Perl para apresurados

Juan Julián Merelo Guervós

Historial de revisiones

Revisión 1.0 Jul 2006
Preparando la primera versión para el curso de Extremadura

1. ¿Quién eres tú?

Eso mismo te estarás preguntando, que quién diablos eres, que a qué dedicas el tiempo libre, todo eso. Así que te vamos a echar una mano. Supongo que ya sabes programar, que el concepto de *variable* no va para ti asociado a la nubosidad ni el de *bucle* a la cabeza de Nellie Olleson. Puede que conozcas el C, sólo para precavidos, o hables con lengua de serpiente (pitón (http://www.python.org)), o incluso que el símbolo mayor y menor van para tí asociados de forma indisoluble a un acrónimo capicúa (http://www.php.org).

Vamos, que puede extrañarte las formas ignotas en las que un nuevo lenguaje de programación repite cachos de código o mete valores en variables o representa listas de datos, pero los conceptos en sí no son nada nuevo para tí. A tí, pues, va dirigido este mini-tutorial.

Supongo también que tienes prisa. Si no, no estarías leyendo este tutorial para *apresurados*. Estarías leyendo uno titulado, por ejemplo, *Perl para los que tienen todo el tiempo del mundo*. Es decir, que es necesariamente breve, con la idea de poder ser impartido (y espero que asimilado) en unas dos horas. Igual no te da tiempo a teclear todos los ejemplos de código, pero este ordenador que estás mirando tiene una cosa maravillosa llamada "corta y pega" con la que sin duda estás familiarizado, y que podrás usar para tu provecho y el de la Humanidad.

Y quizás todavía no lo sabes, pero *necesitas* saber Perl. Para vigilar ese fichero de registro y crear alertas que te avisen de lo inesperado. Para ese CGI terriblemente complicado. Para convertir una página web demasiado compleja en algo que también es complejo, pero que puedes leer con tu lector de cosas complejas favorito. Para hacer lo que siempre quisiste hacer: escribir poesía (http://www.perlmonks.org/index.pl?node=Perl%20Poetry) en tu lenguaje de programación favorito. En fin, donde quiera que haga falta convertir cosas en otras cosas, ahí hace falta saber Perl.

> **Sugerencia:** Y con ello damos entrada a la primera *flamewar* de este tutorial, que es donde tú, que estás entre el público, dices aquello de *Pues yo hago todo eso, y más, en (Fortran|Postscript|Haskell)*. Que vale, que si. Los lenguajes de programación son universales. Se puede hacer de todo con ellos. Y siempre es más fácil hacer algo en el lenguaje que uno conoce mejor. Pero al menos tendrás más donde elegir, ¿no?

Finalmente, aunque no es imprescindible, es conveniente que tengas un ordenador enfrente, y que puedas usarlo. La mayoría de los ordenadores modernos, y muchos de los antiguos, tienen versiones de Perl compiladas. La Nintendo DS todavía no, pero todo se andará.

Sobre todo, que no cunda el pánico. Y no te olvides de la toalla (http://es.wikipedia.org/wiki/Dia_de_la_Toalla).

2. Todo listo para despegar

> Si ya has usado algún lenguaje de scripting, lo más probable es que te aburras como un bivalvo en esta sección. Así que ahórrate un bostezo y pasa directamente a la siguiente. O si no descárgate los fuentes (http://www.cpan.org/src/latest.tar.gz) y echas un ratillo compilándolos en silencio, para no desmoralizarme a la parroquia.

Lo primero que necesitas en tu lista de comprobación son las cualidades de todo programador en Perl: la pereza, el orgullo y la curiosidad. No te preocupes si no tienes ninguna de ellas, las irás adquiriendo con el tiempo. Sobre todo la pereza. Y una cierta habilidad de entender lenguas muertas como el caldeo y el dálmata.

Segundo, necesitas amar a los camélidos. El Perl no es como esos otros lenguajes que incitan a la avaricia a través de la adoración de las piedras preciosas (http://www.ruby-lang.org), o a la hiperactividad por ingestión de bebidas excitantes (http://www.java.com). Los camellos son buenos. Los camellos son útiles. Llevan cosas encima. Tienen joroba. Amemos a los camélidos (las llamas también son camélidos (http://es.wikipedia.org/wiki/llama)).

No menos importante es tener un ordenador con sistema operativo. Incluso sin él (http://www.windows.com). Ejecuta lo siguiente para saber si lo tienes: **`perl -v`** a lo que el ordenador debidamente contestará algo así:

Figura 1. Contestación de un ordenador educado a `perl -v`

```
This is perl, v5.8.7 built for i486-linux-gnu-thread-multi
(with 1 registered patch, see perl -V for more detail)

Copyright 1987-2005, Larry Wall

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

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

si es que está instalado. Si no lo está, es poco probable que conteste eso. Incluso imposible. Dirá algo así como `bash: perl: command not found` e incluso pitará. El muy desagradable.

No hay que dejarse descorazonar por tal eventualidad. Encomendándonos al *Gran Camélido*, y sin necesidad de ver una vez más Ishtar, diremos en voz alta "Abracadabra" mientras que escribimos **`sudo yum install perl`** o bien **`sudo apt-get install perl`** Si es que están en un linux no-debianita (en el primer caso) o en uno debianita (en el segundo). Habrá gente que incluso lo haga sin necesidad de bajarse del ratón. Pero los apresurados no usan el ratón salvo que sea estrictamente necesario. Que no es el caso. En otros sistemas operativos, lo mejor es ir a Perl.com (si es que no has ido todavía) (http://www.perl.com) y te bajes la versión compilada.

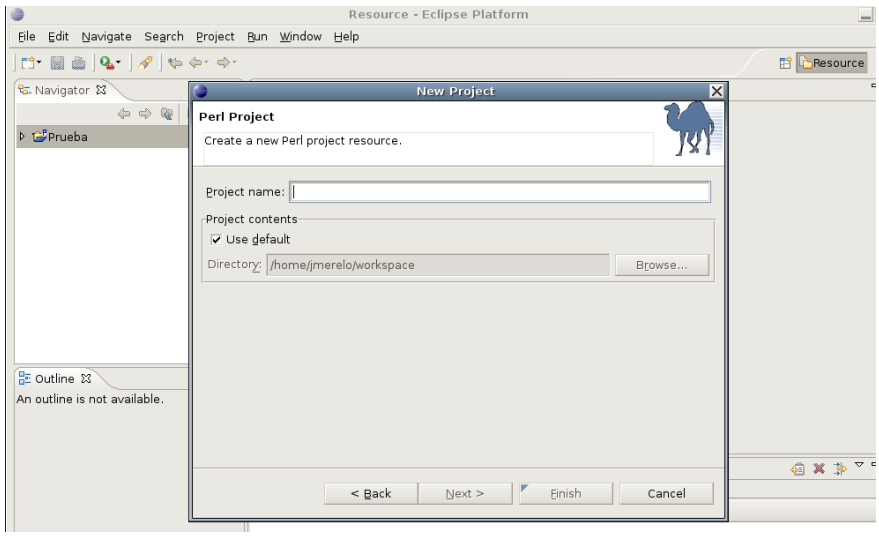
También puedes compilarlo tú. Pero no creo que lo hagas, porque eres un apresurado, y la compilación no está hecha para los apresurados (si eres usuario de Gentoo (http://www.gentoo.org), es el momento de abandonar este tutorial).

Lo que tienes o has instalado es un intérprete de Perl. Perl es generalmente un lenguaje interpretado, con lo que no hace falta ningún encantamiento intermedio para pasar de un programa escrito en Perl a su ejecución. Si te hará falta un editor. No *un* editor. *El* editor.

> **Sugerencia:** Los que apoyen al ínclito (x)emacs de este lado del *flamewar*, los que se queden con el sólido pero escuálido vi(m), de este otro lado. Los que estén con kate, jot, o incluso el kwrite, que elijan armas y padrinos y que pidan hora.

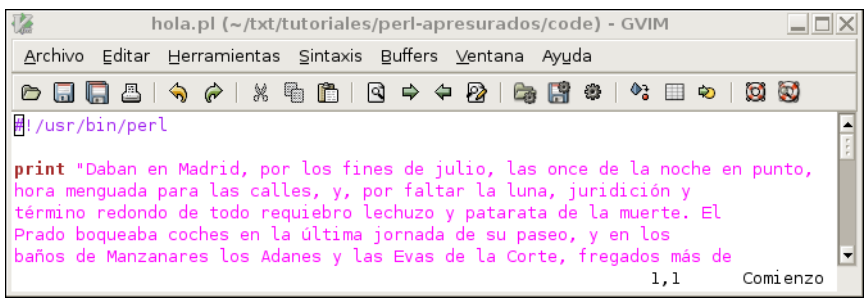
Vuelvo contigo entre el fragor de la batalla hablarte de otras opciones. No es que haya muchas, pero hay alguna. Por ejemplo, puedes usar el conocido entorno Eclipse (http://eclipse.org) con el plugin EPIC (http://e-p-i-c.sourceforge.net) para desarrollar proyectos, como se muestra en la figura siguiente.

Figura 2. Iniciando un proyecto en Perl con EPIC/Eclipse



Otros entornos de desarrollo, como PerlIDE o Komodo, o bien no siempre funcionan o bien son de pago. Si consigues que te lo compren, suertudo de ti. Si no, apoya proyectos de software libre. Suficientes personas han estado desarrollando sobre esos entornos durante el suficiente tiempo como para que presenten la sana apariencia que se muestra en la figura de abajo.

Figura 3. Editando un programilla con gvim



Un editor decente tiene que tener colorines. Y también cerrar paréntesis. Ninguno va a evitar que cometas errores, pero va a hacértelo lo más complicado posible.

Ejercicios. ¿Tienes un intérprete de Perl instalado en tu sistema? ¿Tienes un editor (chulo, si es posible) para editar programas en Perl? Si la respuesta a alguno de ellos es *no*, ¿a qué esperas para tenerlos? Venga, te espero.

3. Comenzando una nueva carrera

Si has llegado hasta aquí, supongo que se te llevarán todos los diablos, porque con la hora que es, las camas están sin hacer y lo que s dice picar código, todavía no has picado nada. Y eso está bien: hay que convertir esa rabia en energía creativa, y aprovechando que uno de los diablos que se te llevan es cojuelo, escribir el siguiente fragmento de literatura:

