+)/;
       print "Los␣dígitos␣de␣las␣IPs␣del␣hosts␣son:␣
          primero␣$1,␣después␣$2,␣después␣$3␣y␣finalmente
          ␣$4\n";
   }
```

Lo más interesante de Perl es que podemos utilizar esas variables dentro de la expresión regular, pero empleando la barra invertida, en lugar del dólar. Por ejemplo, para hacer un programa que tome un texto y sustituya en dicho texto las mayúsculas por una cadena en la que esté la orden de apertura y cierre de entorno de negrita en LaTeX 2ε, y tenga dicha letra en su interior, hacemos:

```
$_ = "Texto␣donde␣queremos␣poner␣las␣mayusculas␣con␣
   Negritas␣tipo␣LaTeX.";
```

```
# El complejísimo programa

        s/([A-Z])/\{\\bf \1\}/g;

# E imprimimos el resultado
 print "$_";
```

Ahora, y antes de pasar a las actividades, revisa en los temas precedentes las expresiones regulares que he empleado. Intenta deducir qué son, y para qué sirven. Para aquí, Manuel; juega un poco con las expresiones regulares, y cuando creas que has entendido el tema, pasa a la actividad.

6.14. Actividades

La actividad de este tema será un programa que parsee historicos tal y como:

```
invent.scansafe.net - - [17/Dec/2011:15:41:13 +0100] "GET
   /wp-content/themes/adventure-journal/images/mp-
   navigation-main-sprite.png HTTP/1.1" 304 - "http://www.
   arquerosdebosque.es/" "Mozilla/4.0 (compatible; MSIE
   7.0; Windows NT 5.1; GTB7.0; .NET CLR 1.1.4322; .NET
   CLR 2.0.50727; .NET CLR 3.0.04506.648; .NET CLR
   3.5.21022; InfoPath.2; Tablet PC 1.7; .NET CLR
   1.0.3705; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729;
   LLY-EMA-EN; LLY-EMA-EN)"
invent.scansafe.net - - [17/Dec/2011:15:41:13 +0100] "GET
   /wp-content/themes/adventure-journal/images/mp-sprite-
   header-bg2.png HTTP/1.1" 304 - "http://www.
   arquerosdebosque.es/" "Mozilla/4.0 (compatible; MSIE
   7.0; Windows NT 5.1; GTB7.0; .NET CLR 1.1.4322; .NET
   CLR 2.0.50727; .NET CLR 3.0.04506.648; .NET CLR
   3.5.21022; InfoPath.2; Tablet PC 1.7; .NET CLR
   1.0.3705; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729;
   LLY-EMA-EN; LLY-EMA-EN)"
invent.scansafe.net - - [17/Dec/2011:15:42:02 +0100] "GET
   /como-asociarte/ HTTP/1.1" 200 5327 "-" "Mozilla/4.0 (
   compatible;)"
96.244-81.adsl-static.isp.belgacom.be - - [17/Dec
   /2011:15:43:36 +0100] "GET /competiciones.ficheros/
   RESULTADOS_IFAA_MEDITERRANEO_ALICANTE_2011.pdf HTTP
   /1.1" 200 162622 "http://www.arquerosdebosque.es/
   competiciones/" "Mozilla/4.0 (compatible; MSIE 8.0;
   Windows NT 6.0; Trident/4.0; SLCC1; .NET CLR 2.0.50727;
    InfoPath.2; .NET CLR 3.5.30729; .NET CLR 3.0.30729)"
crawl-72-182.googlebot.com - - [17/Dec/2011:15:57:01
   +0100] "GET /2011/12/06/ HTTP/1.1" 200 10811 "-" "
   Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.
   google.com/bot.html)"
```

Esto ya es un fichero real –con las direcciones IPs cambiadas, por privacidad– y corresponde a un histórico de un servidor Apache.

El programa deberá leer de entrada estandard el fichero –en Perl, el manejador de entrada estándar es `<STDIN>`–, parsearlo linea a línea, e introducirlo en una lista de hashes –si otra estructura te parece más cómoda, también me vale–; a partir de ahora voy a ser muy flexible con la forma de hacer las cosas.

La estructura, una vez analizada, la tienes que pasar a una función de impresión que harás, y debe volcar todas las entradas parseadas. Los datos que tienes que extraer de cada entrada son:

- Nombre de la máquina que ha pedido la página.
- Fecha en que la página se ha pedido.
- Hora que la máquina se ha pedido.
- Página obtenida, **sin comandos GET, POST o análogos**.
- Nombre del navegador que pidió la página.

Para que la tarea sea valorada como apta, el programa debe funcionar bien, y hacer lo que te pido. Lo verificaré contra un fichero de históricos real, y te mandaré de vuelta el error si no lo parsea bien; por lo que te recomiendo que el programa sea lo suficientemente verboso como para que puedas identificar el error.

Esta es la última actividad obligatoria del curso; por lo que en el momento que me mandes lo que pido, habrás concluido la parte "obligatoria" del curso, y tendrás derecho a tu diploma. Si quieres seguir, estupendo; hay aún cuatro temas interesantísimos para estudiar y que vamos a ver en los próximos temas.

Te recomiendo antes de que te pongas en faena que pruebes los ejemplos. Si pruebas todos los ejemplos y entiendes bien las expresiones regulares, esta práctica es muy sencilla. Si no lo has entendido, esta práctica es un infierno.

Después de haber probado los ejemplos de los apuntes, y de haber intentado el ejercicio, no dudes en utilizar el foro para preguntar todo lo que no entiendas.

Capítulo 7

Escribiendo proyectos grandes en Perl

Según nuestros programas en Perl se vallan haciendo más grandes, deberemos incluir nuevos métodos de trabajo que permitan manejar la complejidad asociada a proyectos mayores. Esto supondrá usar paquetes, o aprovechar las características orientadas a objetos de Perl y usar clases. También supondrá aprender a depurar Perl.

Los programas, según se hacen más grandes, son mucho más complejos de depurar. Y Perl no es una excepción en ello. A partir de determinado tamaño la depuración se puede volver mucho más compleja, por lo que comienza a ser interesante hacer uso de las características de Perl destinadas a proyectos de gran envergadura: los paquetes y la orientación a objetos.

En este tema vamos a estudiar aquellas características de Perl que permiten desarrollar código fuertemente estructurado, lo que nos permitirá emplearlo como lenguaje de propósito general para grandes proyectos, y no sólo un lenguaje para solucionar rápidamente los problemas del administrador de sistemas –algo que, por otro lado, también hace Perl–. En el próximo tema veremos de forma somera la programación orientada a objetos bajo Perl-

7.1. Definiendo paquetes en Perl

Hasta ahora hemos usado paquetes en todos los programas en Perl que hemos hecho hasta ahora: cada vez que incluíamos la palabra clave `use`, estábamos

usando un paquete[1]. Ahora aprenderemos a crearlos.

El paquete es la unidad de organización del código inmediatamente superior a las funciones: un paquete reúne a varias funciones a las que les queremos dar un uso lógico conjunto.

Sin embargo, no es apenas un mecanismo de organización interna del código: cada paquete tiene un espacio de nombres propio, por lo que podemos tener dos variables distintas o dos funciones distintas con el mismo nombre en un mismo programa, mientras que estén en paquetes distintos. Esto es diferente, por ejemplo, de los ficheros de cabecera de C[2] y de mecanismos similares de organización de subrutinas, que no corresponden con la idea de paquete en Perl.

Realmente en Perl todo, queramos o no, está dentro de un paquete. Si no lo especificamos explícitamente en el fichero, las funciones y las variables que decaremos estarán en el espacio de nombres del paquete principal, el paquete `main`.

Para definir que de una posición de un fichero en adelante lo que definamos estará en un paquete distinto emplearemos la palabra clave `package`. Su sintaxis es:

```
package nombredepaquete
```

y significa, literalmente, "de aquí en adelante, hasta final de fichero o hasta que aparezca otra cláusula `package`, todo lo que definamos en este fichero pertenece al paquete `nombredepaquete`".

Algunos lenguajes tienen conceptos como el de módulo, que permite tener grupos de espacios de nombres distintos. Sin embargo, un paquete no corresponde exactamente con este concepto; ya que, a diferencia de casi todos los lenguajes, en Perl el concepto de paquete es un concepto dinámico. No se definen los paquetes en tiempo de compilación –que no existe, al ser Perl interpretado– ni en tiempo de carga –como podríamos suponer–, sino en tiempo de ejecución. Esto en principio es indistinguible para el neófito del concepto de módulo, pero nos podremos llevar alguna sorpresa al usar `eval` con paquetes. O dicho de otra forma; en Perl un programa se puede modificar a sí mismo, puede crear código dinámicamente, crear funciones dinámicamente, y trabajar con funciones de

[1] Realmente estábamos trabajando con clases, ya que son paquetes que incorporan clases. Pero son paquetes, al fin y al cabo.

[2] De hecho, la existencia de un único espacio de nombres en C puede hacer la incorporación simultanea de varias bibliotecas un infierno.

orden superior; por lo que tiene poco sentido que la estructura de paquetes se organice de forma estática al cargar el programa.

7.2. Usando paquetes en Perl

A través de este curso hemos aprendido un uso básico de los paquetes. Ahora vamos a aprender a usar los paquetes como lo que son –espacios de nombres independientes–.

Para referirnos a una función dentro de un paquete, usaremos el formato:

```
paquete::función
```

Así Perl buscará la función `función` dentro del espacio de nombres `paquete`. Si no indicamos el nombre del paquete, se supone que nos referiremos al paquete `main`, por lo que la llamada:

```
main::función()
```

llama a la misma función que:

```
función()
```

Las variables definidas dentro de un paquete se referencian de forma análoga, pero indicando el carácter de tipo antes del nombre de paquete. Por ejemplo, la variable escalar `$variable` definida en el paquete `paquete`, podemos referenciarla con:

```
$paquete::variable
```

Por omisión, nos referiremos al paquete `main`; por lo que la variable:

```
$main::variable
```

será la misma variable que:

```
$::variable
```

y que:

```
$variable
```

Podemos importar en masa otro espacio de nombres, e incluirlo en `main`. Esto se hará con `use`: ya sabemos qué es lo que realmente hacía el `use` que hemos empleado en los temas anteriores.

También podemos acceder directamente a la tabla de símbolos del espacio de nombres asociado a un paquete. Esto se consigue gracias a que en Perl dicha tabla de símbolos se organiza como un hash, y tiene como nombre el nombre del paquete. Por ejemplo, si tenemos un paquete denominado `mipaquete`, podremos acceder directamente a su tabla de símbolos a través de `%mipaquete`[3].

7.3. Inicializadores y finalizadores de paquetes

Algunos paquetes pueden necesitar que inicialicemos variables, o que ejecutemos determinadas funciones antes de comenzar a ejecutarse. Estas inicializaciones pueden depender de la estructura interna del paquete, por lo que iría contra las buenas costumbres de la programación estructurada forzar al código que usa dicho paquete realizar las inicializaciones.

Perl nos da una característica muy potente para resolver este problema: los inicializadores.

Cuando usemos un paquete, debemos saber que antes del primer uso de cualquier método o función de un paquete se habrán ejecutado siempre todos los inicializadores de este. Definimos un inicializador dentro del paquete con:

```
BEGIN
        {
 líneas de código del inicializador
        };
```

Podemos tener tantos inicializadores como queramos, y se ejecutarán en el orden relativo en el que estén dentro del código. El inicializador no es programación orientada a objetos, y no necesitamos utilizar programación orientada a objetos para hacer uso de un inicializador. Pero si sabes programación orientada a objetos, Manuel, se te debe haber ocurrido una analogía muy pertienente del concepto de inicializador de Perl en el mundo de la programación orientada a objetos: el constructor de clase.

[3] Si no entendemos esto, no nos debemos preocupar. Es otra de las numerosísimas características esotéricas de Perl que no usaremos habitualmente.

Otro problema con el que nos enfrentamos es el de liberar recursos al terminar el uso de un paquete. Si bien la mayor parte de los recursos más críticos los libera Perl –como puede ser la memoria–, hay recursos que es interesante liberar a mano, tal y como un segmento de memoria compartida System V, o un bloqueo de fichero. El primero nunca será liberado por el intérprete de Perl, y el segundo solo será liberado al morir el proceso, lo que puede tardar mucho tiempo en el caso de los servidores.

La solución que Perl nos da son los finalizadores. Los finalizadores siempre se ejecutan en el último uso del paquete, lo que suele corresponder con el justo momento en el que se termina la ejecución del programa en Perl, pero puede ser de otra forma si desimportamos el paquete. Definimos un finalizador con:

```
END
        {
        lineas de código del finalizador
};
```

Podemos tener tantos finalizadores como queramos dentro de un paquete. Se ejecutarán por orden inverso al que estén dentro del fichero que los defina. Como en el caso del inicializador, el finalizador no es programación orientada a objetos, y no necesitamos utilizar programación orientada a objetos para hacer uso de un finalizador. Pero si sabes programación orientada a objetos, Manuel, se te debe haber ocurrido una analogía muy pertienente del concepto de finalizador de Perl en el mundo de la programación orientada a objetos: el destructor de clase.

7.4. Definiendo y usando módulos en Perl

El paquete es una estructura interesante para modular un archivo, pero aún podemos querer un paso más en la estructuración de nuestro programa. Este siguiente paso son los módulos.

Un módulo es un paquete que se almacena en un fichero independiente. Todas las funciones y variables asociadas a dicho paquete pueden ser encontradas en un único archivo, denominado con el nombre del paquete y la extensión `.pm`.

Es importante que destaquemos que la última línea del fichero de módulo debe ser siempre:

```
1;
```

lo que permitirá funcionar correctamente al `use` y al `require`.

La ventaja del módulo sobre el paquete es que el paquete exporta todo su espacio de símbolos al programa que lo usa, mientras que en el módulo el espacio de símbolos permanece privado, y podemos especificar qué exportamos haciendo uso del módulo `Exporter` desde nuestro módulo. Usando este módulo, exportaremos implícitamente apenas los símbolos que estén en la matriz `@EXPORT`, y se podrán importar explícitamente desde el cliente aquellos símbolos definidos en la matriz `@EXPORT_OK`.

Los módulos en Perl se convierten en el componente básico de la reusabilidad de código, ya que nos permiten definir bibliotecas –cada módulo corresponde con una biblioteca–; y el mecanismo de importación de módulos puede incluir tanto un módulo para interpretar –`.pm`– como un módulo compilado –`.so`– de forma transparente al programador del programa que importa el módulo[4].

7.5. Depuración de programas en Perl

Lo primero antes de comenzar a depurar en Perl es hacer las cosas bien: programemos higiénicamente. Usemos orientación a objetos, definamos tipos abstractos de datos, indentemos correctamente nuestros programas, y hagamos subrutinas donde sea necesario.

Hay también algunos consejos que son recomendables: escribir la documentación del programa según se hace, depurar y corregir según desarrollamos el programa, y utilizar un editor de textos con *Syntax highlighting* –lo que nos ayudará con los errores con comillas simples y dobles, así como con los errores de sintaxis–. Debemos de buscar un editor que también sea capaz de comprobar que paréntesis y corchetes están equilibrados. Yo utilizo el editor `vi`, que me da todo eso; pero la decisión de qué editor debes usar es muy personal[5].

Otro consejo muy práctico es poner todos los paréntesis que podamos en las expresiones matemáticas y lógicas. Un exceso de paréntesis no mata, y facilita

[4] Sé que he dicho que Perl es interpretado, y que por ello no se puede compilar. Pero te he mentido, Manuel. Bueno, realmente lo primero es una media verdad, y lo segundo es una flagrante mentira. De cualquier forma, saber esto es intrascendente para programar Perl al nivel que trabajaremos en el curso, por lo que seguiremos ignorando este hecho para no complicar innecesariamente el curso.

[5] En algunos editores, incluso física: probablemente especies con más de ocho dedos por mano puedan mostrar su plena satisfacción con EMACS.

la lectura del código. De hecho, los paréntesis son gratis. Y no afectan al rendimiento. Además, algunas de las reglas de precedencia de Perl no son obvias, y nos podemos encontrar con sorpresas si suponemos demasiadas cosas a este respecto.

Por último, la regla de oro de los trucos básicos en Perl para no tener problemas: programamos para dar instrucciones a la máquina, no para demostrar a nadie que sabemos mucho. Perl permite hacer las cosas realmente complicadas y ofuscadas, pero debemos ir siempre al método más simple y claro de resolver el problema. Y cuando digo que se puede ofuscar, es que se puede ofuscar. Esto es un programa válido en Perl:

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

use Lingua::Romana::Perligata;

maximum inquementum tum biguttam egresso scribe.
meo maximo vestibulo perlegamentum da.
da duo tum maximum conscribementa meis listis.

dum listis decapitamentum damentum nexto
    fac sic
        nextum tum novumversum scribe egresso.
        lista sic hoc recidementum nextum cis vannementa
           da listis.
    cis.
```

Esto es otro programa válido en Perl:

```
@P=split//,".URRUU\c8R";@d=split//,"\nrekcah xinU / lreP
   rehtona tsuJ";sub p{@p{"r$p","u$p"}=(P,P);pipe"r$p","
   u$p";++$p;($q*=2)+=$f=!fork;map{$P=$P[$f^ord($p{$_})
   &6];$p{$_}=/ ^$P/ix?$P:close$_}keys %p}p;p;p;p;p;map{$p{
   $_}=~/^[P.]/&&close$_} %p;wait until$?;map{/^r/&&<$_>} %p
   ;$_=$d[$q];sleep rand(2)if/\S/;print
```

Y esto también es un programa válido en Perl:

```
join eval tell rand reverse ord chr eval split xor
uc prototype eval lcfirst join chmod kill eval ref
split sprintf reverse times xor not eval and srand
tell sqrt formline eval ord lcfirst ucfirst length
 glob gmtime exp defined caller or binmode log ord
abs lc sqrt study alarm split time or formline cos
ne rewinddir kill chdir reset prototype split sqrt
ord int localtime abs oct pack pop eq scalar print
telldir open unpack return and unlink write chroot
hex bless utime split chown split close rmdir join
exp fileno getc sleep redo glob mkdir stat ne pack
reverse getpwnam next lstat gethostent and getpgrp
 eq log ord time xor chr undef and eval caller and
printf srand lstat chown chdir syscall open select
eq -w closedir sleep chr split and quotemeta reset
require ne closedir sleep chr undef or pack unpack
length splice shift umask readpipe pos xor defined
 join system and die or do exit if defined require
hex defined undef or sprintf localtime cmp time or
abs time and undef and open exp getc fileno system
caller eof rewinddir readpipe splice shift defined
kill pop wantarray and readlink eof readpipe split
eval warn join study abs localtime oct log time or
reverse xor open 0; print chr ord while readline 0
,;print chr abs length time for cos length getppid
```

Me niego a enseñarte en un curso de introducción a Perl porqué funcionan estas cosas. A hacer guarradas ya aprenderás solito, no te preocupes.