```
#!/usr/bin/perl                                                    ❶
print "Daban en Madrid, por los fines de julio, las once de la noche en punto..."; ❷
print "\n";                                                        ❸
```

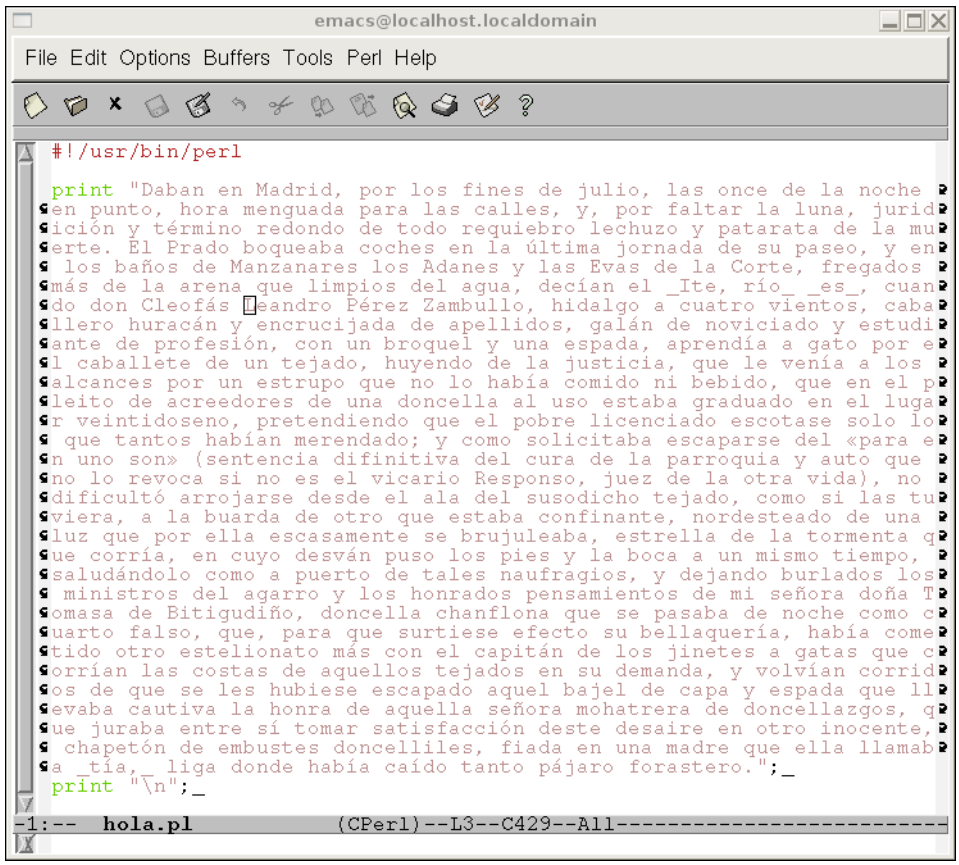
❶ Tratándose de diablos, lo mejor es usar los conjuros lo antes posible. En esta línea, clásica de los lenguajes interpretados y denominada shebang se escribe el camino completo donde se halla el intérprete del lenguaje en cuestión. Si está en otro sitio, pues habrá que poner otro camino. Por ejemplo, podría ser **#!/usr/local/bin/perl** o bien **#!/usr/bin/perl6.0.por.fin**. O **#!perl** y que se busque la vida. Si se trabaja (es un decir) en Windows, esa línea no es necesaria, pero es convenientepara que el programa sea compatible con otros sistemas operativos. Cuando un Unix/GNU/Linux decente y trabajador encuentra esa línea, carga ese programa y le pasa el resto del fichero para que lo interprete.

❷ Aquí se imprime, con el *nihil obstat* obtenido previamente. Obsérvense las comillas y el punto y coma. Las órdenes en Perl acaban con un punto y coma, para que quede bien claro dónde acaban y se puedan meter varias sentencias en una sola línea, con el objetivo de crear programas innecesariamente ofuscados. Lo que no se puede hacer en *esos otros lenguajes*. El `print` es herencia de aquellos primeros tiempos de los ordenadores, cuando el único periférico de salida era un convento de monjes trapenses dedicados a la sana tarea de copiar textos (y que se quedaron sin trabajo cuando el señor Hewlett se unió al señor Packard y crearon la impresora). En aquella época, la salida de un programa venía encuadernada en piel de cabrito y con todas las primeras letras de párrafo bellamente miniadas. Ah, tiempos aquellos, de los que sólo nos queda el nombre de una orden.

❸ Y aquí pasamos a la línea siguiente. Borrón y cuenta nueva. Se acabó lo que se daba. Si ya conoces algún lenguaje de programación, que se supone que lo conoces, pillín, porque te lo he preguntado en la primera sección, no hace falta que te explique que `\n` es un *retorno de carro*, ¿verdad?[1]

Desde un editor que cambie el color (y los tipos de letra) de acuerdo con la sintaxis del programa que se está editando tal como el `emacs`, el resultado debería ser algo similar al que aparece en la captura siguiente

Figura 4. Editando hola.pl en emacs



Aviso

El usar este tipo de texto, que incluye caracteres con acento, es bastante intencionado. En algunos editores puede que aparezcan caracteres extraños; habrá que cambiar la *codificación* para que entienda el conjunto de caracteres `iso-8858-1` o `latin1`.

Ejercicios. Elegir un editor no es un tema baladí, porque te acompañará en tu vida como desarrollador. Prueba diferentes editores disponibles en tu sistema mirando sobre todo a las posibilidades que tienen de adaptación a diferentes cometidos (comprobar la sintaxis y depurar, por ejemplo). Nadie te ata a un editor de por vida, pero cuanto antes lo elijas, antes empezarás a ser productivo. Así que ya estás empezando a usar el (x)emacs.

4. Viéndole las tripas al producto

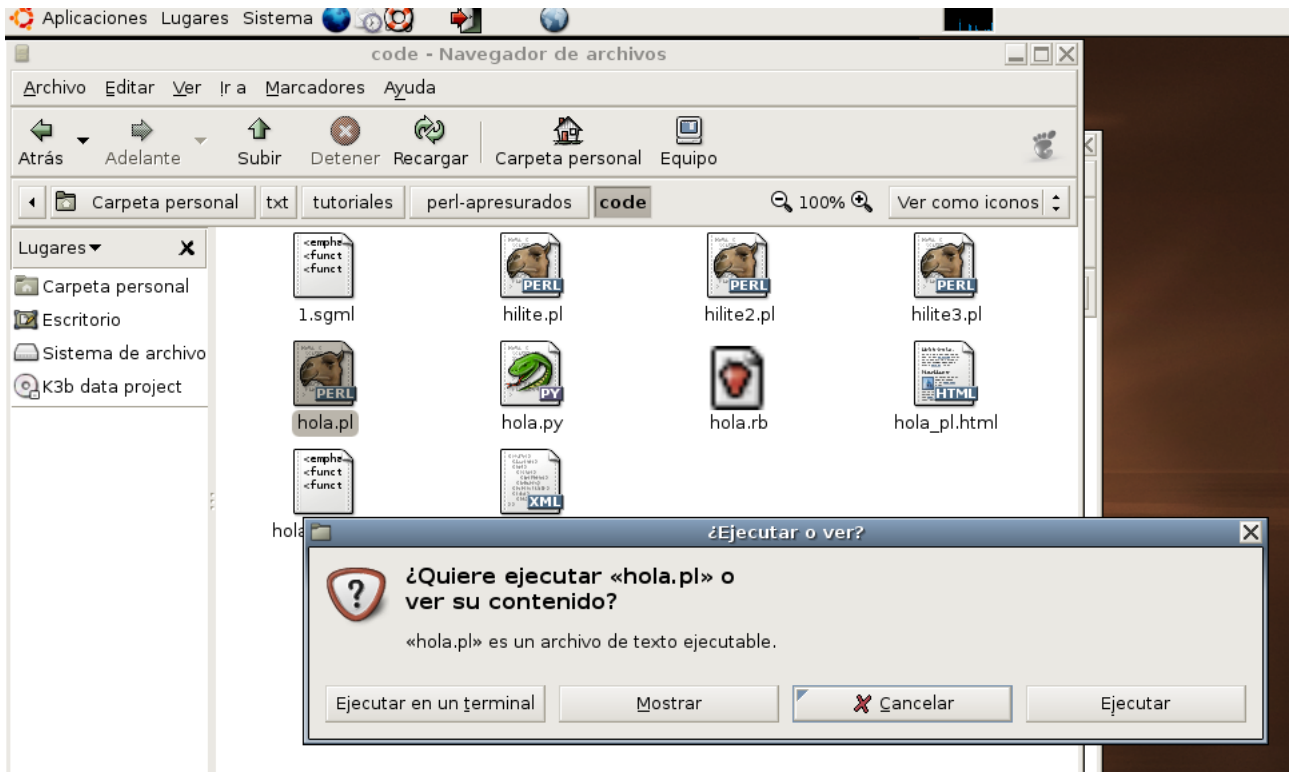
Mucho editar, mucho editar, pero de ejecutar programas nada de nada. Lo que hemos editado no deja de ser un fichero de texto, así que para ejecutarlo tendremos que llevar a cabo algún encantamiento para meterlo en el corredor de la ejecución.

Tampoco hace falta. Lo más universal es irse a un intérprete de comandos, colocarse en el directorio donde hayamos salvado el fichero, y escribir **`perl hola.pl`**. Pero ya que estás puesta, puedes hacer algo más: escribir **`chmod +x hola.pl`**, lo que convertirá al fichero en ejecutable, es decir, en algo que podemos correr simplemente tecleando su nombre, de esta forma:

```
jmerelo@vega:~/txt/tutoriales/perl-apresurados/code$./hola.pl
Daban en Madrid, por los fines de julio, las once de la noche en punto, hora menguada...
```

Pero el encantamiento este actúa también a otros niveles, pudiendo ejecutar el programa directamente desde esos inventos del averno llamados *entornos de ventanas*, como se muestra en la figura siguiente.

Figura 5. Ejecutando un programa en Perl desde Gnome



Como el Nautilus, el manejador de ficheros de Gnome es muy listo, se dice a si mismo (pero bajito):

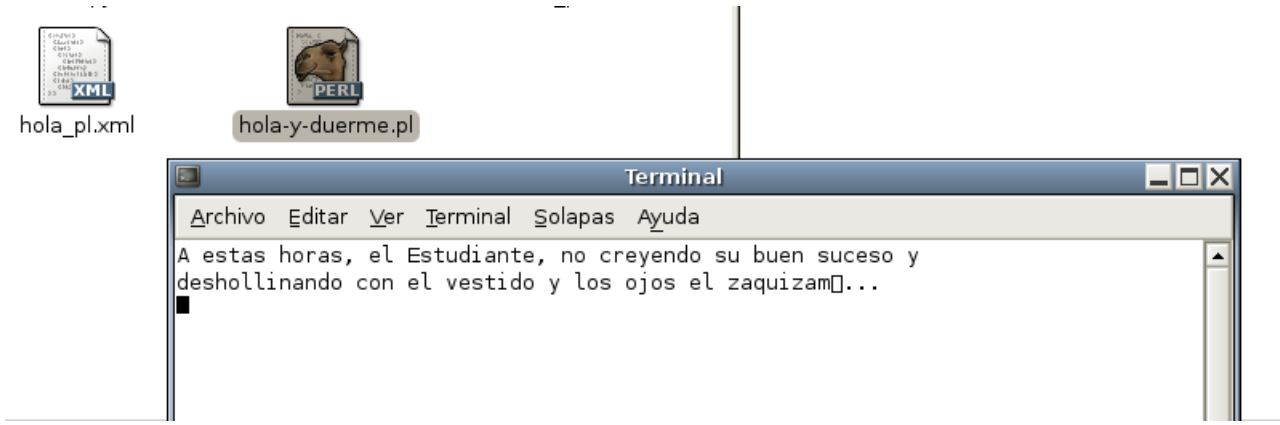
Pardiez, este fichero es ejecutable. ¿Qué puedo hacer con él? ¿Lo ejecuto? ¿Lo abro sin ejecutarlo? La duda me carcome. Se lo preguntaré al honorable usuario. El menú contextual (con el botón derecho del ratón) nos ofrecerá opciones similares. El problema es que si lo ejecutamos será visto y no visto.

Vamos a dejar entonces que el programa se quede clavado hasta nueva orden, con una pequeña modificación, que aparece en el siguiente listado

```
#!/usr/bin/perl
print "A estas horas, el Estudiante, no creyendo su buen suceso y
deshollinando con el vestido y los ojos el zaquizamí...\n";
sleep 10;
```

Que, con un poco de suerte, nos permitirá capturar una pantalla como la siguiente:

Figura 6. Terminal con el resultado de ejecutar el programa `hola-y-duerme.pl`



En otros sistemas operativos, cambiará el icono y la apariencia del terminal donde está el resultado, pero por lo demás, el resultado será el mismo. La única diferencia con el primer programa es la última línea, que le indica al programa que se quede quieto parao (dormido, de hecho) durante 10 segundos. Y que diga *cheeeeeese* (solo en ordenadores con interfaz gestual y/o emocional y/o audiomotriz/parlanchín).

Pero incluso así, puede que sea demasiado rápido para apreciar la sutileza de cada una de las órdenes, y haya que ejecutarlo paso a paso. Más adelante tendrás que *depurar* tus programas, porque cometerás errores, si, errores y tendrás que corregirlos sobre la marcha. De la forma más inteligente, además. Pero no hay que preocuparse, porque Perl tiene un depurador integrado. Ejecuta el programa de esta forma:

```
jmerelo@vega:~/txt/tutoriales/perl-apresurados/code$
perl -d hola-y-duerme.pl
Loading DB routines from perl5db.pl version 1.28
Editor support available.
```

```
Enter h or ‘h h’ for help, or ‘man perldebug’ for more help.

main::(hola-y-duerme.pl:3):     print "A estas horas, el Estudiante, no creyendo su buen suceso y
main::(hola-y-duerme.pl:4):     deshollinando con el vestido y los ojos el zaquizamí..\n";
  DB<1>
```

La opción **-d** del intérprete te introduce en el depurador, así, sin más prolegómenos. A partir de esa línea de comandos, puedes evaluar las expresiones de Perl que quieras, y, por supuesto, depurar el programa, ejecutándolo paso a paso, mirando variables, y todo ese protocolo inherente al mester de la programación. Para empezar, vamos a ejecutarlo pasito a pasito.

```
DB<1> R
Warning: some settings and command-line options may be lost!

Loading DB routines from perl5db.pl version 1.28
Editor support available.

Enter h or ‘h h’ for help, or ‘man perldebug’ for more help.

main::(hola-y-duerme.pl:3):     print "A estas horas, el Estudiante, no creyendo su buen suceso y
main::(hola-y-duerme.pl:4):     deshollinando con el vestido y los ojos el zaquizamí..\n";
```

, lo que empieza a hacerse ya un poco repetitivo. La orden **R** comienza a ejecutar el programa. En realidad, antes lo único que habíamos hecho es indicarle (educadamente) al depurador el programa que íbamos a depurar; ahora es cuando lo estamos ejecutando en serio. Bueno, todavía no, porque en este punto todavía no hemos ejecutado ni siquiera la primera línea. La salida del depurador nos indica <`main::(hola-y-duerme.pl:3):` la siguiente línea del programa (`3`) que vamos a ejecutar (y la 4 de camino, que para eso la orden ocupa dos líneas).

```
  DB<0> n                                                                    ❶
A estas horas, el Estudiante, no creyendo su buen suceso y
deshollinando con el vestido y los ojos el zaquizamí..
main::(hola-y-duerme.pl:5):     sleep 10;
```

❶ **n**, de *next*, siguiente, ejecutal la línea siguiente, es decir, justamente la que aparece al final de el ejemplo anterior

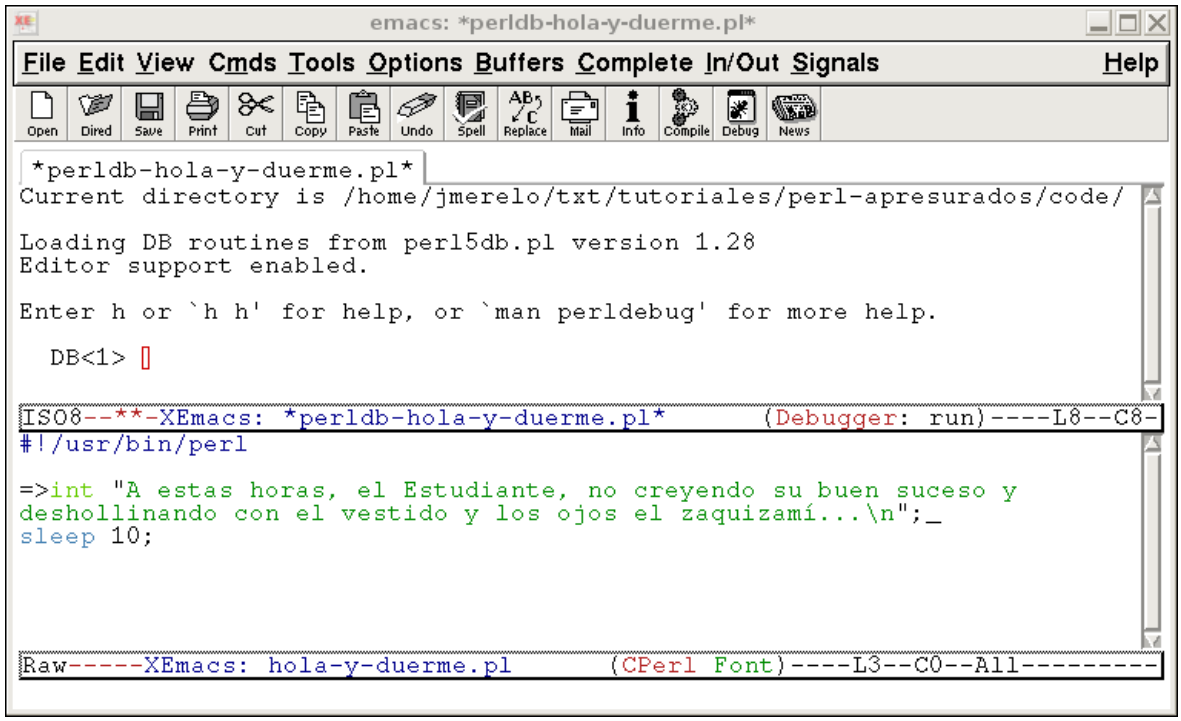
❷ Ésta es la salida de esa línea en particular; lo que hace es escribir lo que se encuentra entre las comillas.

❸ Y muestra la línea siguiente a ejecutar.

Como persona precavida vale por dos diablillos, no es mala idea tener siempre el depurador abierto para ir probando cosas. Te ahorrará más de una vuelta al editor a reescribir lo que ya está escrito. Además, es muy fácil. Si has elegido Un Buen Editor (o sea, el XEmacs), y te ha reconocido el programa como un

fichero Perl, tendrás una opción del menú llamada **perl**; desplegando ese menú, te aparecerá la opción **debugger**, eligiéndola te dará un resultado similar al que se muestra en la siguiente captura de pantalla:

Figura 7. Depurando un programa en el mismo editor XEmacs. La flecha está situada sobre la siguiente línea a ejecutar.



Desde este depurador se trabaja de la misma forma que en la versión de la línea de comandos, pero se pueden colocar puntos de ruptura usando el ratón, por ejemplo, y puedes ver las líneas que se están ejecutando en su contexto.

Con esto, ya estamos listos para abordar empresas más elevadas, y que nos llevarán mucho más lejos.

> **Ejercicios.** Familiarizarse con el depurador, creando un programa con las dos o tres cosas que se conocen, y viendo las diferentes órdenes; por ejemplo, cómo ejecutar un programa sin parar, o hasta una línea determinada, y cómo hacer que la ejecución se pare en una línea determinada. Recuerda, **h** es tu amiga.

5. Usando la sabiduría colectiva

Escribir está bien. Hay dos o tres personas que incluso se ganan la vida con ello[2]. Pero hace falta hacer algo más. Copiar a Faulkner, por ejemplo. Pero no sólo copiarlo. Ser más Faulkner que Faulkner. O mezclar Faulkner con, pongamos por caso, David Sedaris. O quizás Hemingway con Sedaris. Y llegado a este punto, te voy a contar un secreto. No hace falta que programes absolutamente nada. Ya hay gente que ha hecho lo que tú piensas programar en este preciso instante. De hecho, un vietnamita y un chavalote de Mondoñedo que acaba de terminar un módulo de FP segundo grado. Pero ambos dos son buenas personas, y legan su trabajo a la humanidad toda (inclusive tú). Si hay una sola cosa que haga al Perl superior a otros lenguajes de programación, son esas cosas que ha hecho la gente, empaquetado y colocado en un sitio común, llamado CPAN (http://search.cpan.org). CPAN significa, como probablemente ya habías adivinado, *comprehensive Perl Archive Network*, y es un sitio donde hay cienes, qué digo cienes, millardos de módulos que hacen todas esas cosas que se te hayan podido ocurrir, y otras cuantas que, ni harto de vino, se te podrían haber ocurrido. Pero hay que saber usarlo, claro.

> **Nota:** Si has tenido que pedirle a alguien que te instale el Perl, posiblemente sea el momento de que tengas a mano otra vez su teléfono o móvil, porque vas a volver a necesitarlo. No ahora. Más tarde. Mientras tanto, aunque no sea el día de apreciación del administrador del sistema (http://www.sysadminday.com/), aprovecha para pensar en él con cariño. Antes de que la falta de calor humano lo convierta en un operador bastardo del infierno (http://es.wikipedia.org/wiki/Bofh). Para instalar módulos de CPAN para que sean accesibles para todo el mundo hace falta tener privilegios de operador; sin embargo, puedes instalarlos sin problemas en tu propio directorio (por ejemplo, en `/home/miusuario/lib/perl`).

En CPAN hay módulos para todo. En particular, para manejar textos en diferentes idiomas. Por ejemplo, un módulo para dividir en sílabas texto en castellano llamado Lingua::ES::Silabas (http://search.cpan.org/~marcos/Lingua-ES-Silabas-0.01/). Un módulo es simplemente una biblioteca de utilidades para un fin determinado (o ninguno) escritas en Perl, o, al menos, empaquetadas para que se pueda acceder a ellas desde un programa en Perl. Una librería crea una serie de funciones a las que podemos acceder desde nuestros programas. Pero antes hay que instalarla. Y antes todavía, hay que ejecutar CPAN por primera vez:

```
jmerelo@vega:~$ sudo cpan
cpan shell -- CPAN exploration and modules installation (v1.83)
ReadLine support enabled

cpan>
```

Si realmente es la primera vez que lo ejecutas, te preguntará una serie de cosas. En la mayoría es razonable contestar la opción que te ofrezcan por defecto, pero en un par de ellas si tienes que elegir:

- Si no tienes privilegios de superusuario, tendrás que elegir un subdirectorio alternativo para colocar los módulos instalados.
- Es conveniente usar los repositorios más accesibles desde tu país, y por orden de frecuencia de actualización, para tener garantía de frescura de los módulos. Por ejemplo, dos buenas opciones

pueden ser http://debianitas.net/CPAN/ y http://cpan.imasd.elmundo.es/; aunque los otros repositorios con la extensión **.es** también suelen funcionar relativamente bien.

Una vez configurado todo, ya se puede instalar el módulo susodicho. Lo puedes hacer directamente desde la línea de comandos con

```
install Lingua::ES::Silabas
CPAN: Storable loaded ok
LWP not available
Fetching with Net::FTP:
  ftp://ftp.rediris.es/mirror/CPAN/authors/01mailrc.txt.gz
Going to read /home/jmerelo/.cpan5.9.3/sources/authors/01mailrc.txt.gz
CPAN: Compress::Zlib loaded ok
LWP not available
[...más cosas...]
Fetching with Net::FTP:
  ftp://ftp.rediris.es/mirror/CPAN/authors/id/M/MA/MARCOS/CHECKSUMS

  CPAN: Module::Signature security checks disabled because Module::Signature
  not installed.  Please consider installing the Module::Signature module.  You may also need to be al
  keyservers like pgp.mit.edu (port 11371).