7.6. Localizando errores sintácticos

En un lenguaje normal, no habría un apartado sobre "localizando los errores sintácticos". En condiciones normales, el compilador indicaría los errores de sintaxis. El problema es que Perl es tan flexible que algunas cosas que otros len-

guajes darían como error Perl las acepta–por flexibilidad, por comodidad, por potencia, o para permitirte trabajar más rápido cuando tienes urgencia–, por lo que los errores sintácticos pueden manifestarse en tiempo de ejecución de formas no triviales, y por ello será importante saber como localizar estos errores.

Los errores sintácticos los encontramos incluyendo al principio de nuestro programa en Perl la linea:

```
use strict;
```

Esta linea le pide al compilador que se tome la gramática de Perl en serio. **Siempre deberíamos incluir esta linea en todo lo que pase de un hack rápido**. También debemos incluir en todo programa que hagamos en Perl la linea:

```
use warnings;
```

esta linea avisará de cosas que, aun siendo legales en Perl, al compilador le parecen raras; por lo que nos permitirá localizar que algo está fallando antes de que se nos manifieste el error.

Aún podemos obtener más información útil de depuración. Incluyendo la linea:

```
use diagnostics;
```

Obtendremos errores con una ayuda sobre lo que puede estar pasando. La ayuda no siempre es buena, pero al menos será un buen punto de comienzo.

Por último, podemos utilizar el módulo de servicio `Lint`, que hace las funciones del `lint` de C: localizar algunos errores adicionales. Como todos los módulos de servicio, no se incluyen con la opción `use`, sino que se ejecutan al arrancar el script. Para usarlo, lanzamos el script con:

```
perl -MO=Lint programa.pl
```

donde `programa.pl` es nuestro script. El módulo de servicio `Debug` nos dará aún más información sobre cómo está organizado un programa en Perl y así encontrar errores absurdos, pero debemos conocer bastante sobre como está implementado Perl para entender su salida.

7.7. Comencemos por lo obvio

El primer paso al depurar es comenzar por lo obvio: al programar, utilizar programación defensiva.

La programación defensiva supone hacer uso de un número de técnicas y trucos para que, de producirse errores, estos sean más evidentes. Cada programador tiene sus trucos personales, y se podría escribir un libro sobre ellos, pero vamos a limitarnos a los más importantes.

El primero es que ojo con el ==. Si nos equivocamos tecleando, podemos poner por error un =, con lo que, por ejemplo, la operación:

```
if($a==3){$a++};
```

si la tecleamos como:

```
if($a=3){$a++};
```

se convierte en:

```
$a=3;
if(0==0){$a++};
```

es decir:

```
$a=4;
```

Este error nos puede volver realmente locos[6]. Y la solución es simple y elegante: cuando comparamos una constante y una variable, ponemos primero la constante, y después la variable. Poniendo:

```
if(3==$a){a++};
```

es decir, la constante a la izquierda y la variable a la derecha, aunque nos olvidemos de un = no tendremos problema, ya que generaremos un error de sintaxis, que será siempre más fácilmente localizable que un error de ejecución.

Otra operación obvia para no cometer errores es comprobar religiosamente todos los valores de vuelta de todas las funciones que sea sensato comprobar. Las

[6] La de alumnos de primero de carrera que habré suspendido gracias a este error; no porque lo considere eliminatorio, sino porque se encasquillan en el examen aquí y no siguen. Y la de programadores profesionales que siguen cometiendo esto día sí y día también...

funciones pueden no funcionar, pueden generar errores: aquella función de abrir socket puede que no lo abra, o aquella función de mandar un querry a la base de datos puede que genere un error porque la base de datos esté demasiado cargada en dicho momento. Por ello, es recomendable comprobar que las funciones han funcionado correctamente. Para ello es frecuente hacer algo como:

```
$sth->execute() || die("Fallo en conexión con la base de
   datos:$DBI::errstr\n");
```

o:

```
die("Fallo en conexión con la base de datos:$DBI::errstr\n
   ") unless $sth->execute() ;
```

Ambos mecanismos hacen lo mismo: si la función `$sth->execute()` falla, generan un error que aborta el programa.

Quizás la única excepción a esta regla serán las llamadas a funciones en la parte más interna de los bucles, donde dicha comprobación ralentizaría realmente la ejecución del programa. Aquí quizás será una buena idea en la versión de producción comentar la comprobación más interna; siempre y cuando un error no suponga corrupción de datos, y además inmediatamente después de dicho bucle más interno sea posible comprobar eficientemente que se ha hecho toda la operativa correctamente, y se incluya esta comprobación. Estas condiciones no siempre son posibles, por lo que algunas veces tendremos que dejar estas comprobaciones en los bucles más internos de nuestros programas.

Otro consejo obvio es, si tenemos información sobre el error, imprimirla. Por ejemplo, imprimir el aserto que ha sido violado, el parámetro que no tiene sentido, o la variable `$^E`, que, si es error de sistema, contiene una descripción del error generado; o imprimir la variable `$!`, que contiene el código de dicho error de sistema. Quizás la única excepción de esta regla es cuando la información volcada puede suponer un riesgo de seguridad –por ejemplo, números de cuentas bancarias o claves personales–; pero casi siempre la información que manejan nuestros programas no es tan sensible, y no es problemático un volcado en pantalla que permita trazar el error. En el caso de programas en desarrollo, siempre

se debe volcar el resultado de los errores.

7.8. Avisando de los errores

A la hora de advertir de los errores tenemos tres mecanismos, cada uno con sus ventajas y sus inconvenientes: estos son `print`, `warn` y `die`.

- `print`: Es la impresión normal. Este es el mecanismo recomendado para la impresión de variables, de estados internos y de todas esas cosas que queremos leer en desarrollo, pero que quitaremos en producción.
- `warn`: Avisos de que algo puede ir mal. Cuando un módulo detecta un error o un problema, puede que no tenga información para saber si el error es intencionado o no. En este caso, el módulo debe imprimir un aviso con `warn`, y devolver un código de error al llamante. El llamante, con más información, puede que sepa que algo salió mal, pero que sepa que no es problemático, o que pueda arreglarlo.
- `die`: Hay casos en los que el problema es tan grave que el código que descubre el error tiene suficiente información como para tener seguridad que hay que abortar el programa. En este caso se usa `die`, que imprime un mensaje de error y aborta el programa.

Dentro de un bloque `eval` podemos utilizar estas tres instrucciones. Es especialmente interesante en este caso que sepamos que es posible propagar los errores de `warn` y `die`. `die` termina el bloque `eval` automáticamente, almacenando en `$@` el mensaje de `die`. Podemos propagar este error a un `warn` fuera del bloque eval, para avisar al usuario que hemos cazado el problema y lo conocemos, o podemos propagarlo a un `die`, si suponemos que este error es demasiado grande como para poder continuar el desarrollo del programa.

La propagación del error sólo funciona si no indicamos el error en el `die` o en el `warn` que recibe la propagación. Por ejemplo, el programa:

```
eval
{
eval
        {
```

```
                die "Salimos del primero";
        };
        die "Salimos del segundo";
};
warn "Lo he cazado, continuamos";
```

sólo imprimirá el último error, con la salida:

```
Lo he cazado, continuamos at prop.pl line 11.
```

Lo correcto habría sido hacer:

```
eval
{
eval
        {
                die "Salimos del primero";
        };
        die;
};
warn;
```

que sí permite ver como se ha propagado el error fuera de los `eval`, y finalmente se ha recogido para ser solucionado, obteniendo la salida:

```
Salimos del primero at prop.pl line 7.
        ...propagated at prop.pl line 9.
        ...caught at prop.pl line 11.
```

También tenemos dos macros muy útiles para informar sobre el error; que serán `__LINE__` y `__FILE__`. La primera contiene la linea actual, y la segunda el fichero actual. Esto nos permitirá advertir desde un `print` en qué linea estamos, y en qué fichero estamos.

Por último, es interesante que usemos el módulo `Carp`. Este módulo está pensado para permitirnos notificar errores, pero con un mensaje cómodo de entender para un programa que llame al módulo donde se produce un error. En `Carp` tendremos cuatro formas nuevas de notificar errores: `carp`, `cluck`, `croack` y `confess`.

- `carp`: es como `warn`, pero con mensajes cortos de error para que un programa que use el módulo que comete el error entienda el error.
- `cluck`: es como `carp`, pero volcando información mucho más detallada de como se genera el error. Debe importarse de `Carp` explícitamente.
- `croack`: es como `die`, pero con mensajes cortos de error para que un programa que use el módulo que comete el error entienda el error.
- `confess`: es como `croack`, pero volcando información mucho más detallada de como se genera el error.

Además, tenemos la rutina `shortmess`, que nos genera el mensaje de `carp` y de `cluck`, y la rutina `longmess` que nos genera el mensaje de `croack` y `confess`. Tanto `shortmess` como `longmess` deben importarse explícitamente.

7.9. Controlando el paso de parámetros

Perl es un lenguaje de tipabilidad discutible, como sabemos ya a estas alturas.Esto es un problema muy grave, ya que supone que muchos errores del principiante se dan por el paso de parámetros a funciones.

No vamos a entrar en este tema en cómo funciona el paso de parámetros en Perl internamente –apenas basta con que sepamos que Perl internamente mete los parámetros en una pila en la función llamante, y en la función llamada internamente los saca de dicha pila para componer `@_`–. Apenas vamos a entrar en como asegurarnos que los parámetros que se pasa una función son correctos.

El primer paso para un paso de parámetros sin problemas es asegurarnos de que la función tiene aquellos parámetros que necesita. La primera aproximación que podemos tomar en Perl es el uso de prototipos. Un prototipo define exactamente el número de parámetros que necesitará la función. Si tiene más o menos parámetros de los prototipados, el programa generará un error. Definimos un prototipo en la definición de la función con la sintaxis:

```
sub función(prototipo)
{
instrucciones;
}
```

o, antes de la definición, con:

```
sub función(prototipo);
```

Donde `prototipo` es una lista de los siguientes caracteres:

- `$`: el argumento es un escalar.
- `%`: el argumento es una tabla hash.
- `@`: el argumento es un vector.
- `&`: el argumento es una función.

Ojo: tanto `@` como `%` "absorben" las variables posteriores. Esto quiere decir que si mandamos tres escalares a un prototipo `$@$` tendremos un error, ya que el `$` absorbe el primer escalar, y `@` "absorbe" los dos siguientes.

Particularmente, podemos forzar a que se incluya un elemento del tipo determinado en vectores y tablas hash poniendo un `\` antes del símbolo en el prototipo. Por ejemplo, un prototipo:

```
sub fn($@)
```

lo podemos llamar con:

```
fn $m $n $o;
```

Pero un prototipo:

```
sub fn($\@)
```

no lo podemos llamar con:

```
fn $a $b $c;
```

y tenemos que llamarlo forzosamente con:

```
fn $a ($b,$c);
```

o con:

```
fn $a @b;
```

Si queremos poner números de parámetros variables, lo haremos con `;`. Los parámetros que aparezcan después del `;` serán variables.

Podemos poner también prototipos antes de la definición de la función. Basta con indicar la cabecera con su prototipo, pero sin el bloque.

7.10. Comprobando parámetros sin prototipos

Este planteamiento tiene un problema: muchas veces queremos trabajar con funciones que admitan número de parámetros variables, con patrones específicos –por ejemplo, acepta sólo tres o cinco parámetros, pero no aceptes cuatro parámetros- -y el prototipo nos fuerza a trabajar con un número de parámetros fijos, o con cualquier combinación de los parámetros variables –aceptando tres parámetros fijos y dos variables, no tenemos como decir que no queremos exactamente cuatro, sólo tres y cinco parámetros–. La solución está en comprobar el número de parámetros en el procedimiento llamado, por ejemplo, incluyendo unas lineas como:

```
# Comprobamos el número de parámetros
if (3 == @_)
        {
        ( $param1,$param2,$param3)=@_;
        $param4="valorpordefecto";
        }
        elsif(4==@_)
        {
        ( $param4, $param1,$param2,$param3)=@_;
        }
        else
        {
        die "Número de parámetros incorrecto.";
};
```

Esta aproximación tiene un problema: el mensaje de error aparece con un formato bueno para el que desarrolla la rutina llamada, y no la llamante, con lo que es muy difícil trazar el error, sobre todo en programas grandes. La solución

es usar el módulo `Carp` -que ya hemos estudiado-, que nos permite generar un error con más información. El ejemplo anterior se puede convertir en:

```
# Comprobamos el número de parámetros
if (3 == @_)
        {
        ( $param1, $param2, $param3)=@_;
        $param4="valorpordefecto";
        }
        elsif(4==@_)
        {
        ($param4,  $param1,$param2, $param3)=@_;
        }
        else
        {

         confess "Número de parámetros incorrecto.";
};
```

obteniendo en caso de error una salida como:

```
Número de parámetros incorrecto. at pruebaparam.pl line 23
        main::prueba(1,2,3,4,5) called at pruebaparam.pl
           line 34
```

7.11. Comprobando tipos de parámetros

Así como comprobamos el número de los parámetros, también es una buena idea comprobar su tipo, particularmente en el caso de los parámetros numéricos. Esto es muy fácil usando las nunca lo suficientemente valoradas expresiones regulares. Por ejemplo, si una función necesita que un parámetro sea entero, hacemos:

```
unless($edad =~ /^[0-9]+$/) confess "Esperado un número
   entero como segundo parámetro, encontrado ($edad).";
```

El error generado sería algo como:

```
Esperado un número entero como segundo parámetro,
   encontrado (a). at pruebaparam.pl line 25
        main::prueba('a',2,3) called at pruebaparam.pl
           line 32
```

También debemos tener cuidado con los valores devueltos en caso de error no crítico. Mucha gente devuelve un `0` o un `-1` si hay error, pero esto no es una buena idea. Lo mejor suele ser avisar que algo no funciona correctamente con un cluck, y devolver un `undef`:

```
cluck "RUTINA␣piolín:␣Ojo.␣No␣he␣conseguido␣recuperar␣el␣
   dato.␣¿Existe?\n";
return (undef);
```

7.12. Uso de asertos en códigos Perl

Los asertos son predicados que comprueban que se cumplen determinadas condiciones que se suponen ciertas dentro de un programa. Si hay algún error lógico, estas cosas que se suponen ciertas no lo serán, lo que dará lugar a errores que se manifiestan en lugares y en momentos distintos a los que se generaron. Por ello, el aserto no soluciona un error, apenas hace que se manifieste antes, por lo que será más fácil comprobarlo.

Los asertos más conocidos son:

- Comprobar los parámetros de una función al inicio de esta. Ya he comentado en este tema como hacerlo.
- Precondición. Es un conjunto de cosas que deben ser ciertas antes de ejecutar un trozo de código tales que, si no se diesen, el trozo de código puede no generar una solución correcta.
- Postcondición. Es un conjunto de cosas que deben ser ciertas después de ejecutar un trozo de código tales que, si no se diesen, el trozo de código no habría generado una solución correcta.
- Invariante. Aserto que se debe cumplir durante la ejecución de todo un bucle. Ojo con este aserto; aunque es muy útil para localizar errores, en

producción puede castigar el rendimiento.

Uno de los mitos más comúnmente difundidos es que los asertos hacen el código extremadamente ineficiente, y que por ello deben retirarse en la versión de producción. En mi experiencia particular, un aserto inteligentemente colocado no tiene por qué hacer el programa mucho más lento ni ineficiente, y sí facilita la verificación. El truco para poder poner asertos sin afectar al rendimiento es ponerlo fuera de los bucles más internos del código.

7.13. Un libro de estilo de Perl

Los libros de estilo determinan la estética de la programación: como y donde se ponen las variables, donde y como se indenta, y como se definen los nombres de las variables.

En Perl tenemos la fortuna de que Larry Wall –creador de Perl– nos ha legado un libro de estilo, que podemos consultar con `man perlstyle`. Reproduzco aquí un resumen del estilo de programación propuesto por el creador:

- Los nombres de variables serán con nombres claros, que permitan hacerse una idea de lo que contienen, separando las palabras en el nombre por lineas de subrayado. `$mi_variable_complicada`, por ejemplo, es preferible a `$miVariableComplicada`, según Larry Wall.
- Las variables se ponen con todas en mayúsculas si las usaremos como constantes: `$ESTO_ES_UNA_CONSTANTE`,; o con las primeras letras de cada palabra en mayúsculas si son estáticas o globales al paquete; por ejemplo, `$Esto_Es_Una_Variable_Global`, y con todas las letras en minúsculas si corresponde a una variable con ámbito definido con `my` o con `local`; por ejemplo: `$esto_es_una_variable_local`.
- Los nombres de los métodos y las funciones serán todos en minúscula, separando las palabras por _. Por ejemplo, `mi_funcion`.
- Los nombres de paquetes tienen como primera letra una letra mayúscula.
- Las indentaciones son de cuatro espacios[7].

[7] Anotación mía: herejes desalmados reclaman y enseñan que las indentaciones son de ocho espacios. Estos desalmados, que probablemente cuentan con tres monitores panorámicos, serán castigados por el martillo de herejes en revisiones de código y arderán en departamento de programación en Java de una cárnica del PTA.

- Las llaves se abren en la misma linea que su palabra clave. Si no es posible, en la linea consecutiva.
- Hay que poner un espacio antes de la llave de apertura de un bloque de más de una linea.
- Los bloques de una única linea se pueden poner en una única linea –es decir, no debemos ser pedantes en indentar bloques que no son necesarios de indentar–.
- No debe haber espacio antes del punto y coma.
- El punto y coma se omite en bloques de una linea cortos.
- Siempre que podamos, debemos poner un espacio antes y después de cada operador y de cada subíndice entre corchetes.
- Debe haber lineas en blanco entre zonas del código que hagan cosas distintas.
- No debemos poner espacios entre un nombre de función y el paréntesis de apertura de sus operandos.
- Después de cada coma debemos poner un espacio.
- Si debemos escribir una instrucción en más de una linea, debemos pasar a la linea siguiente después de un operador, salvo los operadores lógicos en expresiones lógicas, en cuyo caso pasaremos a la linea siguiente antes del operador.
- Después del último paréntesis que cierre un paréntesis abierto en la linea actual, ponemos un espacio que lo separará de los paréntesis de cierre que cierren paréntesis anteriores a la linea actual. Así el espacio nos permitirá distinguir que está cerrando paréntesis anteriores.
- Aquellas cosas que guarden relación en lineas verticales –parámetros de la misma función, operadores de la misma expresión– en instrucciones de mas de una linea deben estar alineadas.
- No seamos pedantes. Eliminemos aquella puntuación que no afecte a la claridad del código.

Esta es la opinión de Larry Wall. A lo mejor no estoy de acuerdo con algún punto de estos, pero es su libro de estilo. Ahí queda.

7.14. Actividades

La actividad de este tema será convertir las funciones que desarrollastes para las prácticas anteriores en un módulo.

El módulo deberá contener dos inicializadores de paquete, y dos finalizadores de paquete.

Deberás hacer un programa en Perl externo; que incorpore este módulo, y llame a sus funciones utilizando su espacio de nombres.

Además de esto, el programa debe:

- Declarar una variable de tipo matriz asociativa, que tenga un campo, cuya clave sea tu DNI, y que contenga una lista con los dígitos de tu DNI al revés.
- Declarar una variable de tipo lista, que contenga los dígitos de tu DNI.
- Llamar con la matriz asociativa y las listas antes declaradas a la última de las funciones del módulo solicitadas.
- Finalmente, imprimir en pantalla en resultado.

Este programa no debe contener funciones ni implementarlas; sino llamar a las funciones del módulo que has generado en el paso anterior.

Si sospechas que esto es parecido a lo que te pedí en el tema 5, te equivocas, Manuel: no es algo parecido, te estoy pidiendo exactamente lo mismo. No deberías programar ni una función, ni una sentencia de control, si haces lo que te pido; ya que lo que te estoy pidiendo es que adaptes lo que ya tienes hecho a un módulo, y que la parte que llamaba a las funciones ahora invoque las funciones del módulo.

Para que la tarea sea valorada como apta, el programa debe funcionar bien, hacer lo que te pido, y además **debes usar las funciones del paquete que has desarrollado a partir de las funciones de actividades anteriores.**

De cualquier forma, habiendo llegado hasta aquí, ya estás calificado como apto en el curso; por lo que el objetivo es que aprendas. Utiliza el foro para las

dudas que tengas, que ahora serán numerosas, ya que esto es un proyecto ya de tamaño real, comparable con lo que te enfrentarás en el mundo real.

Dada la naturaleza de esta práctica y de las restantes, la mecánica será distinta. Si estás interesado en la siguiente y no quieres hacer esta, me subes la encuesta –como siempre–. Pero a diferencia de las anteriores, cuando no quieras pelearte más con la práctica, me la subes para que te pueda decir en qué esta fallando, y pasas a la siguiente; aunque te recomiendo que le des algunas vueltas antes de mandármela, para que te sea útil.

Después de haber probado los ejemplos de los apuntes, y de haber intentado el ejercicio, no dudes en utilizar el foro para preguntar todo lo que no entiendas.

Capítulo 8

Programación orientada a objetos en Perl

La programación orientada a objetos es un paradigma de programación que hoy en día está de moda. Y Perl la incorpora. Aunque el objeto de este curso no es enseñar orientación a objetos ni programación orientada a objetos en Perl, sí es importante tener algunas nociones que necesitaremos para usar las bibliotecas de CPAN.

Si los módulos no nos parecen un mecanismo lo suficientemente estructurado para nuestros programas, podemos usar uno de los mecanismos de estructuración de código y de modelado más usados en la actualidad: la programación orientada a objetos.

En Perl la semántica orientada a objetos no es obligatoria. De hecho, se puede mezclar dentro de un programa la programación tradicional y la orientada a objetos, según nos convenga. O incluso se puede programar sin programación orientada a objetos en absoluto.

En este tema vamos a aprender a usar CPAN; pero antes aprenderemos unos rudimentos de programación orientada a objetos en Perl que necesitaremos para poder aprovechar mejor CPAN.

8.1. Creando clases en Perl

La unidad básica dentro de la programación orientada a objetos es la clase. Una clase es un conjunto de funciones –denominados métodos– vinculados con

un conjunto de estructuras de datos –denominados atributos–[1]. Hay una serie de propiedades adicionales que un buen lenguaje orientado a objetos debe tener, como son el polimorfismo y la herencia. No vamos a entrar en detalle en describir cada una de las características de la programación orientada a objetos en sí, ya que esto nos llevaría fuera de los objetivos del curso. Supondremos que el lector ya conoce las bases de la orientación a objetos, y nos limitaremos a analizar lo más estrictamente necesario.

En Perl, la creación de una clase es similar a la creación de un módulo: primero es necesario crear un fichero que tenga el mismo nombre de la clase, pero terminado en `.pm`. Todos los métodos y atributos relacionados con la clase deberán estar contenidos en este fichero. Al principio de este fichero debemos declarar qué clase estamos definiendo, lo que hacemos, como en el caso de los módulos, con la palabra clave `package`.

El siguiente paso será declarar el constructor y, si estimamos oportuno, el destructor.

En muchos de los lenguajes orientados a objetos el constructor suele invocarse con el nombre de la clase, o con `new`. Sin embargo, en Perl esto no es necesariamente así: `new` no es una palabra reservada, y podemos usar el nombre de función que queramos como constructor. Sin embargo, por razones de higiene mental es una buena práctica llamar al constructor `new` -a-unque sin olvidar que en programas escritos por otros puede no llamarse así–.

Los atributos los declaramos dentro del constructor.

Por último, declaramos todos los métodos asociados a la clase, como funciones dentro del texto de nuestra clase.

8.2. Nuestra primera clase en Perl

Supongamos la clase `Punto`. Esta clase `Punto` tiene dos atributos, `coordX` y `coordY`. Comenzaríamos definiendo la clase:

```
package Punto;
```

Después incluimos el `use strict`, que siempre es buena idea:

```
use strict;
```

[1] Sí, se que es más que eso. Pero vamos a sobresimplificar mucho la programación orientada a objetos, para quedarnos solo con lo que necesitamos.

Declaramos el constructor. Como hemos recomendado, lo llamaremos `new`:

```
sub new {
        my $clase=shift;
        my $yomismo= {};
        $yomismo->{coordX} = undef;
        $yomismo->{coordY} = undef;
        bless $yomismo,$clase;
        return $yomismo;

}
```

Lo que hacemos en este constructor es declarar una variable `$yomismo`, inicializarla e inicializar sus atributos –en este caso, a indefinido–. `bless` es una función que convierte el hash que le pasamos como primer parámetro en un objeto de la clase que le pasamos como segundo parámetro. Finalmente, devolvemos la variable `$yomismo`.

El archivo `.pm` donde está contenido el objeto debe terminar con `1;`, como pasa en el caso de los módulos.

Con todo esto, ya hemos definido nuestra primera clase en Perl.

8.3. Definiendo métodos en Perl

Vamos a definir nuestros primeros métodos en Perl. Los métodos en Perl se definen exactamente igual que definíamos las funciones en Perl; sólo debemos recordar que el primer parámetro de la función si es un método de instancia será siempre el objeto sobre el que realizamos la operación; y que el primer parámetro de la función si es un método de clase será una cadena de texto con el nombre de la clase.

Los métodos en Perl pueden ser simultáneamente de clase y de instancia. Para hacer esto, debemos comprobar al inicio de nuestro método si el primer parámetro es una cadena –en cuyo caso se tratará de un método de clase, y dicha cadena corresponderá al nombre de la clase–, o una referencia a un objeto, en cuyo caso se habrá llamado como método de instancia.

En nuestro ejemplo definiremos como métodos a los métodos de proyección de la clase `Punto` definida anteriormente. Los métodos de proyección permi-

ten obtener los atributos del objeto, en lugar de acceder a ellos directamente. El uso de métodos de proyección es muy interesante, ya que permite que el uso y el acceso a los atributos sea independiente de la codificación interna de dichos atributos.

En nuestro caso, por ejemplo, definiremos dos métodos de proyección: `coordenadaX` y `coordenadaY`; que corresponden a las coordenadas cartesianas del punto. Dichos métodos serán:

```
sub coordenadaX {
      my $yomismo = shift;
      if (@_) {$yomismo->{CoordX}= shift}
      return $yomismo->{CoordX};
      }
sub coordenadaY {
      my $yomismo = shift;
      if (@_) {$yomismo->{CoordY}= shift}
      return $yomismo->{CoordY};
      }
```

Estos métodos siempre devuelven el valor de la coordenada que les corresponde. Si reciben un único parámetro, este parámetro es el objeto. Si reciben más de uno, suponen que el segundo parámetro será el nuevo valor de la coordenada, lo almacenan en el atributo y lo devuelven.

Podemos usar la clase que hemos creado con las sentencias anteriores con un programa tal y como:

```
use strict;
use warnings;

use Punto;

my $mipunto=Punto->new;

$mipunto->coordenadaX(12);
$mipunto->coordenadaY(2);
print "(".$mipunto->coordenadaX.",".$mipunto->coordenadaY.
   ")\n";
```

Ojo, Manuel: realmente con lo único con lo que quiero que te quedes es con este programa: cómo llamas al constructor, y cómo llamas a los métodos de clase. El resto te lo explico por culturilla, porque está fuera de los objetivos del curso. Y es importante que sepas cómo llamar al constructor y a los métodos de clase porque lo harás cuando utilices CPAN.

8.4. Destructores en Perl

El destructor en Perl tiene algunas características especiales que debo mencionar. Como recordamos, en Perl no tenemos una instrucción para liberar memoria, ya que el lenguaje tiene un recolector de basura, y las variables se liberan cuando no hay ninguna referencia apuntando hacia ellas. Del mismo modo, los objetos en Perl también usan este recolector de basura, por lo que no es necesario llamar explícitamente al destructor, sino que Perl llama al destructor cuando el objeto no tiene mas referencias –tal y como hacen lenguajes como Java–. De hecho, en muchos casos ni siquiera es necesario hacer el destructor de nuestras clases.

El hecho de que el destructor sea llamado automáticamente fuerza a que dicho destructor deba tener siempre el mismo nombre, para que el interprete pueda localizarlo. El nombre que debe tener el destructor de cualquier clase en Perl es `DESTROY`; y está en mayúsculas todo el nombre ,como corresponde a todas las funciones que el intérprete ejecuta automáticamente.

Como Perl tiene recolector automático de basura –y este funciona realmente bien–, no es necesario en la mayor parte de los casos definir el destructor de una clase. El sistema ya se encargará de liberar los atributos del objeto correctamente. Sin embargo, hay tres casos en los que nos puede ser interesante programar un destructor. El primer caso es si queremos generar algún efecto lateral sobre otro objeto cuando el objeto sea destruido. Esto suele ser una mala idea, ya que no sabemos a priori cuando se va a ejecutar el destructor –solo sabemos que será ejecutado alguna vez después de no tener el objeto más referencias–.

El segundo caso de uso del destructor es cuando tenemos estructuras de datos en el objeto que puedan tener ciclos internos. Es el caso de los grafos o de las listas circulares. En estas estructuras un recolector de basuras tendrá problemas para limpiar la estructura, ya que borrando a partir de los nodos no referenciados nos encontraremos con que los ciclos no pueden ser correctamente

eliminados. Por ello, es común en las clases que usen estructuras de datos con ciclos romper dichos ciclos mediante asignaciones a `undef` de algunos de sus elementos del ciclo en el destructor.

Un tercer caso de uso del destructor es para liberar los recursos del sistema operativo que Perl no pueda liberar automáticamente. Destacamos los recursos del sistema de intercomunicación entre procesos System V, que deben ser liberados explícitamente por la aplicación antes de morir esta; como Perl no lleva control de ello, deberían ser los destructores los que limpiasen los recursos reservados por sus objetos.

8.5. La herencia en Perl

No podemos hablar de orientación a objetos si no disponemos de la posibilidad de heredar de unas clases a la hora de crear las otras. Y Perl dispone de herencia. De hecho, Perl tiene un mecanismo de herencia múltiple muy flexible.

Toda clase en Perl hereda, al menos, de la clase `UNIVERSAL`. Y puede heredar de tantas clases más como se quiera.

La herencia en Perl se hace a través de la matriz `@ISA`, definida dentro de cada clase. Esta matriz contiene una lista compuesta por elementos, donde cada elemento corresponde a una de las clases antecesoras de la clase que estamos definiendo. Por ejemplo, si queremos definir la clase `PuntoColorido`, que desciende de la clase `Punto`, hacemos:

```
@ISA=qw(Punto);
```

dentro de la clase `PuntoColorido`. Por otro lado, si queremos que la clase `Capitulo` descienda de `Libro` y de `Articulo`, hacemos:

```
@ISA=qw(Libro Articulo);
```

dentro de la clase `Capítulo`.

A la hora de buscar un método o un atributo en una clase se busca en primer lugar dentro de dicha clase. Si no está, se busca en las clases base –es decir, en las definidas en el `@ISA`–. Si no, busca en la clase `UNIVERSAL`. Por último, si es un método y aun no lo ha encontrado, busca el método definido como `AUTOLOAD`, asignando la variable global `$AUTOLOAD` al método que se estaba buscando, y pasando a ejecutar dicho método `AUTOLOAD`.

En Perl no heredamos atributos, apenas los métodos. Pero podemos heredar también los atributos del padre llamando al constructor del ascendiente desde el constructor del descendiente; lo que hacemos con:

```
SUPER::nombreconstructorpadre;
```

En Perl no podemos esconder un método a un descendiente. De hecho, ni siquiera podemos esconder los métodos y los atributos a las clases y objetos clientes.

8.6. La función `bless`

`bless` es una función muy importante: es la responsable de convertir el hash en un objeto. Debemos invocarla siempre en el constructor de nuestro objeto, con la sintaxis:

```
bless objeto clase;
```

o hacer simplemente:

```
bless objeto;
```

Si estamos dentro de la clase que queremos asignar.

Recordemos que `bless` convierte un hash en un objeto. Antes de `bless`, tenemos un hash. Después, un objeto. Un error frecuente es olvidarnos del bless, y después sorprendernos de que no funciona la orientación a objetos en Perl.