Checksum for /home/jmerelo/.cpan5.9.3/sources/authors/id/M/MA/MARCOS/Lingua-ES-Silabas-0.01.tar.gz ok
Scanning cache /home/jmerelo/.cpan5.9.3/build for sizes
Lingua-ES-Silabas-0.01/
Lingua-ES-Silabas-0.01/Silabas.pm
Lingua-ES-Silabas-0.01/README
Lingua-ES-Silabas-0.01/Makefile.PL
Lingua-ES-Silabas-0.01/Changes
Lingua-ES-Silabas-0.01/MANIFEST
Lingua-ES-Silabas-0.01/test.pl

  CPAN.pm: Going to build M/MA/MARCOS/Lingua-ES-Silabas-0.01.tar.gz

Checking if your kit is complete...
Looks good
Writing Makefile for Lingua::ES::Silabas
cp Silabas.pm blib/lib/Lingua/ES/Silabas.pm
Manifying blib/man3/Lingua::ES::Silabas.3
  /usr/bin/make  -- OK
Running make test
PERL_DL_NONLAZY=1 /usr/local/bin/perl5.9.3 "-Iblib/lib" "-Iblib/arch" test.pl
1..9
# Running under perl version 5.009003 for linux
# Current time local: Mon Jul 10 23:34:36 2006
# Current time GMT:   Mon Jul 10 21:34:36 2006
# Using Test.pm version 1.25
ok 1
ok 2
ok 3
ok 4
```

```
ok 5
ok 6
ok 7
ok 8
ok 9
  /usr/bin/make test -- OK
Running make install
Installing /usr/local/lib/perl5/site_perl/5.9.3/Lingua/ES/Silabas.pm
Installing /usr/local/share/man/man3/Lingua::ES::Silabas.3
Writing /usr/local/lib/perl5/site_perl/5.9.3/i686-linux-thread-multi-ld/auto/Lingua/ES/Silabas/.packl
Appending installation info to /usr/local/lib/perl5/5.9.3/i686-linux-thread-multi-ld/perllocal.pod
  sudo make install  -- OK
```

que, efectivamente, descarga el módulo del repositorio espejo de CPAN más cercano (en este caso ftp.rediris.es), lo "compila", hace una serie de tests (sin los cuales no se instalaría siquiera), y efectivamente lo instala para que esté disponible para todos los programas que quieran usarlo (que no creo que sean muchos, pero alguno puede caer).

Pero, ¿andestá la documentación? se preguntará el preocupado (aunque apresurado) lector. No preocuparse. Todo módulo en CPAN está documentado, y se accede a él usando el mismo sistema que se usa para todo Perl: el programa `perldoc`. En este caso, **`perldoc Lingua::ES::Silabas`** nos devolverá algo así:

```
Lingua::ES::Silabas(3)User Contributed Perl DocumentatioLingua::ES::Silabas(3)



NOMBRE
       Lingua::ES::Silabas - Divide una palabra en silabas

SINOPSIS
         use Lingua::ES::Silabas;

         $palabra = âexternocleidomastoideoâ; # muchas silabas ;-)

         ## en contexto de lista,
         ## lista de silabas que componen la palabra
         @silabas = silabas($palabra);

         ## en contexto escalar,
         ## el numero de silabas que componen la palabra
         $num_silabas = silabas($palabra);
```

Vamos a ver ahora como se usa ese pozo de sabiduría para escribir un programa que toma una serie de frases, y las escribe divididas en sílabas.

```
use Lingua::ES::Silabas;                                                    ❶

print join(" / ", silabas(join("", split(/[\s,]/,<<EOC)))), "\n";                ❷
```

```
Yo soy, señor Licenciado, que estoy en esta redoma, adonde me tiene❸
preso ese astrólogo que vive ahí abajo, porque también tiene su punta
de la mágica negra, y es mi alcaide dos años habrá.
EOC
```

❶ Para empezar, que no cunda el pánico. Si estamos a Perl, estamos a Perl, y hay que ver cómo Perl hace las cosas de forma diferente a cualquier otro lenguaje de programación. Dicen que Perl no es amistoso para el usuario, pero lo cierto es que Perl *elige a sus amigos*. Así que es cuestión de ver a Perl como a un Calimero cualesquiera, y entender su peculiaridades, como las que aparecen aquí. Bueno, no aquí, más abajo. Por lo pronto, sólo que no cunda el pánico.

En esta línea lo único que se hace es cargar la librería que antes, debidamente, hemos instalado. Y quizás merezca la pena señalar un poco la estructura del nombre. Todos los módulos en CPAN están organizados en espacios de nombres, para hacer más fácil su búsqueda y evitar colisiones de funcionalidad (y de nombre también). En este caso, el espacio de nombre es el `Lingua`, que incluye muchos más módulos cada vez más esotéricos. Pero este espacio está bien organizado, porque luego vienen un par de caracteres que indican a qué lengua se aplica el módulo susodicho; en este caso, `ES`. Finalmente, el último apartado es el realmente específico.

> **Nota:** En cada sistema de ficheros específico, el nombre también indica en el directorio en el que estará almacenado, dentro de los directorios donde se suelen almacenar los módulos.

Si te das cuenta, no ha habido que especificar dónde diablos se busca ese módulo para incorporarlo al programa. Mágicamente, el intérprete de Perl busca que te busca en todos los directorios razonables, hasta que lo encuentra. Pero aunque mágico, no es telepático, y no sabe donde tú, precisamente, has podido instalar ese módulo. Especialmente si no tienes privilegios de superusuario y has tenido que instalarlo en tu `${HOME}`, tendrás que especificarle en qué directorio buscar, de esta forma:

```
use lib "/home/esesoyyo/lib/perl";
use Lingua::ES::Silabas;
```

Es decir, *antes* de tratar de cargar el módulo.

❷ Aquí viene un embutido de diferentes órdenes en Perl, que, como suele ser habitual, es mejor leerlas de derecha a izquierda. O del punto y coma para el otro lado. Porque no me voy a poner a explicar la derecha de quién quiero decir. Lo del `\n` ya no lo sabemos, al menos, desde aquel primer párrafo del primer tranco del diablo cojuelo, pero lo que sigue es un poco más raro. Y requiere un punto y aparte.

Aunque Perl sabe que existe una cosa llamada sintaxis, no se la toma muy en serio. Las cosas se pueden escribir de muchas formas diferentes. De hecho, hasta los programas en perl pueden cambiar su sintaxis, su semántica, y hasta su prosodia. Encontrarse con comas, o sin ellas, no debe inducir al pasmo, sino a una profunda reflexión. Así que a partir de ahora, agárrense a su toalla y prepárense a olvidarse de cualquier concepción rígida de lo que es la sintaxis de un lenguaje de programación

Aviso

Y es aquí es donde los programadores de COBOL y FORTRAN dejan este tutorial y vuelven definitivamente a su hogar del pensionista al jugar al Julepe.

Y lo que nos encontramos que ofende a nuestra conciencia sintáctica es `<<EOC`. O sea, la leche. Uno espera encontrarse una cadenita, tan mona, con sus comillas en los extremos, y se encuentra esto. Que no es otra cosa que un *docaquí*[3], como los más avispados ya habrán averiguado. Sirve para lo siguiente: cuando la cadena es muy larga, y ocupa varias líneas, es mejor escribirla tal cual. Así que en vez de meterla entre incómodas comillas, se usa esta construcción: `<<CADENA`, donde `CADENA` es cualquier cosa que no aparezca dentro y que indique el fin de la misma. Cuando el intérprete se encuentra esa construcción, comienza a leer en la línea siguiente y sigue hasta que se encuentra el conjuro de fin de texto; `EOC` en este caso. ¿Lo cogen?

Pero no hay descanso para los intrépidos, e inmediatamente nos encontramos con `split(/[\s,]/,`. Empecemos por el `split` esta vez: split divide una cadena, es decir, la convierte en una serie (por ahora lo dejamos ahí) de palabras, eliminando los separadores, que son los que están entre corchetes (y lo dejamos ahí también). `split` recorre la cadena, se encuentra un espacio (`\s`) o una coma y desgaja un eslabón. La siguiente, un eslabón. Al final, lo que se encuentra uno es una serie de eslabones, sin nada de morralla enmedio. Que es lo que precisamente unimos (`join`) usando la cadena nula. Es decir, nos vamos a encontrar una sola cadena sin espacios ni comas. Una gran palabra, un abracadabra diabólico que contiene todas las palabras de las frases anteriores. Y ese abracadabra es el que le damos a la humilde función que hemos importado del módulo previamente, *silabas*, que precisamente nos devolverá una serie de eslabones, que volveremos a unir para dar el resultado esperable:

```
Yo / soy / se / ñor / Li / cen / cia / do / que / es / to / ye / nes / ta / re / do / ma / a / dor
```

. Vale, no es perfecto. Pero es que llevamos una hora (o así) apresurada de aprender Perl, tampoco se le pueden pedir peras al olmo.

❸ Y este es precisamente el texto que vamos a dividir. ’Nuff said.

Ejercicios. Hay gigas y gigas de módulos, a cada cuál más útil y sorprendente. Los módulos `Acme::`, por ejemplo, son absolutamente inútiles, y no tienen equivalente en ningún otro lenguaje. Instalar el módulo `Acme::Clouseau` por ejemplo, y ejecutar el programa de prueba que se incluye en su *Sinopsis*.

6. Ley de Murphy

Errar es humano, sobre todo cuando uno tiene tantos dedos, y tan pocas teclas y se juntan todas para que

uno se equivoque. Y tampoco puede acordarse uno de la sintaxis de un lenguaje que no tiene sintaxis. Sería una paradoja que daría lugar al fin del universo tal como lo conocemos. Así que uno se equivoca. Pero como llegados los 67 minutos de este Perl para apresurados tampoco sabe uno mucho (y se nos echa el tiempo encima) tampoco puede equivocarse uno mucho. Pero hay un par de errores que se cometen con bastante asiduidad. El primero puede ser algo así:

```
#!/usr/bin/pelr

print "Da igual, porque va a petar";
```

que, al ejecutarse, da un sorprendente:

```
jmerelo@vega:~/txt/tutoriales/perl-apresurados$
code/peta1.pl
bash: code/peta1.pl: /usr/bin/pelr: bad interpreter: No existe el fichero o el directorio
```

, que viene a decir que vale, que muy bien, pero que ese intérprete no existe. Cambiar alguno de los otros caracteres, la admiración (que es una expresión de admiración al creador del programa encarnado en el intérprete), la almohadilla, produce errores igualmente pintorescos. Por ejemplo, quitar la admiración o la almohadilla da este:

```
Warning: unknown mime-type for "Da igual, porque va a petar" -- using "application/*"
Error: no such file "Da igual, porque va a
petar"
```

Este error aparecerá también si pasas un fichero de Windows (que incluyen al final de línea dos caracteres, retorno de carro y fin de línea) a Unix/Linux/GNU (que incluye uno solo). El intérprete de órdenes, tan agudo en otras ocasiones, en esta ocasión interpretará el carácter extraño (representado con ^M en los editores) como parte del nombre, y dará el mismo tipo de error.

El segundo tipo de error, y el más frecuente, se produce una vez que el intérprete se ha cargado correctamente, por ejemplo en el situiente programa:

```
print "Da igual, porque va a petar\n"
print "Pero solo si el error no está en la última
línea\n";
```

que hace que el intérprete responda con un informativo

```
syntax error at code/peta2.pl line 4, near "print"
Execution of code/peta2.pl aborted due to compilation errors.
```

y todo eso, por un humilde punto y coma. Si Guido van Rossum levantara la cabeza.

Evidentemente, otros errores de sintaxis darán su mensaje correspondiente. En general, serán bastante más informativos. Y, en todo caso, Google es tu amigo.

> **Ejercicios.** Este bloque no tiene ejercicios. No te voy a pedir que escribar un fichero con errores y digas *Um, parece que tiene un error*. Pero acuérdate de esta sección, te será util. O no.

7. Lo escrito, escrito está

"Que si, que mucho Perl para apresurados y mucha gaita gallega, pero mira la hora que es y no nos hemos comido un jurel", estará diciendo a estas alturas el (apresurado) lector. Tenga paciencia vuecencia, que sin pausa pero sin prisa, todo llegará. En particular, llega que, no conforme con escribir cosas que estén *dentro* de un programa, alguien quiera escribir algo que esté, por el contrario, fuera de un programa. Y para más inri, con spoilers traseros y todo: añadiéndole números de línea. Pues para eso estamos (está Perl):

```
my $leyendo = shift                                                    ❶
  || die "Uso: $0 <nombre de fichero>\n";                                    ❷
open my $fh, "<", $leyendo                                                  ❸
  or die "No puedo abrir el fichero $leyendo por $!\n";while (<$fh>) {
  print "$.$_";
}close $fh;                                                          ❹
```

❶ Vamos a dejar las cosas claras desde el principio. `$leyendo` es una variable. Una variable cara, porque lleva el dólar al principio. Y por eso es mía, y le he puesto el `my`. Sólo mía. Bueno, de hecho, es una variable de ámbito léxico, que será invisible fuera del bloque. Este bloque abarca todo el programa. Pero puede no hacerlo; los bloques están encerrados entre llaves (`{}`). Hay uno un poco más abajo.

Y en cuanto a la variable propiamente dicha, es una variable escalar. Por eso lleva un dólar delante, porque el dinero también es escalar. Creo. En todo caso, en las variables escalares puede haber números, o cadenas alfanuméricas; a Perl le da igual. La interpretará como uno, o como otro, dependiendo del contexto.Por ejemplo

```
$foo="13abc"
print $foo + 1
14
```

En realidad, no es necesario declarar las variables. Cuando se usa una variable por primera vez, aparece automágicamente con valor nulo, pero el ámbito será global. Y las variables globales son *una mala cosa*.

Lo que viene después, aunque no lo parezca, es el primer argumento que se le pasa al programa por la línea de comandos. En realidad, `shift` saca el primer elemento de una ristra, en este caso, la ristra de nombres de ficheros que se le haya pasado. ¿Y qué pasa si ejecutamos el programa tal cual, sin pasarle ningún nombre de fichero?

❷ Pues que el programa irremisiblemente muere (`die`). Pero no muere por las buenas, sino que te da un mensaje que te indica que, obligagoriamente, bajo pena de muerte (`die`) tienes que pasar el nombre de un fichero como argumento, de esta forma:**`escrito.pl diablo-cojuelo.txt`**. Si no lo hacemos:

```
./escrito.pl
Uso: ./escrito.pl <nombre de fichero>
```

Pero no hemos dicho nada de los `||` al principio de la línea. Como esto es un lenguaje decente, para terminar una sentencia hace falta el `;`, que no está hasta el final de esta línea. Con lo que esta línea y las anteriores vienen a decir "Esta variable será el primer argumento, *o si no* me muero y dejo este mensaje" Ese *o si no*, `||`, que representa a `OR`[4]hace que se ejecute su parte derecha solo si la parte izquierda tiene como resultado un valor verdadero (será falso si `shift` devuelve `undef`, es decir, si no hay nada en la línea de comandos.

Esta construcción viene a ser un condicional, y el caso más general es <expresión> <condicional> <sentencia>. Lo veremos mucho.

Y de camino hemos visto como funciona la interpolación de variables en cadenas en Perl: una variable dentro de una cadena se sustituirá por su valor al ejecutar el programa. Y no es una variable cualquiera. Es una variable predefinida, que son variables globales cuyo valor depende de ciertas circunstancias. En este caso, la cadena con la que se ha invocado al programa.

❸ Habrá que abrir (`open`) el fichero, claro. Eso es lo que hacemos aquí. A `open` se le pasa la variable que vamos a usar para referirnos al fichero abierto[5], que declaramos sobre la marcha, el modo de apertura, en este caso `"<"` para lectura (podía ser `">"` para escritura), y, como es natural, el nombre del fichero, que tenemos en la variable `$leyendo`. Pero las cosas pueden ir mal: puede haberse comido el fichero el perro, puede no existir ese fichero, o puede haber cascado en ese preciso instante el disco duro. Así que hay que prever que la operación pueda ir mal, y dejar que el programa fallezca, no sin un epitafio adecuado.