Es conveniente llamar a `bless` con sus dos parámetros: esto nos permitirá heredar el constructor del padre y, por lo tanto, también sus atributos.

8.7. El uso de `SUPER`

Podemos emplear `SUPER` en cualquier momento como si se tratara de una clase. `SUPER` referenciará a la clase padre. En la práctica, significa que el mecanismo de búsqueda de atributo o método se salta la clase actual, y comienza la búsqueda directamente en la matriz `@ISA`.

8.8. CPAN

Ahora entramos en lo realmente útil, y la razón por la que he explicado lo de los objetos en este tema.

CPAN –Comprehensive Perl Archive Network– es un repositorio de bibliotecas para Perl de gran calidad, que contiene más de cien mil módulos. Siempre que trabajes en Perl y te enfrentes a un problema nuevo, **siempre**, lo primero que debes hacer es entrar en CPAN y mirar si ya está programado, para no reinventar la rueda. Y creeme, Manuel: seguro que está implementado en CPAN. De forma que el 60 % de programar en Perl es saber hacer hacks rápidos, y el 40 % restante es buscar en CPAN lo que tienes que hacer, e importarlo.

De los cien mil módulos de CPAN, la práctica totalidad de ellos se importan como paquetes, y se utilizan como clases. Por ello, usar un módulo de CPAN es casi siempre llamar al constructor de una clase importada, y lanzar métodos del objeto que te ha creado el constructor. Así de fácil.

No es solo que tengamos una cantidad ingente de código ya hecho en CPAN. Es que además en la web de CPAN encontramos documentación extensiva de los módulos, con decenas de ejemplos. Es muy frecuente –y te aviso, Manuel, que lo sé por experiencia directa– que aquello que queremos hacer, y teóricamente serán meses de trabajo intensivo, se limite a buscar en CPAN, instalar el módulo, y tocar uno de los ejemplos para ajustarlo a nuestro problema. Meses de trabajo resueltos en minutos. Eso es la magia de Perl con CPAN. Y por eso Larry Wall dice que Perl es el lenguaje de los vagos.

Trabajar con CPAN tiene tres problemas, por lo tanto: buscar qué módulo implementa lo que queremos, instalarlo, y utilizarlo. Respecto a utilizar módulos, los tres últimos temas del curso van sobre ello, con tres módulos que vas a utilizar con seguridad: el módulo para procesar correo, el módulo para navegar por la web, y el módulo de bases de datos. Los otros dos problemas los trataremos en este curso.

8.9. Instalando CPAN

Si tenemos Perl instalado, el instalador de CPAN ya viene “de serie”; aunque a veces tengamos que hacer algún paso más.

En Perl para Windows suele venir un script, denominado `cpan`, que podemos ejecutar, para lanzar desde dicho script los comandos de instalación de los

módulos que queramos. Si no tenemos el script disponible, debemos instalar desde el sistema de gestión de paquetes del entorno de Perl que hemos cogido o CPAN, o XS CPAN –toma nombres distintos en las distintas distribuciones de Perl para Windows–.

En Mac OS X Perl viene preinstalado, pero para lanzar CPAN debemos instalar el paquete "developer" que acompaña al DVD de instalación de Mac OS X. Realmente nos bastará con la opción"Unix tools"; pero viene bien instalarlo completo.

En los Unix, suele acompañar a Perl, por lo que no tendremos que hacer mucho más.

A veces, podemos lanzar CPAN desde línea de comandos utilizando el script:

```
cpan
```

Otras veces, debemos lanzar directamente el módulo desde Perl, haciendo:

```
perl -MCPAN -e shell
```

En cualquiera de los sistemas operativos comentados, si realizas los pasos que comento, te funcionará o el script, o funcionará lanzarlo directamente desde Perl, o funcionarán ambos métodos.

8.10. Buscando en CPAN

Tenemos la web de CPAN en:

http://www.cpan.org/

La opción de búsqueda la tenemos arriba, a la derecha.

Ponemos las palabras de lo que queremos buscar –por ejemplo, si queremos un módulo para conectarnos con Amazon, ponemos "Amazon"–, y le damos a "CPAN Search". Nos aparecen los módulos soportados por CPAN relativos a esta tecnología –en este caso, solo 671 módulos–. Seleccionamos qué queremos -interfaz con el EC2 para computación en la nube, el de Amazon S3, o el módulo DBI para interactuar con el sistema de comercio electrónico de Amazon, por poner algunos ejemplos-. Entrando en el módulo que queremos, tendremos abundantisima documentación, muchos ejemplos de uso, y el nombre del módulo. Este nombre es el que necesitaremos para instalarlo.

8.11. Instalando módulos en CPAN

Una vez que tenemos el nombre del módulo, invocamos al script de CPAN:

```
cpan
```

o directamente:

```
perl -MCPAN -e shell
```

para instalar un módulo, hacemos:

```
install nombremódulo
```

por ejemplo, si queremos un interfaz para trabajar con comodidad y facilidad con Amazon desde Perl, buscamos en CPAN por "Amazon", y encontramos el módulo `Net::Amazon`, documentado en la página:

http://search.cpan.org/~boumenot/Net-Amazon-0.59/lib/Net/Amazon.pm

Lo instalamos haciendo desde CPAN:

```
install Net::Amazon
```

es importante que sepamos que si tenemos una instalación "virginal" de Perl, los primeros módulos que instalemos a través de CPAN instalan muchísimas dependencias; por lo que el proceso puede demorar bastante. Nos limitamos a decir que `yes` a todo lo que nos pregunte –pulsar `enter` repetidas veces suele ser suficiente–, y ya está.

8.12. Actividades

Supongamos que tenemos que conectamos con una base de datos FluidDB para web semántica, en computación de cloud. Nuestro jefe nos da la cantidad de tiempo estándar y razonable para hacer algo de lo que no hemos oído hablar en nuestra vida, y desconocemos por completo: una tarde. En una tarde tenemos que ser capaces de conectarnos a una base de datos Fluidinfo, autentificarnos correctamente, y obtener la información básica: obtener un objeto por identificador, obtener un objeto por descripción, y buscar objetos.

Tenemos dos opciones: buscar la documentación del protocolo, estudiar la arquitectura de Fluidinfo, e implementar todo lo que necesites, o buscar otra opción.

Para conseguir tu objetivo, tienes dos opciones. Cualquiera de las dos la doy por buena:

- O implementarme desde cero todo: protocolo, enlace con arquitectura de Fluidinfo, y API. Por supuesto, verificado. Si optas por esta opción, me tienes que mandar todo el código fuente, las rutinas de prueba de que funciona lo que haces, y la documentación apropiada.

- O encontrar otra forma de conectar un script Perl con una base de datos Fluidinfo. En caso de que optes por esta solución, me basta para calificarte la práctica como APTA con que me mandes un fichero de texto con los pasos que has dado para conseguir soportar Fluidinfo, y un volcado de lo que Perl te ha generado para conseguir esa compatibilidad. Como la salida será de texto, un copia y pega de la consola será suficiente.

Esta práctica se puede hacer en diez minutos –de los cuales nueve los invertirás en el copia y pega de la salida, ya que si tienes una instalación "virginal" de Perl la salida de lo que tienes que hacer será grande–, o pueden ser varios meses de trabajo. Incluso años de trabajo, si optas por la elección erronea.

Si optas por la forma de hacer esto en diez minutos no espero que me mandes código en Perl, ya que las rutinas funcionan.

En cualquier caso, si optas por la opción correcta, has entendido la filosofía detrás de Perl, y lo que he intentado transmitir durante el curso.

Capítulo 9

Interactuando con aplicaciones Unix desde Perl

Muchas cosas se hacen con más facilidad desde línea de comandos, que desde programación. En Unix, uno de los aspectos filosóficos fundamentales es tener muchas herramientas que hagan una cosa, y solo una cosa: pero que la hagan bien. Si hemos administrado sistemas ya, estamos acostumbrados con estas herramientas de línea de comandos, y no querremos prescindir de ellas.

Y como podemos sospechar, sí; en Perl tendremos como lanzar cualquier herramienta de linea de comandos. Y como no podría ser de otra forma, en Perl –como siempre– hay más de una forma de hacerlo.

De hecho, vamos a ver cuatro formas de hacerlo. Cada una tiene unas diferencias sutiles, pero importantes. Comencemos por el primer método.

9.1. La función `system`

La primera forma que aprenderemos para llamar programas externos desde Perl es la función `system`. Esta función se caracteriza porque lanza el programa especificado como parámetro sin redireccionar sin capturar su salida.

Puede tener como parámetros un escalar, o un vector. Si tiene como parámetros un escalar, llama al escalar sin más; si tiene como parámetros un vector, el programa lanzado será el primer elemento del vector, y los restantes elementos del vector serán los parámetros que se le pasarán al programa llamado.

En ambos casos, pues, podemos pasar parámetros: en el primero, formarán parte de la cadena escalar llamada. En el segundo, serán elementos posteriores al primero.

Por poner un ejemplo, para ver por extenso todos los archivos de un directorio con sus atributos, haremos:

```
system("ls -la");
```

lo que es lo mismo que hacer:

```
system("ls","-la");
```

Esta función asignará a la variable comodín `$?` el valor de salida del comando lanzado, tal y como lo devolvería si lo llamáramos en C empleando la llamada `wait`. Podemos obtener el valor de retorno Unix tomando el valor de retorno `wait` y rotándolo a la derecha ocho posiciones; es decir, el valor de retorno Unix sería `$? >> 8`.

En caso de que `$?` sea -1 significa que ha habido un error. Podemos saber la razón del fallo a través de la variable comodín `$!`. Por ejemplo, podemos hacer:

```
system("micomando", "argumento1","argumento2");
 if ( $? == -1 )
 {
   print "El comando ha fallado; razón: $!\n";
 }
 else
 {
   printf "El comando ha funcionado, el código devuelto ha
      sido %d", $? >> 8;
 }
```

9.2. La función `exec`

`exec` es similar a las llamadas `exec` de POSIX: se llama a un programa externo, pero el llamante desaparece, y el llamado pasa a tener el PID del llamante; lo que se hace en C después de un `fork` para crear un hijo.

Como en el caso de `system`, esta llamada puede tener como parámetros un

escalar, o un vector. Si tiene como parámetros un escalar, llama al escalar sin más; si tiene como parámetros un vector, el programa lanzado será el primer elemento del vector, y los restantes elementos del vector serán los parámetros que se le pasarán al programa llamado.

En ambos casos, pues, podemos pasar parámetros: en el primero, formarán parte de la cadena escalar llamada. En el segundo, serán elementos posteriores al primero.

Si falla, el error por el que ha fallado estará en la variable comodín `$!`. Si la llamada funciona, no recogeremos el código devuelto en ningún lado, porque el programa llamante habrá dejado de existir. Por ejemplo, es común usar `exec` de la siguiente forma:

```
exec("micomando", "argumento1","argumento2");
print "El comando ha fallado; razón: $!\n";
```

9.3. El operador comillas invertidas `

Hasta aquí estamos pudiendo hacer lo que ya podíamos hacer con facilidad en C o BASH. Ahora, sin embargo, es cuando esto se va a poner divertido.

Un operador interesante es el operador comillas invertidas `. Este operador lanza el código que encontramos entre comillas simples; pero, a diferencia de los comandos anteriores, devuelve algo: la salida del programa. Es decir, que lo que genera el programa ejecutado lo introducirá en la variable a la que estemos asignando el resultado de dicho operador. Por ejemplo, si hacemos:

```
$listado= `ls -la /home`;
```

almacenará en la variable `$listado` una cadena que consiste en la salida del comando `-la /home`, es decir, en el contenido completo del directorio `/home`, con los atributos de cada fichero.

Este comando puede ser invocado en contexto escalar –como ha sido este ejemplo–, o en contexto vectorial. En contexto escalar, la variable asignada contendrá una cadena con toda la salida; lo que corresponderá probablemente con varias líneas. Pero en contexto vectorial la salida de este operador es aún más interesante, ya que generará un vector donde cada línea de salida del programa se almacena en un elemento del vector. Esto, añadiéndole `foreach` iterando el

vector, más expresiones regulares, y nos podemos hacer una idea de los usos prácticos de el operador de comillas invertidas.

Un ejemplo de uso del operador de comillas invertidas en contexto vectorial:

```
@listados= `ls -la /home`;

foreach $listado (@listados)
        {
                printf("Encontrado␣$listado\n");
        };
```

En caso de que el comando especificado por el operador de comillas invertidas no se pueda lanzar por alguna razón Perl devolverá la cadena vacía; o la lista sin elementos –según estemos, respectivamente, en contexto escalar o vectorial–. También almacenará en la variable comodín `$!` el error por el que no se pudo lanzar el comando especificado.

El operador de comillas invertidas también se puede expresar mediante `qx//`. Por ejemplo, podemos reescribir los ejemplos anteriores como:

```
$listado= qx/ls -la \/home/;
```

y

```
@listados= qx/ls -la \/home/;

foreach $listado (@listados)
        {
                printf("Encontrado␣$listado\n");
        };
```

siendo exactamente iguales en ambos casos. Personalmente lo encuentro menos operativo, ya que la barra de directorio tendremos que escaparla; pero aunque no lo usemos, podemos encontrarlo en algún script Perl de otra persona.

Es importante que recordemos que este operador almacenará en la variable la salida estándar, no la salida de error. Si queremos almacenar también la salida de error tenemos dos formas; la primera, redireccionar la salida de error a la estándar, haciendo:

```
 $variable = `comando 2>&1`;
```

La segunda usar una función aún más potente: la función `open`.

9.4. La función `open`

La función `open` es la función más potente que tenemos para llamar aplicaciones; nos permitirá tanto capturar la salida de un comando, como alimentar un comando con datos generados desde el propio script de Perl. Dicho de otra forma, podemos tanto "alimentar" la entrada del programa, como leer su salida.

La base de `open` es que, a diferencia de las demás funciones que hemos visto, vincula la entrada o la salida estándar de la ejecución de un archivo a un descriptor de ficheros, por lo que podemos controlar su entrada o recoger su salida.

La sintaxis de `open` es:

```
open(DESCRIPTOR,comando)
```

Donde `DESCRIPTOR` es el descriptor de ficheros desde el que operaremos, y `comando` un comando Unix, con todos sus parámetros. La cadena `comando` contendrá una barra al principio de la cadena si queremos redireccionar la entrada del comando al descriptor de ficheros indicado, y contendrá una barra al final de la cadena si lo que queremos es redireccionar la salida del comando al descriptor de ficheros indicado.

Por ejemplo, para analizar los procesos en ejecución, lo podemos hacer utilizando `open` y redireccionando la entrada:

```
open(PSAX,"ps␣-ax␣|")
       or die "No␣se␣pudo␣lanzar␣ps:␣$!\n";

while ( <PSAX> )
{
  # Aquí introducimos las operaciones que haremos sobre
  # cada línea del fichero
}
```

También podemos redireccionar la salida. Por ejemplo, para mandar un correo directamente desde Perl en Unix, podemos hacer:

```
open(MAILER, '|␣mail␣irbis@orcero.org')
```

```
        or die "No se pudo lanzar el mailer: $!\n";

# Mandamos el sbj;
 print MAILER "Histórico de lo que está pasando";

# Generamos y mandamos el cuerpo;
$loqueestapasando=`ps afx`;
print MAILER $loqueestapasando;

# Cerramos el mensaje
print MAILER  ".\n";
```

O, si queremos grabar los datos comprimidos en disco en formato `bzip2`:

```
 open(HISTORICOCOMP, " | bunzip2 -9vvzc > datos.bz2")
        or die "No se pudo abrir el histórico comprimido:
           $!\n";
```

Y lo que grabemos en `HISTORICOCOMP` con `print`, se grabará comprimido.

Si queremos lanzar un ejecutable, y controlar a la vez su entrada y su salida, podemos utilizar la llamada `open2`; en la que ahora el primer parámetro será un handler de fichero, y el segundo será otro handler de fichero. El primero corresponderá a la salida estándar, y el segundo a la entrada estándar.

La sintaxis de `open2` es:

```
open2(DESCRIPTOR_SALIDA,DESCRIPTOR_ENTRADA,comando)
```

Donde `DESCRIPTOR_SALIDA` es el descriptor de ficheros del que leeremos la salida estándar del proceso, `DESCRIPTOR_ENTRADA` el descriptor de ficheros tal que lo que escribamos en él se redireccionará a la entrada estándar del proceso, y `comando` un comando Unix, con todos sus parámetros. Es importante que recordemos que debemos incorporar el paquete que contiene `open2`; por lo que debemos poner en nuestro programa la línea:

```
use IPC::Open2;
```

Antes de usar `open2`.

Por ejemplo, para resolver un sistema de doce ecuaciones con doce incógnitas en Perl haremos:

```
use IPC::Open2;

open2(SALOCTAVE,ENTOCTAVE,"octave")
        or die "Fallo al lanzar octave: $!\n";


print ENTOCTAVE
        "a=[7,-7,1,0,-13,2;9,12,1,-1,1,-1;1,2,-3,1,1,1;"
        ."0,0,1,-2,-3,-4;-1,1,-1,1,-1,1;3,4,1,2,1,1]\n";


print ENTOCTAVE "b=[5;12;1;2;3;-1]\n";


print ENTOCTAVE "a\\b\n";

while ( <SALOCTAVE> )
        {
                print $_;  # Aquí procesamos la salida

        };
```

Generando la salida:

```
GNU Octave, version 3.0.1
Copyright (C) 2008 John W. Eaton and others.
This is free software; see the source code for copying
   conditions.
There is ABSOLUTELY NO WARRANTY; not even for
   MERCHANTIBILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type `
   warranty'.

Octave was configured for "i586-mandriva-linux-gnu".
```

```
Additional information about Octave is available at http
   ://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/help-
   wanted.html

Report bugs to <bug@octave.org> (but first, please read
http://www.octave.org/bugs.html to learn how to write a
   helpful report).

For information about changes from previous versions, type
    'news'.


a =

     7    -7     1     0   -13     2
     9    12     1    -1     1    -1
     1     2    -3     1     1     1
     0     0     1    -2    -3    -4
    -1     1    -1     1    -1     1
     3     4     1     2     1     1

b =

     5
    12
     1
     2
     3
    -1

ans =
```

```
-0.69968
 1.58442
-0.26416
-2.71284
-1.35669
 1.80790
```

Si queremos capturar también la salida de error, emplearemos la función `open3`; que permite redireccionar a descriptores de ficheros la entrada estándar, la salida estándar, y la salida de error. La sintaxis de `open3` es:

```
open3(DESCRIPTOR_ENTRADA,DESCRIPTOR_SALIDA,
   DESCRIPTOR_ERROR,comando)
```

Donde el primer parámetro `DESCRIPTOR_ENTRADA` será el descriptor de ficheros tal que lo que escribamos en él se redireccionará a la entrada estándar del proceso, `DESCRIPTOR_SALIDA` es el descriptor de ficheros del que leeremos la salida estándar del proceso, `DESCRIPTOR_ERROR` es el descriptor de ficheros del que leeremos la salida de error del proceso, y `comando` un comando Unix, con todos sus parámetros. Es importante que recordemos que debemos incorporar el paquete que contiene `open3`; por lo que debemos poner en nuestro programa la línea:

```
use IPC::Open3;
```

Antes de usar `open3`.

Por ejemplo, para usar `open3` haremos:

```
use IPC::Open3;

my $PID=open3(ENTOCTAVE,SALOCTAVE,ERROCTAVE,"octave")
        or die "Fallo al lanzar octave: $!\n";


print ENTOCTAVE
        "a=[7,-7,1,0,-13,2;9,12,1,-1,1,-1;1,2,-3,1,1,1;"
        ."0,0,1,-2,-3,-4;-1,1,-1,1,-1,1;3,4,1,2,1,1]\n";


print ENTOCTAVE "b=[5;12;1;2;3;-1]\n";


print ENTOCTAVE "a\\b\n";

while ( <SALOCTAVE> )
        {
                print $_;   # Aquí procesamos la salida

        };

# La salida de error la tenemos en ERROCTAVE
```

Es muy importante que nos percatemos que **el orden de los argumentos es distinto en** `open2` **que en** `open3`.

En ambos casos podemos controlar la terminación del proceso llamado; esto lo haremos con:

```
waitpid( $pid, 0 );
my $status_salida_hijo = $? >> 8;
```

Este comando `waitpid` establece una barrera, por la que el script en Perl no pasará de `waitpid` hasta que no haya muerto el proceso lanzado. Es muy útil para esperar a que termine el hijo lanzado, antes de dar por terminada la

operación.

Si probamos esperar a terminar el proceso hijo con `waitpid` contra el ejemplo de `octave`, debemos recordar que para salir de octave tenemos que mandarle un:

```
print ENTOCTAVE "quit\n";
```

por lo que hasta que no mandemos el `quit` desde el llamante de Octave, octave no terminará, y el llamante no pasará la barrera del `waitpid`.

Hasta aquí, la artillería ligera. Si queremos artillería pesada, podemos usar el paquete `IPC::Run`, extremadamente más potente, y que está documentado en:

http://search.cpan.org/~toddr/IPC-Run-0.90/lib/IPC/Run.pm

Es importante que te vayas acostumbrando a usar CPAN. A partir de ahora, y según llegue el curso hasta el final, iré referenciando más CPAN; con objeto de "quitar el andamiaje" y pienses y trabajes como se hace en Perl.

9.5. Enriqueciendo lo anterior: variables que contienen programas

Si nos damos cuenta de los ejemplos que he puesto, en todo momento lo que hemos pasado como programas han sido realmente cadenas de caracteres.

En el momento que tanto programas que lanzamos como parámetros son cadenas de caracteres, podemos modificarlos sobre la marcha; lo que significa que podemos hacer cosas realmente más complicadas. Por ejemplo, podemos obtener un listado de ficheros con el comando `find` o con el comando `ls`. Los podemos recoger en una variable en contexto vectorial, con objeto de que cada fichero esté en una línea distinta. Podemos en un bucle `foreach` procesar fichero a fichero. Podemos, mediante expresiones regulares, generar el nombre de un fichero destino a partir de un fichero fuente. Finalmente, podemos realizar una operación dentro del bucle `foreach` teniendo como fuente cada línea extraída con las comillas invertidas, como destino el generado vía expresión regular sobre el nombre de la fuente, y analizar dicha operación con una expresión regular para saber si ha tenido éxito o no; y si no ha tenido éxito, según el error, lanzar otro programa. Esto parece complicado, pero no lo es en Perl –y será con lo que te tendrás que pelear en tu actividad–.

9.6. Actividades

La actividad de este tema consistirá en hacer un filtro de ficheros no duplicados.

Se llamará con los siguientes parámetros:

```
eliminaduplicados fuente1 fuente2 destino
```

Comparará cada fichero que esté dentro del directorio `fuente1` y sus subdirectorios con cada directorio del directorio `fuente2` y sus subdirectorios. Si el fichero del directorio `fuente1` y subdirectorios no están en el directorio `fuente2` y subdirectorios, lo copiará en el directorio destino.

Cuando supondremos que dos ficheros son iguales:

- Ficheros de Word: extrae el texto con `catdoc` en ambos ficheros, y compara los resultados con `diff`.
- Ficheros jpeg: conviertelos a compresión al máximo sin pérdida adicional con `jpegoptim -p`, y compara los resultados con `diff`.
- Ficheros gif: conviertelos a compresión al máximo sin pérdida adicional con `gifsicle -O2 -b`, y compara los resultados con `diff`.
- Ficheros png: conviertelos a compresión al máximo sin pérdida adicional con `optipng -o7`, y compara los resultados con `diff`.

Qué necesitas saber –esto es una pista, por si no sabes suficiente Unix–:

- Puedes obtener todos los ficheros contenidos dentro de un directorio con su ruta completa con el comando `find directorio -type f -print`.
- Puedes obtener el tipo de un fichero con el comando `file fichero`.
- Si necesitas un temporal, créate un subdirectorio “ad hoc” en el “tmp” de tu `$HOME`, y copia el fichero temporal en el para procesarlo.
- Puedes comprarar dos ficheros mediante $diff$. Observa el valor que devuelve.
- Copias ficheros con el comando `cp`.

Esta práctica te será de utilidad administrando sistemas web. Además, con pocas modificaciones la puedes emplear para optimizar sitios web –ya que realmente el preprocesamiento que le damos a las imágenes antes de compararlas es aumentar la compresión al máximo sin pérdida–. Es muy verbosa, parece difícil, pero haciéndola de la forma apropiada es apenas media pantalla de código. No es difícil, pero sí requiere que tengas claro como llamar a un ejecutable desde Perl, cómo obtener los datos de vuelta y como analizar el resultado con expresiones regulares.

En cualquier caso, ya tienes tu certificado, hagas lo que hagas; y en cualquier caso, te mandaré la próxima y tendrás todo el material completo del curso. Pero creo que estos temas, que son optativos y densos, son lo suficientemente interesantes como para que les inviertas algo de tiempo; aunque supera los objetivos del curso, y por ello no es obligatorio.

Invierte en lo que resta del curso el tiempo que creas que merece, sabiendo que ya tienes tu certificado; y profundiza cuanto quieras. Aprovecha la oportunidad de que dispones del foro para hacer preguntas.

Capítulo 10

Utilizando bases de datos desde Perl

Muchos lenguajes tienen un problema grave con el acceso a la base de datos: simplemente, no pueden tenerlo. Otros tienen un mecanismo de acceso a la base de datos distinto para cada base de datos. Otros cuentan con un mecanismo único para cualquier base de datos –sea ODBC, o JDBC–.
Perl opta por la solución más sencilla: es el `DBI`*, que permite mandar sentencias* `SQL` *a la base de datos, y que los mecanismos de inserción de datos, actualización de datos, generación de la consulta, ejecución de la consulta y extracción de resultados sean todo lo independientes de la base de datos que permite SQL. Además, si se puede hacer en SQL, se puede hacer en DBI.*

10.1. La idea detrás de DBI

Casi todos los sistemas gestores de base de datos serios que existen en la actualidad intentan ser compatibles con un estándar de acceso a la base de datos llamado SQL92. De hecho, podemos considerar que casi todos son compatibles en su mayor medida con dicho estándar. Hay incompatibilidades en la creación de tablas y bases de datos, pero en los programas no es común andar creando tablas constantemente, sino andar introduciendo datos en la base de datos y consultando la base de datos.

La idea básica detrás de DBI es utilizar SQL para comunicarse con la base de datos, y abstraer las partes dependientes de la base de datos a través de controladores DBD específicos para cada base de datos.

Cualquier programa que utilice DBI tiene siempre la misma estructura:

- Conexión a la base de datos.
- Operación con la base de datos, u operaciones con la base de datos.
- Desconexión con la base de datos.

Cada operación con una base de datos se compone de las siguientes etapas en DBI:

- Preparamos la sentencia SQL en una cadena de texto.
- Ejecutamos la consulta.
- Extraemos los datos de retorno de la consulta.
- A veces nos puede interesar cerrar la consulta después de su uso.

Vamos a estudiar como hacer todas estas operaciones con DBI.

10.2. Conexión con la base de datos

La conexión a la base de datos se hace con el comando `connect`, que tiene como sintaxis:

```
connect(driver,usuario,clave);
```

Donde `driver` es el controlador DBD de la base de datos con sus parámetros, `usuario` es el usuario que usamos para acceder a la base de datos y `clave` es la clave del usuario que accede a la base de datos.

Este comando devuelve el handler de la base de datos abierta. En Perl, a diferencia de otros lenguajes, podemos tener abiertas tantas bases de datos como queramos. Es importante que almacenemos el handler en una variable, ya que lo necesitaremos para preparar las consultas.

El parámetro `driver` suele tener una estructura tal y como:

```
DBI:driver:parametro1=valor1;parametro2=parametro2
```

Donde `driver` es el nombre del módulo DBD de acceso a la base de datos que queremos usar debe estar instalado en nuestro sistema, si no viene con nuestra distribución podremos bajarlo de CPAN-. Después del nombre del módulo tenemos dos puntos, y una lista de asignaciones de valores a parámetros separados por `;`. Estos parámetros se pasan al módulo DBD indicado para establecer la conexión con la base de datos y las propiedades de conexión. De entre los posibles parámetros, la mayor parte de los módulos DBD necesitan el nombre de la base de datos, la dirección IP de la máquina donde está el sistema gestor de base de datos y el puerto en el que escucha.

Algunas veces la sintaxis de esta cadena es:

```
DBI:driver:basededatos
```

Donde `driver` es el nombre del módulo DBD, como el caso anterior, y `basededatos` el nombre de la base de datos.

El comando `connect` puede tener un tercer parámetro, que es un `hash` con las propiedades de la conexión. Las propiedades más empleadas son `RaiseError`, que hace que si lo ponemos a `1` el CGI aborta si recibe un error de base de datos; y `AutoCommit`, que determina si se cierran las transacciones de forma automática o no vemos más adelante con más detalle el tema de las transacciones.

Si hay un error en la conexión, podemos ver cual ha sido el error en el atributo `errstr` del handler obtenido como valor devuelto por `connect`.

10.3. Preparando la sentencia SQL

Necesitamos tener nuestra sentencia SQL en una cadena, que prepararemos para poder ejecutar. La sentencia de SQL puede ser cualquier tipo de sentencia, que haga cualquier cosa.

La sintaxis de la preparación de una sentencia SQL es:

```
$dbh->prepare($sql);
```

donde `$dbh` es el handler de la base de datos sobre la que queremos lanzar la sentencia SQL, y `$sql` es el SQL que queremos lanzar. En algunos sistemas gestores de base de datos, `prepare` compila la sentencia SQL en el sistema gestor de base de datos. Este comando devuelve el handler de la sentencia SQL compilada en el sistema gestor de base de datos, o de la operación, según el caso.

Si hay un error en la conexión, podemos ver cual ha sido el error en el atributo `errstr` del handler obtenido como valor devuelto por `connect` al crear la base de datos.

10.4. Ejecutando la sentencia SQL

Para ejecutar una sentencia SQL, hacemos:

```
$sentencia->execute();
```

donde `$sentencia` es el handler devuelto por la función `prepare`.

El valor devuelto por `execute` es una matriz con el resultado de la operación. En el caso particular de un `select`, por ejemplo, es una lista de filas que cumplen con las restricciones impuestas.

10.5. Extrayendo los datos de la consulta

Hay varios métodos de extraer los datos de una consulta. El más sencillo es importar todo el resultado de la consulta al espacio de nombres de Perl, lo que podemos hacer con:

```
$consulta->fetchrow_array;
```

donde `$consulta` es la consulta de la cual queremos extraer los datos. Esto nos devuelve una matriz en la que cada elemento corresponde con un registro; podemos recorrer la matriz con un `foreach` para operar registro a registro, u operar con todos los registros haciendo uso de que la los tenemos en una matriz.

También nos puede ser útil el método `rows`, que permite saber el número de registros que nos ha devuelto una consulta en particular.

En el ejemplo adjunto vemos como funcionan tanto `fetchrow_array` como `rows`.

10.6. Cerrando la consulta

Aunque no siempre es necesario, es interesante cerrar la consulta inmediatamente después de que no la necesitemos más porque hemos extraído todos

los datos de ella, por ejemplo-, ya que liberamos recursos del sistema gestor de bases de datos las consultas se pueden reciclar-. En una base de datos con un cliente no se nota el que se cierren o no las consultas, pero si tenemos cinco mil clientes sí.

Para liberar una consulta, hacemos:

```
$consulta->finish;
```

donde `$consulta` es el handler de la consulta.

10.7. Desconexión con la base de datos

Cuando terminemos de usar la base de datos, es una buena idea cerrar el acceso a la base de datos para no desperdiciar recursos. En una base de datos con un cliente no se nota el que liberen las conexiones a base de datos cuando no se necesitan más, pero si tenemos cinco mil clientes sí.

La sintaxis para desconectarse de la base de datos es:

```
$dbh->disconnect;
```

donde `$dbh` es el handler de la base de datos.

10.8. Las transacciones en DBD

Las transacciones en DBD dependen de que la base de datos sobre la que actuamos tenga o no tenga transacciones. Si la base de datos no tiene soporte a transacciones, no hay nada que hacer. Si tiene dicho soporte, las transacciones están plenamente soportadas.

Al abrir una base de datos con `connect`, podemos tener o no activado el `autocommit` de la base de datos por defecto, según su DBI. Por ejemplo, el DBD para PostgreSQL tiene activado el `autocommit` por defecto, y el DBD de Oracle no. Lo recomendable es activar o desactivar el autocommit manualmente, como vemos en el ejemplo, para evitarnos sustos.

Con el `autocommit` activado –valor a 1–, toda operación SQL preparada con `prepare` se interpreta como una transacción, y se ejecuta de forma atómica: o se ejecuta entera, o no se ejecuta nada de ella, pero no se puede ejecutar de forma parcial.

Con el `autocommit` desactivado –valor a 0–, al abrir la base de datos hacemos implícitamente un `BEGIN TRANSACTION`, y en cualquier momento podemos hacer:

```
$dbh->commit();
```

con lo que todos los cambios hechos a la base de datos serán permanentes, y DBD hace un nuevo `BEGIN TRANSACTION` por nosotros, o:

```
$dbh->rollback();
```

con lo que se deshacen todos los cambios hechos desde el último `BEGIN TRANSACTION`, y DBD hace un nuevo `BEGIN TRANSACTION` por nosotros.

En los dos casos, `$dbh` es el handler de una base de datos abierta.

Si tenemos el `autocommit` desactivado y salimos del programa, se desharán todos los cambios en la base de datos realizados por el programa.

Podemos activar solo el soporte transaccional en un pequeño trozo con el `autocommit` activado haciendo:

```
{
    local($dbh->{AutoCommit}) = 0;

    eval {
        #Realiza las operaciones
        $dbh->commit();
         };
 if (my $err = $@) $dbh->rollback();

  }

# Una vez que llega aquí, DBD vuelve a modo autocommit y
   si no ha ejecutado el rollback, hace un commit.
```

Pero es un método poco higiénico, por lo que es recomendable usar el primer método comentado.