```
./escrito.pl este.fichero.no.existe
No puedo abrir el fichero este.fichero.no.existe por No existe el fichero o el directorio
```

El epitafio lo proporciona $!, otra de las *variables por defecto o implícitas* de Perl que contiene el último mensaje de error del sistema.

> **Nota:** Las variables predefinidas o implícitas se pueden consultar escribiendo **`perldoc perlvar`**.

❹ Esto es un bucle `while`. Un pirulín para quien lo haya averiguado. Pero es un bucle raro, porque dentro de la condición de bucle (lo que hay entre paréntesis), lo que decide si se sique o no, tiene lo que parece una etiqueta HTML. Pues no lo es. Por alguna razón ignota, los paréntesis angulares (que así se llaman) es un operador que, entre otras cosas, lee la siguiente línea del filehandle incluido dentro. Y devuelve verdadero; cuando no puede leer más líneas, devuelve falso. Como era de esperar, este bucle va a recorrer las líneas del fichero. Detrás de while siempre va un bloque, y los bloques siempre llevan llaves. Como los coches. Aunque tengan una sola línea. Quién sabe qué podría pasarle a un bloque indefenso, si no llevara llaves.

❺ Se interpolan dos variables en una cadena, también predefinidas. El bucle recorre las líneas del fichero, pero no sabemos dónde las va metiendo. Pues van a parar a `$_`, la variable por defecto por excelencia, donde van a parar un montón de cosas que sabemos y otras muchs que ignoramos. Muchas funciones actúan sobre esta variable por defecto sin necesidad de explcitarla, y si hay un bucle huérfano sin variable de bucle, allí que te va esa variable por defecto para ayudarla. Y para que, a su vez, no se quede solita, le ponemos delante la `$.`, que contiene el número de línea del fichero que se está leyendo. Y ni le ponemos el retorno de carro detrás, porque en Perl, cuando se lee

de un fichero, se lee con sus retornos de carro y todo, para que uno haga con ellos lo que quiera (quitárselos de enmedio, y no acordarse de que están ahí, en la mayor parte de los casos).

❻ Los ficheros abiertos hay que cerralos, que si no pasa corriente y se puede resfriar el disco duro. Incluso habría que comprobar si se han cerrado correctamente, para ser más papistas que el camarlengo, pero lo vamos a dejar para mejor ocasión.

El resultado que obtenemos, será tal que así:

```
jmerelo@vega:~/txt/tutoriales/perl-apresurados/code$ ./escrito.pl ../diablocojuelo.txt | head
1 The Project Gutenberg EBook of El Diablo Cojuelo, by Luis Vélez de Guevara
2
3 This eBook is for the use of anyone anywhere at no cost and with
4 almost no restrictions whatsoever.  You may copy it, give it away or
5 re-use it under the terms of the Project Gutenberg License included
6 with this eBook or online at www.gutenberg.net
7
8
9 Title: El Diablo Cojuelo
10
```

Pero lo cierto es que en Perl hay otra forma de hacerlo. Leer ficheros es lo una de las principales aplicaciones de Perl, así que el programa anterior se puede simplificar al siguiente:

```
#!/usr/bin/perl
while (<>) {
  print "$. $_";
}
```

Programa minimalista donde los haya. Cuando Perl se encuentra los paréntesis angulares en un programa, hace todo lo siguiente: toma el primer argumento de la línea de comandos, lo abre como fichero, y mete la primera línea en la variable por defecto `$_`. Si no se ha pasado nada por la línea de comandos, se sienta ahí, a verlas venir, esperando que uno escriba cosas desde teclado (entrada estándar) y le de a `Ctrl-D` para equivale al carácter `EOF`, fin de fichero. Pero es que todavía se puede simplificar más, usando opciones de ejecución:

```
#!/usr/bin/perl -n
print "$. $_";
```

que se quita de enmedio hasta el while, que queda implícito por la opción `-n` que le pasamos al intérprete.

Nota: Y si se puede hacer eso en Python, que venga Guido Van Rossum y lo vea.

También se puede complicar más, claro. Por ejemplo, no me negaréis que esas líneas vacías con el numerito delante ofenden a la vista. No hay nada en esas líneas, nadie va a decir "Por favor, lean esta

línea" porque está vacía. Así que lo mejor es quitarle también los números. Y, de camino, no está de más curarnos en salud antes de abrir un fichero comprobando si se puede leer o no:

```
my $leyendo = shift
  || die "Uso: $0 <nombre de fichero>\n";if ( ! -r $leyendo ) {
  die "El fichero $leyendo no es legible\n";                          ❶
}

open my $fh, "<", $leyendo
  or die "No puedo abrir el fichero $leyendo por $!\n";while (<$fh>) {
  chop; chop;
  print "$. " if $_;
  print "$_\n";
}
close $fh;                                                            ❷
```

❶ En este caso, usamos el `if` en su forma más tradicional,
`if (condición)`
`{bloque}`
. Lo que no es tan tradicional es la condición: el operador `-r` comprueba si el fichero es legible (para el usuario), y devuelve falso si no lo es. De forma que:

```
jmerelo@vega:~/txt/tutoriales/perl-apresurados/code$ touch no-legible
jmerelo@vega:~/txt/tutoriales/perl-apresurados/code$ chmod -r no-legible
jmerelo@vega:~/txt/tutoriales/perl-apresurados/code$ ./escrito-if.pl no-legible
El fichero no-legible no es legible
```

❷ Para quitar de enmedio las líneas vacías, primero hay que tener en cuenta que no lo están. Tienen al final dos caracteres (los de retorno de carro y salto de línea), que hay que eliminar con sendos `chop`, que hace precisamente eso, elimina un caracter al final de la línea. Y con eso, en la línea siguiente escribimos el número de línea, pero sólo si queda algo en la variable por defecto.

En Perl siempre hay más de una forma de hacer las cosas. Se puede elegir la más elegante, o la más divertida. O la que uno conozca mejor, claro.

> **Ejercicios.** Ahora ya no tenemos excusa para no hacer nada. Vamos a hacerlo simple: un programa que cuente el número de líneas que no estén en blanco en un fichero.

8. Partiendo de una base

En realidad, sólo hemos rascado la superficie del diablo cojuelo, unas pocas líneas del principio. Pero hay que tragárselo todo, asimilarlo, sintetizarlo, y volverlo a asimilar. Y para hacerlo, lo mejor es

dividirlo en cachos fácilmente deglutibles, no vayamos a empacharnos. Y eso es lo que vamos a hacer con el programa siguiente, que divide la novela en cada uno de los *trancos* que la componen

```
#!/usr/bin/perl
use File::Slurp;                                                    ❶

@ARGV || die "Uso: $0 <fichero a partir por trancos>\n";
my $text = read_file( $ARGV[0] ) ;                                  ❶
my @trancos=split("TRANCO", $text);                                          ❷

for (@trancos[1..$#trancos]){                                                ❷
  print substr($_,0,40), "\n", "-"x40, "\n";
}
```

❶ Esta vez, para variar, usamos otro módulo diferente para leer ficheros: uno que se traga el fichero sin rechistar, y en una sola línea, y lo mete en una variable (`$text`). El módulo `File::Slurp` tendrás que instalarlo previamente, claro, de tu bienamada y nunca suficientemente ponderada CPAN (http://search.cpan.org).

❷ Pero vamos al meollo del asunto, es decir, la matriz. Que es lo que aparece en esta línea, una matriz, y como el asunto es de 11 arrobas, tiene una arroba delante. Una `@`, vamos. Todo lo *vectorial* en Perl tiene una @ delante. Los vectores/matrices son variables, y hay que declararlos, como hacemos con `@trancos`; también hay vectores predefinidos, como `@ARGV`, que contiene los argumentos que se le pasan al programa. Precisamente en la primera de las líneas aquí referenciadas lo que se hace es comprobar si existe ese vector de argumentos. Si no hay ninguno, se sale con un mensaje de error.

El vector `@trancos` se define a partir del texto, partiéndolo (`split`, como hemos visto anteriormente) por la palabra `TRANCO`, que es la que hay al principio de cada capítulo de *El Diablo Cojuelo*.

Los elementos de un vector se pueden sacar uno por uno, como en `$ARGV[0]`, que como es un e$calar lleva el $ delante. Los vectores empiezan por 0, o también a puñados, como en `@trancos[1..$#trancos]`, que extrae del segundo elemento (que tiene índice uno) hasta el último (que no es otra cosa lo que representa `$#trancos`).

> **Nota:** Las estructuras de datos de Perl vienen explicadas en la página de manual `perldata` (ya sabéis, **`perldoc perldata`**) incluyendo muy al principio los vectores.

❸ El bucle sigue el esquema habitual (en Perl). La variable de bucle no está declarada por que es `$_`; dentro del bucle, mediante `substr`, se extraen los primeros 40 caracteres y se imprimen, y luego, usando el operador `x` de multiplicación de cadenas (`"ab"x3` dará `"ababab"`), se completa este bonito y útil programa.

> **Nota:** Más información sobre los operadores de perl tecleando **`perldoc perlop`**, inclusive reglas de asociatividad.

Ejercicios. Una vez hechos los trancos, lo suyo sería dividir por párrafos cada uno. Y añadirle etiquetas HTML, qué diablos. Así que el ejercicio consiste en modificar el programa anterior para que divida cada tranco en párrafos y le añada etiquetas HTML, teniendo en cuenta que los párrafos comienzan (o terminan) con dos retornos de carro, pero que dependiendo del sistema operativo origen, los dos retornos de carro pueden ser `\n\n` o `\r\n\r\n` (que será el caso, si usáis el mismo fichero que yo).

9. Ni bien ni mal, sino regular

A trancos y barrancos llegamos a un punto en que el tema se pone interesante. Si todavía estás con nosotros, agárrate, que ahora viene lo gordo: hacer un índice onomástico, es decir, los nombres de las gentes que aparecen en la novela, y dónde diablos aparecen. Lo bueno es que allá por el siglo XVIII la gente era muy educada, y a todo el mundo lo trataban de *Don*, así que un nombre será algo en mayúsculas después de un Don. Ahí vamos