10.9. Ejemplo de uso de la base de datos en Perl

Terminaremos con un ejemplo de todo lo comentado.

```
#!/usr/bin/perl

use DBI;

# Variables temporales
my $nombre;
my $clase;
my $fechalogin;
my $horalogin;
my $activo;

# Nos conectamos a la base de datos.
my $dbh = DBI->connect('DBI:Pg:dbname=aplicacion;host
   =10.1.1.104;port=5432',"usuario","clave",{RaiseError
   =>1,AutoCommit=>1});



# Algunas variables temporales
my $userid="manolo";
my $sesionid="nXXX.328201.fsndDDCC.dXCX";
my $login="si";

# Componemos el SQL
my $sql="SELECT␣nombre,clase,fechalogin,horalogin,activo
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣FROM␣login
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣WHERE␣userid=\'$userid\'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣AND␣login=\'$login\'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣AND␣sesionid=\'$sesionid\';";

# Montamos la consulta
my $consulta= $dbh->prepare($sql);
```

```
# Ejecutamos la consulta
$consulta->execute();

# Extraemos los datos
my @filas=$consulta->fetchrow_array;

if (0==$consulta->rows)
          {
          # Aquí el código relativo a que no hemos
             encontrado datos.
           };

# Cerramos la consulta
$consulta->finish;

# Operamos con los registros
foreach (@filas)
         {
         ($nombre,$clase,$fechalogin,$horalogin,$activo)=$_
            ;
         # Operamos con los datos
         };

# Nos desconectamos de la base de datos
$dbh->disconnect;
```

10.10. Concluyendo

Ahora, Manuel, antes de pasar a la actividad echa un ojo a:
http://search.cpan.org/~timb/DBI-1.616/DBI.pm

Verás que hay mucha información, y de calidad excepcional: hasta incluye ejemplos de todo. No te debe sorprender: las clases de CPAN están documentadas de forma tan excepcionalmente buena que lo más sensato cuando tienes una duda es mirar en CPAN. La información actualizada y con ejemplos.

En los siguientes temas vamos a trabajar directamente sobre CPAN y sobre un ejemplo de uso. Es importante de que cuanto antes te acostumbres a la búsqueda en CPAN, ya que será tu fuente de información constante.

10.11. Actividades

Para la actividad de este tema debes hacer cuatro programas distintos.

El primero tomará la práctica que hicistes con anterioridad de análisis de históricos. Crearás una tabla en la base de datos que contenga un campo por cada campo parseado del fichero, y modificarás el programa de análisis de históricos para que almacene los datos en la base de datos. En caso de que haya cualquier problema o error al lanzar la aplicación de análisis de históricos, dicha aplicación terminará; y la base de datos deberá quedar como si no se hubiese lanzado la aplicación en la ejecución que dio lugar a error. Como supondrás, esto impone una restricción sobre la base de datos que puedes utilizar que tendrás que indicar como comentario al inicio de la práctica.

El segundo será un programa que tomará como parámetro de entrada un rango de fechas, y a partir de la información de la tabla de históricos y el rango de fechas, generará por salida estándar un volcado de histórico para las fechas contenidas en dicho rango, en formato Apache –es decir, el mismo formato que parseastes en el analizador de históricos de Apache–.

El tercer programa es modificar el filtro de ficheros no duplicados. Crearás una tabla en la base de datos que contenga un dos campos de ruta, y dos campos de nombre de fichero. Modificarás el filtro para almacenar en la base de datos por cada par de ficheros duplicados una tupla que contenga la ruta de ambos ficheros duplicados, y los nombres de ambos ficheros duplicados. En caso de que el programa falle, se mantendrá en la base de datos los duplicados sobre los que se haya operado ya.

El cuarto programa leerá de línea de comandos un fichero determinado con su ruta, y verificará en la base de datos si ha aparecido un duplicado de dicho fichero en la operación anterior.

Esta práctica se parece bastante al uso que harás con frecuencia de bases de datos. Si tienes soltura con SQL y has realizado las prácticas anteriores, esta práctica es de diez minutos. Las bases de datos SQL son omnipresentes hoy en día, por lo que te recomiendo si no sabes hacerla que busques información sobre SQL, porque lo necesitarás en tu vida laboral con seguridad.

Habiendo llegado aquí, ya estás calificado como apto en el curso; por lo que el objetivo es que aprendas. Utiliza el foro para las dudas que tengas, que ahora serán numerosas, ya que esto es un proyecto ya de tamaño real, comparable con lo que te enfrentarás en el mundo real.

Dada la naturaleza de esta práctica y de las restantes, la mecánica será distinta. Si estás interesado en la siguiente y no quieres hacer esta, me subes la encuesta –como siempre–. Pero a diferencia de las anteriores, no te mandaré el material de la siguiente hasta que retengas un mes esta práctica, o me la concluyas con éxito. Si la terminas y quieres la siguiente, me mandas el resultado. Por otro lado, cuando no quieras pelearte más con la práctica, me la subes para que te pueda decir en qué esta fallando, y pasas a la siguiente; aunque te recomiendo que le des algunas vueltas antes de mandármela, para que te sea útil.

En cualquier caso, ya tienes tu certificado, hagas lo que hagas; y en cualquier caso, te mandaré la próxima y tendrás todo el material completo del curso. Pero creo que estos temas, que son optativos y densos, son lo suficientemente interesantes como para que les inviertas algo de tiempo; aunque supera los objetivos del curso, y por ello no es obligatorio.

Invierte en lo que resta del curso el tiempo que creas que merece, sabiendo que ya tienes tu certificado; y profundiza cuanto quieras. Aprovecha la oportunidad de que dispones del foro para hacer preguntas.

Capítulo 11

Mandando y filtrando correo desde Perl

Una de las cosas más importantes que debemos aprender a hacer desde un lenguaje de scripting es mandar correo y filtrar el correo recibido. Entremos en contexto. No aprendemos un lenguaje de scripting por "culturilla general"; sino para automatizar tareas de administración de sistemas. Y, como dicen los anglosajones, "shit happens". La clave aquí es que debemos como administradores de sistemas recibir la información, cuanto antes, de que algo malo está pasando, para poder arreglarlo. También debemos poder mandar órdenes a sistemas remotos para que hagan cosas concretas. Y, ¿Cual es la mejor forma de hacer esto? El correo electrónico. Vía correo podemos hacer lo más urgente, y luego completar con túneles ssh.

11.1. Revisitando CPAN

El gran problema con CPAN no es que no existan clases para hacer de todo. Nos va a costar encontrar algo que no esté soportado por CPAN. El problema es que hay más de una clase que hace las mismas cosas, y algunas pueden no ser las más apropiadas. En este tema y el siguiente entraremos en un par de problemas concretos en los que encontraremos esta dificultad: no es que no hay clases que soporten la funcionalidad buscada, sino que muchas clases la soportan y tenemos que escoger la mejor.

Ahora, Manuel, vamos a hacer un ejercicio. Es importante que lo hagas, ya que eso es lo más importante que vas a aprender en estos dos últimos temas. El

primero: busca en CPAN una clase que permita mandar correo. Verás que hay varias. En base a la documentación, decide una. Cuando lo hayas hecho, pasa la página y veremos si nuestras elecciones coinciden.

...y mi elección es `MIME::Lite::TT`.

¿Por qué?

En primer lugar, porque funciona. He utilizado esa clase durante varios años, le he hecho toda clase de perrerías, y ha funcionado estupendamente.

En segundo lugar, porque soporta codificaciones no ASCII sin problemas; incluyendo UTF-8 y las distintas ISO.

En tercer lugar, porque soporta añadir ficheros adjuntos sin problemas y de forma fácil.

En cuarto lugar, porque esta clase permite envío simultáneo de la versión HTML y de texto del mensaje.

En quinto lugar, porque es extremadamente fácil usarla para envíos de correos masivos, ya que soporta parametrización.

Y finalmente porque soy un vago. Y aunque es cierto que hay clases que permiten hacer algunas de estas cosas, disponer de una clase que permite hacer todo esto a la vez con un mínimo de esfuerzo es un plus.

Ahora echamos un ojo a la documentación de CPAN, antes de continuar:
http://search.cpan.org/~horiuchi/MIME-Lite-TT-0.02/lib/MIME/Lite/TT.pm
Manuel: no pases la página hasta haber verificado este enlace en CPAN, y ver si entiendes la documentación **antes** de mi explicación.

¿Lo has hecho? Pues continuamos.

11.2. Usando la clase `MIME::Lite::TT`

Para usar esta clase, comenzamos importándola:

```
use MIME::Lite::TT;
```

Después debemos crear un par de variables; una para las opciones, y otra para los parámetros de los mensajes de correo, que se estructurarán como templates. Lo más común que haremos será definir como opciones la ruta del template:

```
$opciones{INCLUDE_PATH} = '/ruta';
```

Y como parámetros, los que incluiremos en el correo. Por ejemplo, podemos hacer:

```
$parametros{tratamiento}='Sr.';
$parametros{nombre}='Juan';
$parametros{cantidad}='1200,23';
$parametros{factura}='1243/32/2011';
```

El mensaje lo guardamos en un fichero de texto; por ejemplo, puede contener:

Estimado [% tratamiento %] [% nombre %]:

Le escribo para cordialmente recordarle que nos adeuda a día de hoy [% cantidad %] euros, a causa de la factura [% factura %] que aún no ha abonado.

Puede saldar esa deuda abonando los [% cantidad %] euros en mano, o transferiéndolo a la cuenta 2312-3233-23-023123213 de nuestra empresa, indicando en el campo observaciones “pago fact [% factura %]”.

Agradeciéndole de antemano su deferencia por contar con nosotros, atentamente:

David Santo Orcero

Autor del curso de Perl

Este template de mensaje –que para nuestro ejemplo supondremos que está en el fichero `template.txt.tt`– lo podemos reaprovechar para varios progra-

mas; incluso podemos, como sospechamos, extraer los parámetros del template de la base de datos, o de iterar en un bucle una estructura con los destinatarios del correo.

Lo siguiente será crear el mensaje desde el código, llamando al constructor de la clase. Los parámetros más comunes del constructor de clase son:

```
my $msg = MIME::Lite::TT->new(
        From    =>  remitente,
        Bcc     =>  destinatariosconcopiaoculta,
        Cc      =>  destinatariosconcopia,
        To      =>  destinatarios,
        Subject     =>  tema,
        Template    =>  ficherotemplate,
        TmplOptions =>  \%variableopcionestemplate,
        TmplParams  =>  \%variableparametrostemplate,
        Charset => codificación
        );
```

Donde `From`, `To`, `Subject`, `Bcc` y `Cc` tienen exactamente el mismo sentido que en un correo electrónico; y estos dos últimos campos –`Bcc` y `Cc`– son opcionales. Los campos restantes son:

- `Charset`: Corresponde a la codificación del mensaje.
- `Template`: Fichero con el template del mensaje.
- `TmplOptions`: Variable con las opciones del template.
- `TmplParams`: Variable con los parámetros del template: será donde ubicaremos lo parametrizable del correo.

Es importante que recordemos que, a efectos de parametrizabilidad, no solo podemos parametrizar el correo: cualquiera de los parámetros antes especificados de remitente, destinatario y temática, por poner ejemplo, puede corresponder a variables.

Para adjuntar ficheros, utilizaremos el método `attach`; con la sintaxis:

```
$msg->attach(
        Type     => tipo,
        Path     => "/ruta/ficheroaadjuntar",
        Filename => "nombreadjunto",
        Disposition => 'attachment'
        );
```

Donde:

- `Type` es el tipo de fichero.
- `Path` el nombre de fichero que queremos adjuntar, con su ruta completa.
- `Filename` el nombre que tendrá el fichero adjuntado. Esta última opción es especialmente útil, especialmente si generamos archivos de forma automatizada; ya que el que recibe el correo no ve el nombre que tenía el archivo al crearlo, sino el que indicamos en la opción `Filename`.

Finalmente, mandamos el mensaje de correo con:

```
 $msg->send;
```

Si tenemos un gestor de correo tipo `sendmail` instalado localmente o análogo; o con:

```
MIME::Lite->send('smtp', $servidorsmtpremoto,
        Hello=>$nombremáquinalocal,
        Port=>25,
        Debug=>0,
        AuthUser=>$usuariosmtp,
        AuthPass=>$clavesmtp,
        LocalAddr=>$iplocal);
```

En caso de que tengamos que hacer uso de un servidor remoto.

Los parámetros de `send` con conexión a servidor remoto son:

- `Hello`: Nombre local que tiene nuestra máquina.
- `Port`: Puerto del servidor SMTP remoto.
- `Debug`: 0 en caso de que no queramos información de depuración, 1 si la queremos. Esta información es extremadamente útil para depurar si no conseguimos mandar correos por problemas de autentificación.
- `AuthUser`: Usuario remoto del servidor SMTP.
- `AuthPass`: Clave remota del servidor SMTP.
- `LocalAddr`: IP local si tenemos más de una IP de salida, y queremos salir por una de las IPs en concreto.

11.2.1. Un ejemplo de mandar correo en Perl

Un ejemplo completo de código para mandar un correo en Perl es:

```
use MIME::Lite::TT;

$opciones{INCLUDE_PATH} = '/home/irbis/mandarfacturas';

$parametros{tratamiento}='Sr.';
$parametros{nombre}='Juan';
$parametros{cantidad}='1200,23';
$parametros{factura}='1243-32-2011';


 my $msg = MIME::Lite::TT->new(
        From        =>  'irbis@orcero.org',
        Bcc          =>  'facturasenviadas@orcero.org',
        To          => '␣juan@nopagamos.net',
        Subject     =>  "Factura␣no␣abonada␣de␣$parametros
           {cantidad}␣euros",
        Template    =>  "impago.txt.tt",
        TmplOptions =>  \%opciones,
```

```
        TmplParams  =>  \%parametros,
        Charset => 'utf8'
        );


$msg->attach(
        Type     => 'APPLICATION/PDF',
        Path     => "latex/factura.compilada.$parametros{
           factura}.pdf",
        Filename => "
           factura_david_santo_orcero_$parametros{factura
           }.pdf",
        Disposition => 'attachment'
        );

MIME::Lite->send('smtp', 'mail.orcero.org',
        Hello=>'servidor.orcero.org',
        Port=>25,
        Debug=>0,
        AuthUser=>'irbis',
        AuthPass=>'yquemasteda',
        LocalAddr=>'10.2.1.32');
```

11.3. Filtrando correo de entrada

Personalmente he probado muchas formas para filtrar correo de entrada desde Perl. La más potente es empleando Unix, y haciendo funcionar lo que podemos saber de cómo funciona la administración de sistemas Unix con Perl. Este método tiene numerosas ventajas respecto a la alternativa –hacer polling–: menor consumo de recursos, menor propensión a errores, y la más importante: se atiende de forma inmediata el correo recibido. Solo requiere instalar un MDA compatible con `procmail` correctamente instalado en la máquina.

Según sea lo que queramos hacer, podemos crear un usuario para mandarle órdenes, o filtrar correos sobre usuarios ya existentes. En ambos casos, con el usuario creado, editamos el fichero `.procmail` del usuario, añadiendo las líneas:

```
:0fw: nombrescript.lock
| /usr/bin/perl /rutascript/nombrescript.pl
```

Donde `nombrescript` será el nombre de nuestro script, y `rutascript` la ruta de nuestro script.

Haciendo esto, nuestro script se lanzará una vez por cada correo entrante; podemos parsear el correo que entra línea a línea con:

```
% Código de inicialización

while (defined($line = <STDIN>)) {
% Código para parsear y gestionar cada línea
                }

% Código con lo que hacemos una vez parseadas las líneas
   del correo entrante

% Cerramos lo abierto
```

Ojo: trazar un programa hecho así es un infierno,; ya que genera una instancia del script por cada correo entrante, y si falla, falla sin tumbar el servicio –estas dos cosas son muy buenas en producción, pero complican el desarrollo– por lo que es recomendable que el programa trabaje generando un volcado de lo que hace; para ello, en el código de inicialización hacemos:

```
open (HISTORICO, ">>␣/rutahistórico/nombrescript.log");
print HISTORICO "Nuevo␣mensaje␣entrante.\";
```

Y en el código de cierre:

```
print HISTORICO "Mensaje␣gestionado␣con␣éxito.\n";
close (HISTORICO);
```

Personalmente también me gusta volcar las líneas del mensaje de correo se-

gún las estoy gestionando, poniendo:

```
print HISTORICO "($line)\n";
```

Dentro del bucle de gestión del correo.

Con estas cosas, hacer un programa que gestione correos entrantes y verificarlo es fácil; y el resultado de hacerlo así en entornos de producción es espectacular, tanto en eficiencia como en fiabilidad y tolerancia a fallos.

11.4. Actividades

Este tema cuenta con dos actividades.

La primera es modificar el script de gestión de históricos de Apache para que cuando detecte un acceso desde el departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga cualquier fichero con extensión pdf en el servidor, mande al administrador de sistemas un correo por cada fichero pdf accedido. Notificando todos los datos sobre dicho acceso.