```
my $fichero_a_procesar = shift
  || die "Uso: $0 <nombre de fichero>n";
open my $fh, "<", $fichero_a_procesar  || die "No puedo abir el fichero. Error $!\n";
my %indice;
while(<$fh>) {
  if ( /[Dd]on ([A-Z][a-záéíóúñ]+)/ ) {                                    ❶
    $indice{$1} .= "$.";
  }
}
for (sort {$a cmp $b} keys %indice ) {
  print "*Don $_\n\t$indice{$_}\n";                                        ❷
}                                                  ❶
```

❶ Los diccionarios son útiles entre otras cosas para equilibrar una mesa, pero también para almacenar definiciones. Son muy rápidos para buscar información: vas directamente a la letra, y buscas secuencialmente. Los vectores no lo son: tienes que ir examinando uno por uno todos los elementos. Los vectores son útiles para almacenar información a la que se va a acceder secuencialmente, sin embargo, los diccionarios sirven para almacenar información a la que se va a acceder usando una palabra clave. En Perl, los diccionarios se llaman *hashes* o variables asociativas, y a la palabra clave que se usa para acceder a su contenido, se le llama *key* o clave[6]. Los *hashes* tienen un `%` delante, que es lo más parecido a una K si se mira de lejos y con los ojos entrecerrados. Eso es lo que declaramos en esta referencia; sin embargo, cada uno de los contenidos de estos hashes son escalares, y se usa la misma convención que en los vectores, como sucede en el resto de las líneas marcadas.

En este programa se va creando un hash con los nombres que se van encontrando, y el contenido del mismo son las líneas en las que aparece el nombre (para las que usamos la variable predefinida `$.`).

El índice se imprime una vez recorrido el fichero, en las últimas líneas del programa. Para empezar, el comienzo del bucle es de esos que le dan al Perl un mal nombre, pero recorriéndolo de dercha a izquierda podemos irlo decodificando. A la derecha está el nombre del hash; y un poco más allá la

función `keys`, que devuelve un vector con las claves del hash. Recorrer una matriz es fácil: empiezas por 0 y acabas por el último vector, pero para recorrer un hash necesitas saber cuáles son las claves que tiene. Y resulta más útil si lo recorres siguiendo un orden determinado. En este caso,

```
sort {$a cmp
$b}
```

clasifica (`sort`) usando un bloque (`$a cmp $b`). `$a` y `$b` representan los dos elementos que se comparan en la clasificación, y `cmp` es una comparación alfabética, que devuelve -1, 0 o 1 dependiendo de si el primero es mayor, son iguales o lo es el segundo. `sort` toma como argumento un vector y lo devuelve clasificado, numéricamente si no se le indica ninguna expresión, y usando la expresión si la hay. Por ejemplo, si quisiéramos que recorriera el hash por orden de número de apariciones (que equivalen a la longitud de la cadena que las representa), podríamos poner

```
sort {length($indice{$a}) cmp
length($indice{$b})
```

.

Dentro del bucle, nada inesperado: la variable por defecto `$_` toma el valor de cada una de las claves del hash, y se escribe tal clave (con el Don por delante, que no falte) y el contenido de la misma (`$indice{$_}`).

> **Nota:** Una vez más, como aconsejamos para los vectores, se puede ampliar información escribiendo **`perldoc perldata`**

❷ Si algo caracteriza al lenguaje Perl, son las expresiones regulares. Posiblemente fue el primer lenguaje de programación que las introdujo de forma nativa, hasta el punto que lenguajes posteriores han adoptado su formato. Una expresión regular es una forma sintética de expresar la forma de una cadena alfanumérica. Las expresiones regulares usan símbolos para representar grupos de caracteres (por ejemplo, números, o espacios en blanco) y repeticiones de los mismos (que aparezcan 1, n o más veces) y, lo que es más importante, qué parte de la cadena nos interesa para hacer algo con ella (por ejemplo, extraerla o sustituirla por otra).

En Perl, la forma más común de encontrarse las expresiones regulares entre dos *slash* `//`. Por ejemplo, nos lo encontramos donde dividimos al diablo cojuelo en sílabas en la forma siguiente:

```
/[\s,]/
```

. Esta expresión regular indica un grupo de caracteres (`[]`) que pueden ser o bien un espacio en blanco (`\s`, se refiere tando al espacio como a retornos de carro diversos o tabuladores) o bien una coma. En ese programa, como se vio, siempre que `split` se encontrara uno de estos dos caracteres, crearía un nuevo elemento del vector.

En este programa tenemos una expresión regular similar:

```
/[Dd]on ([A-Z][a-záéíóúñ]+)/
```

La primera parte concidirá con cualquier cadena que empiece por `don` o `Don`; `[Dd]` coincide con un solo caracter que esté entre el grupo entre corchetes. Y algo similar es la expresión entre paréntesis (que indican que es la parte que queremos guardar para luego), comienza don `[A-Z]`, que es una serie de caracteres que comienzan por A y terminan por Z, lo que vienen siendo las mayúsculas, y luego cualquier letra minúscula (inclusive letras con acento y la ñ, que no están dentro del alfabeto anglocabrón), pero que aparezcan una o más veces (de ahí el `+`; un `*` habría significado 0 o más

veces, y una `?` un elemento opcional, que puede aparecer o no). En resumen, esta expresión regular coindirá con `Don Cleofás` o `don Domingo`, pero no con `Sir James`, ni con `don dinero` (la segunda palabra no está en mayúsculas), ni siquiera con `Don Dinero` (porque tiene dos espacios). Coincidirá precisamente con lo que queremos que coincida.

> **Nota:** Una vez más, la documentación incluida en perl es nuestra amiga: **`perldoc perlrequick`** es una referencia rápida,**`perldoc perlretut`** un tutorial más extenso, y **`perldoc perlre`** un manual de referencia.

Ejecutado sobre el diablo cojuelo, este programa dará una salida tal que así:

```
*Don Adolfo
258 293 476 3487 3652 5864 9392
*Don Agustín
3794 5462 8955
*Don Alfonso
8830
*Don Alonso
2858 3562 3676 6061 6771
*Don Alonsos
1198
*Don Alvaro
1233 2043 3534
*Don Ambrosio
7703
*Don Américo
5513
*Don Antonio
2110 2611 2657 2983 3472 5335 5460 5800 6133 9048
*Don Apolo
3219
*Don Baltasar
2647 2653
*Don Beltrane
7439 7457
*Don Beltrán
7436 7451
*Don Bueso
6906
*Don Carlos
2671 4238
*Don Cayetano
3463
*Don Clarian
5846
*Don Claudio
2697
```

```
*Don Cleofas
323
*Don Cleofás
658 694 719 727 735 763 782 792 807 810 825 844 860 869 914 936 950 961 979 993 1007 1020 1045 1052 1
*Don Cristóbal
2894 3459 8950
```

, donde queda bastante claro quién es el prota (aparte del Diablo Cojuelo, que por ser diablo no tiene Don).

Notas

1. Lo que es recuerdo también de aquellos mismos tiempos en que `STDOUT` era un convento, en el que para seguir en la página siguiente tenían que esperar que retornara el carro que les traía las pieles de becerro curtidas en las que escribían lo que el programador les ordenaba.
2. Los monjes trapenses de la congregación periférica de E/S ya no, desgraciadamente, y se dedican a la elaboración de un delicioso licor de alcachofa
3. Traducción del sajón antiguo *heredoc*. Pero no es la única. Se admiten otras.
4. Aunque no es exactamente lo mismo
5. Lo que viene siendo un *filehandle* de toda la vida.
6. En realidad, los vectores serían un tipo especial de diccionarios, que sólo admitirían números como palabra clave; así es, además, como se implementan internamente en Perl