La segunda actividad es un programa de gestión remota vía correo. Suponemos que el correo contiene en su tema una clave, en su primera línea un comando, y en el restante de líneas un conjunto de cosas. Ante cada correo entrante, el programa debe hacer lo siguiente:

- Verificar que la dirección de correos de origen está en una tabla de direcciones de correo permitidas, que tenemos en una base de datos.
- Verificar que la clave es la correcta, respecto a una almacenada en la base de datos.
- Verificar que el comando está permitido, respecto a una lista de comandos permitidos especificada en una tabla de la base de datos.
- Ejecutar el comando especificado, pasándo a dicho comando como órdenes aquello que aparezca en el mensaje de correo hasta el final de dicho mensaje, como si fuese su entrada estándar.
- Reenviar la salida estándar y la salida de error del programa ejecutado al remitente del correo.

Estos dos programas formarán parte de tu juego de soluciones a problemas reales; y creeme, Manuel, que te sacarán de algún problema real en el futuro.

Después de haber probado los ejemplos de los apuntes, y de haber intentado el ejercicio, no dudes en utilizar el foro para preguntar todo lo que no entiendas.

Capítulo 12

Programando un *spider* en Perl

Lo último que aprenderemos en nuestro curso será a programar un spider *en Perl. Como en el caso del envío y recepción de correo, y otras muchas cosas en Perl, el problema no es que no haya ninguna forma de hacerlo: muy al contrario, hay muchísimas formas de hacerlo, y tenemos numerosas clases en CPAN que nos permiten programar un* spider *con facilidad. Veremos la que mejor funciona, aprenderemos a utilizarla, y pondremos a trabajar todo nuestro conocimiento de Perl en conjunto.*

12.1. Qué es un *spider*

Un *spider* es un programa que recorre de forma automatizada la WWW, con objeto de recopilar información para su procesado automático.

Podemos querer hacer esto para tareas tan simples como verificar de forma automatizada que una aplicación web está activa y funciona correctamente –no solo que Apache muestra las páginas, sino que funciona la aplicación–. Podemos querer desarrollar una batería de pruebas automatizadas en una aplicación web, de la forma más realista que sea posible –lo que supone miles de accesos concurrentes, haciendo operaciones con la aplicación que tengan sentido–. Podemos querer mandar órdenes de forma automatizada a un router remoto, que solo soporta accesos vía web. Podemos querer automatizar tareas de gestión de un foro. Recopilar información para realizar tareas de inteligencia de negocio –o de contrainteligencia–. O queremos hacer como que estamos haciendo una cosa, mientras que realmente estamos haciendo otra.

El problema de un buen *spider* es que parece fácil, pero no lo es. Tenemos que hacernos pasar por un humano, o las páginas nos mostrarán información distinta a la que queremos –o incluso ni nos mostrarán información–. Además, las páginas están cambiando constantemente; y tenemos que seguir enlaces y formularios que cambian constantemente, por lo que no nos basta con "recordar" un camino: hay que parsear la página web, y seguirla como la navega un humano: disparamos a un blanco móvil. Y después tenemos lo de las "cookies", y el seguimiento desde servidor a través de variables de sesión...

Mucho trabajo. ¿O no?

Aquí, Manuel, comenzaremos repitiendo el ejercicio con el que comenzamos el tema anterior. Busca en CPAN como ya has aprendido, y localiza los paquetes para automatizar la navegación web. Escoge el que te parezca mejor. Cuando lo hayas decidido, pasa a la página siguiente y veremos si tu decisión coincide con lo que ha sido mi experiencia. No te preocupes: aprenderás a manejar el mejor paquete disponible para programar *spiders* en Perl a lo largo de este último tema.

. . . y mi elección es `Mechanize`.

¿Por qué?

Porque lo hace `todo`. Además, seguir lo que es una "navegación normal" es muy sencillo; y trae funciones para extraer los elementos de uso más común de una página web. Es la clase que yo utilizo habitualmente, y funciona –realmente bien, de hecho–.

Ahora echamos un ojo a la documentación de CPAN, antes de continuar. En este caso, tenemos cuatro páginas, que en orden de utilidad son:

`http://search.cpan.org/dist/WWW-Mechanize/lib/WWW/Mechanize/Cookbook.pod`

`http://search.cpan.org/dist/WWW-Mechanize/lib/WWW/Mechanize.pm`

`http://search.cpan.org/dist/WWW-Mechanize/lib/WWW/Mechanize/FAQ.pod`

`http://search.cpan.org/dist/WWW-Mechanize/lib/WWW/Mechanize/Examples.pod`

El tercer y el cuarto enlaces es información que es recomendable que leas *después* de haber leído el tema completo, ya que son densos y demasiado específicos, en mi opinión.

Sin embargo, el primer y el segundo enlaces son muy interesantes. El segundo es una guía bastante extensa del API que acompaña a esta clase. Más extensa incluso que el material que acompaña a este tema. Es importante que guardes la referencia.

El primero es un enlace muy importante; ya que da una explicación muy sencilla de cómo funciona `Mechanize`. Es muy recomendable, Manuel, que antes de seguir con el curso pares, pruebes los ejemplos indicados en:

http://search.cpan.org/dist/WWW-Mechanize/lib/WWW/Mechanize/Cookbook.pod

Aunque aún no entiendas lo que estás haciendo, y ojees la documentación:

http://search.cpan.org/dist/WWW-Mechanize/lib/WWW/Mechanize.pm

Y no pases la página hasta haber verificado estos enlaces en CPAN.

¿Lo has hecho? Específicamente, ¿Has probado en tu ordenador los ejemplos del primer enlace? Pues continuamos.

12.2. Usando la clase `Mechanize`

Mechanize es la más potente biblioteca que existe en Perl para programar `spiders`. Y es la más potente porque hace absolutamente de todo.

Tal y como hemos comentado, encontamos toda la información necesaria sobre `Mechanize` en la web:

`http://search.cpan.org/~jesse/WWW-Mechanize-1.72/lib/WWW/Mechanize.pm`

Esta biblioteca es tan rica que podríamos escribir un libro entero sobre ella[1]; y tenemos multitud de artículos escritos sobre la biblioteca de interés. La biblioteca tiene decenas de métodos, para hacer cosas realmente específicas. Comentaremos solo los más importantes; aunque lo normal es que con mandar información a través de formularios y extraer páginas y enlaces tengamos todo lo necesario.

El constructor de la clase es:

```
new()
```

Que permite crear un objeto de clase `Mechanize`. Este objeto guardará la información del estado de la navegación, y podemos interpretarlo como una persona navegando con una web, que realizará las peticiones que se realicen sobre el objeto, de forma secuencial. La forma de llamarlo es algo como:

```
my $mech = WWW::Mechanize->new(
    agent => nombreagente,
    autocheck => verificación,
    quiet => verbosidad
    )
```

De los parámetros del constructor, los más importantes son:

- `agent`: Indicamos el nombre del agente; es decir, quién dice el navegador que es. Fundamental para hacerse pasar por un navegador humano.
- `agent_alias`: Indicamos un agente por el que nos haremos pasar, de entre la siguiente lista:
  - `Windows IE 6`
  - `Windows Mozilla`

[1] De hecho, O'Reilly tiene publicado un libro completo sobre ella.

- `Mac Safari`
- `Mac Mozilla`
- `Linux Mozilla`
- `Linux Konqueror`

Si usamos `agent_alias`, no debemos usar `agent`. Por otro lado, con el método `known_agent_aliases()` podemos obtener una lista completa de todos los agentes emulados mediante la versión concreta de `Mechanizer` que estemos usando.

- `autocheck`: Si asignamos 1 a este atributo, tendremos que en caso de error en descarga el método que llamamos devolverá un error. Si le asignamos un 0, el método no devolverá error si ha habido un error, por lo que debemos verificar explícitamente que ha funcionado todo.
- `quiet`: Si asignamos 1 al atributo, el objeto será especialmente verboso; dándonos información de avisos y de cosas que pueden no estar funcionando correctamente. Esta opción es fundamental para depurar la navegación con `Mechanizer`; así como también lo es imprimir en pantalla la página web obtenida mientras que estamos navegando en pruebas, durante desarrollo.

Todos estos atributos del constructor tienen un método de idéntico nombre que permite modificar el valor en cualquier momento futuro.

Una vez creado el objeto de la clase `Mechanizer`, nos podemos poner a descargar páginas. La forma más sencilla de hacerlo es con:

```
$mech->get( $url,
        ':content_file' => $tempfile )
```

El parámetro `':content_file'` es opcional, y se emplea para volcar en un fichero lo obtenido por `Mechanizer`.

Este comando nos ha permitido extraer la página con `GET`; si queremos extraerla con `PUT`, haremos:

```
$mech->put( $url,
        ':content_file' => $tempfile )
```

Donde el parámetro `:content_file'` es opcional; y, como en el caso anterior, también se emplea para volcar en un fichero lo obtenido por `Mechanizer`.

También podemos añadir cabeceras al agente que se añaden a la petición HTTP con:

```
$mech->add_header( nombre => $valor,
                   nombre => $valor,
                   ...
                   nombre => $valor)
```

Pudiendo añadir todas las que queramos, mientras que las separemos por comas. Las etiquetas más comunes son *Referer*, o *Encoding*; pero podemos tener muchas más. Asignando el valor de una etiqueta a `undef`, no se envía más dicha etiqueta al servidor.

Podemos "pulsar" el botón de "hacia atrás" de nuestro navegador emulado, con:

```
$mech->back()
```

Y podemos "pulsar" el botón de "recarga" de nuestro navegador emulado, con:

```
$mech->reload()
```

Siempre que queramos, podemos "clonar" nuestro navegador con:

```
$mech2=$mech->clone()
```

Después de esto tendremos dos navegadores: `mech` y `mech2`. Ambos tendrán las mismas cookies y el mismo estado de navegación, y podremos tener navegaciones "distintas" en ambos navegadores como si a partir de este punto fueran navegadores separados.

12.3. Extrayendo información de las páginas descargadas

Suponiendo que hayamos descargado ya la primera página por alguno de los métodos anteriormente comentados, podemos comenzar a extraer toda la información que necesitamos de la página.

Podemos saber si ha ido bien la descarga, consultando:

```
$mech->success()
```

Si queremos obtener el URL de lo que realmente hemos descargado, en contexto de cadena llamamos a la función:

```
$mech->uri()
```

Podemos obtener el código de estado HTTP devuelto por el servidor con:

```
$mech->status()
```

Esto corresponde con el código de tres dígitos numéricos que da el servidor –"OK" es 200, "Prohibido" es 403, "No encontrado" será 404...—.

Obtenemos información de si nos hemos descargado o no una página HTML con:

```
$mech->is_html()
```

Aunque podemos obtener el tipo de contenido con:

```
$mech->content_type()
```

El URL base de la respuesta, lo obtenemos con:

```
$mech->base()
```

Obtenemos el título de la página descargada con:

```
$mech->title()
```

Podemos obtener una lista de los formularios de la página utilizando:

```
$mech->forms()
```

o el formulario actualmente activo con:

```
$mech->current_form()
```

vamos a volver a los formularios más adelante.

Podemos obtener una lista de los enlaces de la página descargada con:

```
$mech->links()
```

Volveremos a los enlaces más adelante, cuando veamos como seguirlos.

Podemos obtener el contenido de la página que acabamos de descargar con:

```
$mech->content( base_href => base)
```

Lo que nos devolverá la página web, con los enlaces referenciados respecto de `base`. Si no indicamos `base_ref`, este método empleara como valor de `base_ref` aquel valor devuelto por la función `$mech->base()`.

Podemos obtener directamente el volcado de la página en texto plano, con el HTML eliminado, después de parseado el HTML. Esto lo haremos con:

```
$mech->content(format => 'text' )
```

Pero debes recordar, Manuel, que esto solo funcionará si antes tienes instalada la clase `HTML::TreeBuilder`.

12.4. Siguiendo enlaces

Mediante el método `links`, hemos obtenido los enlaces de la página. Una de las características más prácticas de `Mechanizer` es que no necesitamos extraer enlaces, procesarlos, y realimentarlos con `GET` o `POST`. No es la forma como navegamos normalmente, por lo que no tiene sentido como mecánica. La mecánica común de uso de `Mechanizer` es localizar un enlace, y seguirlo. Localizamos un enlace con la función:

```
$mech->find_link(text_regex=>criterio)
```

Donde `criterio` es una expresión regular; este método devolverá un enlace que cumpla la expresión regular. Por ejemplo, si queremos seguir en la página web actual el enlace que contiene la palabra `enviar`, hacemos:

```
$mech->find_link( text_regex => qr/enviar/i )
```

Podemos también localizar un enlace por URL relativo:

```
$mech->find_link(url=>enlace)
```

Por URL absoluto:

```
$mech->find_link(url_abs=>enlace)
```

Por expresión regular en el enlace:

```
$mech->find_link(url_regex=>criterio)
```

O por etiqueta:

```
$mech->find_link(tag=>etiqueta)
```

Finalmente, podemos seguir un enlace por su número de enlace con:

```
$mech->find_link(n=>número)
```

Hay muchos criterios adicionales; aunque en mi experiencia estos son los más comunes. También podemos utilizar varios de estos criterios a la vez; con el objeto de identificar unívocamente el enlace a seguir; por ejemplo, podemos hacer:

```
$mech->find_link(
        text_regex => qr/enviar/i,
        url_regex => qr/sitionavegado\.com/i,
        tag=>'alta')
```

Es clave que entendamos que debemos ser lo suficientemente específicos como para que las condiciones que pasamos a `find_link` nos devuelvan un único enlace, pero lo suficientemente genéricos como para asegurarnos que ante pequeños cambios en las páginas que navegamos no tengamos que retocar nuestro *spider*. Mi experiencia es que si las páginas están bien hechas, habitualmente obtendremos los mejores resultados filtrando mediante los `tags` y una expresión regular que nos evite salirnos del sitio web navegado; y si está mal hecho, que la expresión regular sobre el texto del enlace y la expresión regular que nos evite salirnos del sitio web navegado será suficiente. Por regla general, es una muy mala idea navegar buscando nombres concretos en la URL que seguimos, ya que las páginas web son blancos móviles y este criterio de selección nos dará muchísimo trabajo de mantenimiento; y seguir un enlace por su número de enlace dentro de la página es aún peor idea por la misma razón.

Es importante destacar que no seguimos solo los enlaces "comunes"; sino que mediante `find_link` seguiremos los enlaces definidos por `<a href="">`, `<area href="">`, `<frame src="">`, `<iframe src="">`, `<link href="">` y `<meta content="">`. Por su mecanismo de conservación de estado, `Mechanize`

es razonablemente robusto siguiendo enlaces ofuscados, incluso a ofuscados con javascript.

A veces no queremos un enlace en concreto; sino un conjunto de enlaces que luego seguiremos sucesivamente. Esto lo hacemos con:

```
$mech->find_all_links()
```

Los parámetros de filtro son exactamente los mismos que con `find_link`.

Llegados a este punto, tenemos un enlace obtenido con `find_link`; o un conjunto de enlaces obtenidos con `find_all_links`. El primero nos devolverá un objeto de clase `WWW::Mechanize::Link`; y el segundo una estructura de objetos de clase `WWW::Mechanize::Link`[2].

Una vez que tenemos el objeto de clase `WWW::Mechanize::Link`, seguimos el enlace mediante los métodos `get` o `put`, que ya hemos estudiado.

Podemos buscar y seguir el enlace en un único paso, con:

```
$mech->follow_link(criterios)
```

donde en `criterios` podemos emplear los mismos criterios que hemos estudiado para `find_link`; y básicamente `follow_link` hará lo mismo que haciendo un `find_link` con un `get` posterior. Si el criterio de búsqueda no localiza un único enlace válido, este método devuelve `undef`.

12.5. Rellenando y mandando formularios

El punto donde fallan la mayor parte de los mecanismos de navegación web automatizada es en cómo trabajan los formularios. De hecho, tanto `find_link` como `follow_link` son un mecanismo adecuado para navegar una página con `GET`, pero son un incordio para formulatios.

`Mechanize` también resuelve el problema de los formularios.

La primera función que debemos conocer para trabajar con formularios es:

```
$mech->forms
```

esta función nos permite obtener una estructura[3] con todos los formularios

[2] Para ser precisos, lo que obtenemos es una matriz en contexto de lista, y una referencia a matriz en contesto escalar.

[3] Para ser precisos, una matriz en contexto de lista, y una referencia a matriz en contesto escalar.

de la página; realmente, este método nos devolverá una estructura de objetos de clase `HTML::Form`.

Podemos seleccionar un formulario en concreto por número de formulario con:

```
$mech->form_number(número)
```

Importante, *comenzando por uno*[4]. O seleccionar un formulario en concreto con:

```
$mech->form_name(nombre)
```

Donde escogerá el primer formulario con ese nombre.

Si queremos coger un formulario por ID, haremos:

```
$mech->form_id(id)
```

Escogiendo el primer formulario con esa ID.

Personalmente, Manuel, no te recomiendo ninguna de estas opciones, ya que están muy sujetas a cambios en la página, y obligan a un mantenimiento alto de la página. La opción más robusta para escoger un formulario de la página es:

```
$mech->form_with_fields( @campos)
```

Donde `@campos` contiene un subconjunto de los campos que ese formulario debe que tener; con lo que, por ejemplo, si estamos rellenando un formulario donde esperamos rellenar un `username` y un `passwd`, por más que nos cambien los nombres a las cosas y nos reorganicen la página, mientras que mantengan los nombres de los campos que pasamos a la página siguiente –lo que es muy común–, lo que hagamos será muy robusto a cambios de página. Y si deja de funcionar nuestro *script* porque no pasa a través del formulario, ya sabemos que nos han tocado los nombres de los parámetros del formulario: cambiamos los nombres, y todo arreglado. Fácil de depurar, puesto que solo hay una cosa que potencialmente puede fallar si cambian –por lo que sabemos a priori donde mirar cuando falle–, y además es lo que es menos probable que cambien.

Llegados a este punto, tenemos un formulario seleccionado –o `undef`, si no

[4] Esto, en concreto, me hizo perder una tarde completa en su día. Y es que las personas de bien comienzan a contar desde cero…

existía un formulario con el criterio que hemos tomado–. Ahora pasamos a trabajar sobre el formulario en concreto.

Obtenemos el valor de un campo en concreto del formulario seleccionado con:

```
$mech->value( nombre,
                número)
```

Donde `nombre` será el nombre del campo del formulario cuyo valor queremos recuperar; y `número` será un parámetro opcional que determinará en caso de que el formulario tenga más de un parámetro con el mismo nombre, de a cual de ellos obtendremos el valor. Si no lo indicamos, `value` devolverá el valor del primer campo que encuentre cuyo nombre sea `nombre`. Como en el caso de los números de formulario, comienza contando por 1. Esta función es muy importante, ya que a veces necesitamos recuperar el valor por defecto de un campo de formulario como mecanismo para calcular el valor de otro campo del formulario. Es un mecanismo muy común para bloquear *spiders*, y haciendo uso de la función `value` podemos saltárnoslo.

Por otro lado, podemos rellenar un campo en concreto del formulario seleccionado con:

```
$mech->field(nombre,
        valor,
        número )
```

Donde asignaremos al parámetro `nombre` del formulario el valor `valor`. `valor` puede ser un escalar o un vector; si `valor` es un vector y el campo no es de asignación múltiple, asignará el primero. Si es de asignación múltiple, asignará todos los valores indicados en el vector.

`número` un parámetro opcional, que determinará en caso de que el formulario tenga más de un parámetro con el mismo nombre, a cual de ellos le asignamos el valor especificado. Si no lo indicamos, lo asignará al primero. Como en el caso de los números de formulario, comienza contando por 1.

Podemos asignar todos los parámetros del formulario activo de una sola vez empleando:

```
$mech->set_fields(  )
```

Por ejemplo, podemos hacer:

```
$mech->set_fields( username => 'pepe',
                   passwd=>'pepe0' )
```

En el caso concreto de los checkbox, podemos marcarlos con:

```
$mech->tick( nombre,
                   valor )
```

Los desmarcamos haciendo:

```
$mech->tick( nombre,
                   valor,
                   false)
```

Lo que es lo mismo que:

```
$mech->untick( nombre,
                   valor)
```

Podemos conseguir el mismo resultado empleando `undef` en lugar de `false`.

Un mecanismo de navegación muy común para bloquear a los *spiders* son los campos de formulario de tipo "fichero". La idea es poner una imagen, y que se pulse en un lugar concreto.

Cómo encontrar las coordenadas donde pulsar es algo en lo que no vamos a entrar; aunque hay soluciones, de la misma forma que para saltar los captchas[5]. Sí como pulsar en un punto concreto. Lo hacemos con:

```
$mech->click(nombre,
                   coordenadaX,
                   coordenadaY
                   )
```

A veces se trata de un control de audio; en cuyo caso lo hacemos con:

```
$mech->click(nombre,
                   coordenadaX
                   )
```

[5]En algún lado, Manuel, hay que parar el contenido del curso; además de que ya entra dentro del material "Black Hat", y eso no lo voy a explicar.

Finalmente, tenemos que enviar el formulario. Hay varias formas de hacerlo; la primera es enviar la página, sin pulsar botón alguno:

```
$mech->submit()
```

La segunda, pulsar una imagen; si no tenemos ninguna posición del botón que nos interese pulsar de forma específica, podemos hacer:

```
$mech->click(nombre)
```

Esto es muy común hacerlo cuando solo queremos enviar el formulario ya rellenado, y tenemos identificado el nombre del botón que tenemos que rellenar para enviarlo.

A veces, no tenemos identificado el nombre del botón, por lo que tenemos que "buscarlo". Para ello, emplearemos el método:

```
$mech->click_button(criterio)
```

Este método buscará en el formulario activo un botón concreto, según el criterio determinado por `criterio`. Este parámetro `criterio` es análogo a los parámetros de criterio que hemos empleado para identificar enlaces y formularios; y en el caso concreto de los botones habitualmente son `name` o `number`; siendo `number` un número igual o superior a uno –como en casos anteriores, comenzamos contando por la unidad–.

Para terminar, podemos seleccionar, rellenar y enviar un formulario con una única instrucción. Esta será:

```
$mech->submit_form(
        form_number => número,
        form_name => nombre,
        form_id => ID,

        with_fields => campos,

        fields => campos,

         x => x, y => y,
        button => nombrebotón
        )
```

Muchos parámetros de los aquí especificados son opcionales, otros tienen características específicas. Vamos a verlos uno a uno:

- `form_number`: Este parámetro se emplea para seleccionar el formulario a través del número de formulario, dentro de la página. Como siempre, contando a partir del uno.
- `form_name`: Este parámetro se emplea para seleccionar el formulario a través del nombre del formulario.
- `form_id`: Este parámetro se emplea para seleccionar el formulario a través del ID del formulario.
- `fields`: Empleamos este parámetro para indicar los valores de los campos, y el parámetro `campos` será un hash en el que enviamos los campos rellenos; las claves serán los nombres de los campos, y el valor asociado a cada clave el valor que enviemos para dicho campo. Si no tenemos definido el valor de algún campo, `Mechanize` enviará el valor por defecto.
- `with_fields`: Esta opción es muy interesante, ya que será la que emplearemos con más frecuencia si empleamos `submit_form`. Primero busca el formulario cuyos campos coincidan con los pasados en el hash `campos`. Seleccionado el primero de los formularios que cumpla esta condición, interpretará ahora `campos` como un hash en el que enviamos los campos rellenos; las claves serán los nombres de los campos, y el valor asociado a cada clave el valor que enviemos para dicho campo. Si no tenemos definido el valor de algún campo, `Mechanize` enviará el valor por defecto. Si incluimos esta opción, `Mechanize` ignorará lo que le hemos pasado como valores en los parámetros `form_number`, `form_name`, `form_id` y `fields`.
- `x`: Campo opcional; corresponde con la coordenada X del botón pulsado.
- `y`: Campo opcional; corresponde con la coordenada Y del botón pulsado.
- `button`: Botón pulsado. Si no indicamos explícitamente el botón pulsado, es lo mismo que si llamamos a `$mech->submit()` después de buscar el formulario y rellenar sus campos.

A diferencia de otras formas de hacer esto, si mediante el mecanismo de selección de formularios por el que hemos optado no selecciona ningún formulario –porque ninguno cumple la condición– seleccionará el primer formulario de la página web aunque no cumpla la condición, le mandará los campos que hemos seleccionado con los valores que hemos seleccionado, y pulsará el botón indicado; si el botón tampoco existe, le mandará un `$mech->submit()`.

12.6. Operando con imágenes

Dentro de los enlaces, a veces no queremos seguir el enlace de por sí, sino descargar el contenido al que enlaza. Con frecuencia se trata de imágenes, ya que es uno de los contenidos sobre los que suele haber más interés.

Para obtener todas las imágenes, utilizamos el método:

```
$mech->images
```

Método que nos devuelve una estructura[6] con todas las imágenes de la página actual. Este método, como casi todos los que trabajan con imágenes de la clase, devolverá un objeto de clase `WWW::Mechanize::Image`.

Podemos obtener una imagen concreta de entre las que referencia la página mediante el método:

```
$mech->find_image(criterio)
```

Donde `criterio` es el criterio que emplea `Mechanize` para seleccionar de entre las imágenes de la página la imagen que buscamos; si ninguna imagen cumple el criterio, el método devuelve el valor `undef`. Los criterios más comunes de esta función son:

- `alt`: Empareja con el atributo `ALT` de la imagen, realizando un emparejado idéntico.
- `alt_regex`: Empareja con el atributo `ALT` de la imagen, realizando un emparejado mediante una expresión regular. En una página bien hecha, este criterio suele ser el más efectivo.
- `url`: Empareja con la URL relativa de la imagen, realizando un emparejado idéntico.

[6] En contexto escalar una referencia a una matriz, en contexto vectorial, una lista.

- `url_regex`: Empareja con la URL relativa de la imagen, realizando un emparejado mediante una expresión regular.
- `url_abs`: Empareja con la URL absoluta de la imagen, realizando un emparejado idéntico.
- `url_abs_regex`: Empareja con la URL absoluta de la imagen, realizando un emparejado mediante una expresión regular.
- `tag`: Empareja con el tag de la imagen, realizando un emparejado idéntico. `Mechanizer` soporta solo los tags `<img>` e `<input>`.
- `tag_regex`: Desde el momento que `Mechanizer` soporta solamente dos tags, personalmente no le veo mucho sentido a esta opción.
- `n`: la enésima imagen, comenzando a contar desde uno.

Si ponemos más de una condición, se buscará la imagen que cumpla con todas las condiciones especificadas a la vez; por ejemplo, podemos hacer:

```
$mech->find_image(
                  alt_regex=> qr/captcha/i,
                  url_regex => qr/orcero\.org/ ,
                 tag_regex=> qr/img/);
```

También podemos buscar un conjunto de imágenes que cubre un criterio concreto; lo que haremos con el método:

```
$mech->find_all_images(criterio)
```

Como con el caso anterior, `criterio` es el criterio que emplea `Mechanize` para seleccionar la lista de imágenes que buscamos, y devuelve una estructura[7] con todas las imágenes de la página actual. Como en los casos anteriores, las imágenes se almacenan en objetos de clase `WWW::Mechanize::Image`.

Si no se indica ningún criterio, la función `$mech->find_all_images()` devolverá una lista que contendrá todas las imágenes de la página. Como en el caso anterior, los criterios serán:

- `alt`: Empareja con el atributo `ALT` de la imagen, realizando un emparejado idéntico.

[7] En contexto escalar una referencia a una matriz, en contexto vectorial, una lista.

- `alt_regex`: Empareja con el atributo `ALT` de la imagen, realizando un emparejado mediante una expresión regular. En una página bien hecha, este criterio suele ser el más efectivo.
- `url`: Empareja con la URL relativa de la imagen, realizando un emparejado idéntico.
- `url_regex`: Empareja con la URL relativa de la imagen, realizando un emparejado mediante una expresión regular.
- `url_abs`: Empareja con la URL absoluta de la imagen, realizando un emparejado idéntico.
- `url_abs_regex`: Empareja con la URL absoluta de la imagen, realizando un emparejado mediante una expresión regular.
- `tag`: Empareja con el tag de la imagen, realizando un emparejado idéntico. `Mechanizer` soporta solo los tags `<img>` e `<input>`.
- `tag_regex`: Desde el momento que `Mechanizer` soporta solamente dos tags, personalmente no le veo mucho sentido a esta opción.

Como en el caso de `find_images()`, si indicamos más de un criterio se buscarán las imágenes que cumplan todos los criterios a la vez; pero a diferencia de `find_images()`, que espera retornar solo emparejar con una imagen, y por lo tanto devolver un único objeto `WWW::Mechanize::Image`, este método devolverá varias imágenes.

12.7. Volcados a fichero

Finalmente, podemos querer volcar todo o parte de nuestra navegación a ficheros. Volcamos las cabeceras a un fichero, con:

```
$mech->dump_headers(descriptor)
```

donde `descriptor` es un descriptor de ficheros. En caso de no indicar el descriptor, vuelca el resultado a la salida estándar.

Volcamos los enlaces relativos con:

```
$mech->dump_links(descriptor)
```

Como en el caso anterior, `descriptor` es un descriptor de ficheros. En caso de no indicar el descriptor, vuelca el resultado a la salida estándar. Los enlaces absolutos podemos volcarlos usando:

```
$mech->dump_links(descriptor, absoluto)
```

si `absoluto` vale `true`. Como en el caso anterior, `descriptor` es un descriptor de ficheros. En caso de poner como primer parámetro de este método el valor `undef`, la función vuelca el resultado a la salida estándar.

Tenemos una función análoga para imágenes:

```
$mech->dump_images(descriptor, absoluto)
```

Para formularios:

```
$mech->dump_forms(descriptor)
```

Y para el texto de la página:

```
$mech->dump_forms(descriptor)
```

Cuyos parámetros tienen la misma interpretación que los de `dump_links`.

12.8. Actividades

Llegamos al final del curso. Esta es la última actividad.

El objetivo de esta actividad es "atravesar" un formulario.

El formulario lo encuentras en:
`http://www.orcero.org/irbis/cursoperl/practicaspider.php`

La página tiene numerosos enlaces y formularios. El que interesa es uno que tiene dos campos visibles –`nombre` y `apellidos`– y uno oculto `apellidosV`.

El objetivo de la práctica es conseguir llegar a la felicitación. Para conseguirlo, hay que pasar a través del formulario correcto; mandando como parámetro `apellidos` y `apellidosV` los valores que ya tienen por defecto, y en `nombre` la suma aritmética de ambos valores.

Hay alguna "sorpresita" –hay cosas que cambian constantemente de las páginas cada vez que te las bajas; lo que no suele ocurrir así, pero sí ocurre que la página cambia a lo largo del tiempo–. Pero aún con eso, hay soluciones a este problema, usando `Mechanizer` que son robustas a las "sorpresas", y no ocupan más de media docena de líneas de código.

No intentes solucionar la práctica "a la fuerza bruta". Intenta hacer un buen uso de las facilidades de `Mechanizer`. Aprenderás mucho, y lo que aprendas te va a ser muy útil.

12.9. Conclusión y bibliografía anotada

Con esto, Manuel, has terminado finalmente el curso de Perl. Sé que ha sido largo, sé que ha sido denso. Pero hemos visto cosas que me han sido de utilidad, y sospecho que a ti probablemente te lo serán.

Si te quedas con lo que has visto en este curso, ya sabes una cantidad interesante de Perl. Pero quizás te has quedado con el "gusanillo" de aprender más. Para ello, tienes algunos libros y foros que te serán de utilidad:

- Respecto a libros, O'Reilly tiene muchos libros muy buenos sobre Perl. La referencia base es el "Programming Perl", escrito por Larry Wall, Tom Christiansen y Jon Orwant. Actualmente va por su tercera edición. O'Reilly tiene muchos libros sobre aplicaciones concretas de Perl, por lo que probablemente te interese el 'Programming Perl" y además el específico para la aplicación de Perl que tengas en mente. Es material muy recomendable.

- El "Manual de referencia de Perl" de Martin Brown editado por Osborne – McGraw-Hill dedica menos de cincuenta páginas al lenguaje en sí. El resto es sobre aplicaciones de Perl. Personalmente lo encuentro confuso y obsoleto, y no lo recomiendo. El libro de "Perl sin errores", del mismo autor y editorial, tampoco lo recomiendo; ya que podría ser resumido en la décima parte de su tamaño sin pérdida de profundidad. En este curso hemos comentado más material incluso del que tenemos en ese libro.

- Los paquetes y las clases de CPAN suelen estar realmente bien documentadas en el repositorio. Dado que los API son "blancos móviles", no te recomiendo comprar libros específicos de APIs, y sí acostumbrarte a buscar la información en el propio CPAN.

- Dentro de foros y comunidades, hay muchísima información en el sitio *Perl Monks*, que encontramos en:

  `http://www.perlmonks.org/`

  Aquí encontraremos trucos y trozos de código que resuelven problemas concretos de Perl.

- Otra cosa que debemos tener en cuenta es que con frecuencia tenemos grupos de usuarios de Perl cerca de nuestra zona. El ente para localizar grupos de usuarios de Perl son los *Perl Mongers*, que encontramos en:

`http://www.pm.org/`

Aunque fuera de Madrid y Barcelona, los grupos de *Perl Mongers* españoles están bastante muertos.

- Lo que sí está muy activo es la comunidad de Perl de habla hispana, a través de la web:

  `http://perlenespanol.com/`

  Desde este portal accedemos a decenas de tutoriales, centenares de artículos, y un foro muy activo donde intercambia información diariamente gente con niveles de Perl muy diversos –algunos realmente buenos–.

Si tuviese que recomendarte solo tres fuentes de información, te recomendaría "Programming Perl" de O'Reilly, la propia información de CPAN, y el foro de `http://perlenespanol.com/`. Y como cuarta y adicional, `http://www.perlmonks.org/`. Con eso tienes para avanzar y resolver muchos problemas reales complicados de forma que te de poco tiempo y trabajo –que es de lo que trata Perl, al fin y al cabo–.