Perl para apresurados

Juan Julián Merelo Guervós

Historial de revisiones

Revisión 1.0 Jul 2006
Preparando la primera versión para el curso de Extremadura
Revisión 1.1 26 de Julio de 2006
Versión campus party *probada en combate*. Simplificados algunos ejemplos, corregida ortografía y gramática. Añadi
Revisión 1.2 3 de Agosto de 2006
Algunos arreglos mínimos

Tabla de contenidos

¿Quién eres tú?

Este tutorial te puede estar llegando en una de varias formas diferentes. Estas son todas las posibles:

- Fuentes en DocBook.[2] Tiene que haber gente pa tó.
- Versión de *Perl para apresurados* en PDF[3]. Posiblemente, la que se vea con más claridad.
- Versión de *Perl para apresurados* en HTML, dividido en varias páginas[4]

Además, te puedes descargar los ejemplos[5] usados en este tutorial.

Eso mismo te estarás preguntando, que quién diablos eres y que a qué dedicas el tiempo libre. 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[6]), o incluso que el símbolo mayor y menor vayan para ti asociados de forma indisoluble a un acrónimo capicúa[7].

Vamos, que pueden 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 ti. A ti, 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[8] en tu lenguaje de programación favorito. En fin, donde quiera que haga falta convertir cosas en otras cosas o convertir programadores en poetas, 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[9].

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[10] 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 impaciencia[11]. 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[12], o a la hiperactividad por ingestión de bebidas excitantes[13]. Los camellos son buenos. Los camellos son útiles. Llevan cosas encima. Tienen joroba. Amemos a los camélidos (las llamas también son camélidos[14]).

No menos importante es tener un ordenador con sistema operativo. Incluso sin él[15]. Ejecuta lo siguiente para saber si lo tienes: **`perl -v`** a lo que el ordenador debidamente contestará algo así:

```
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.
```

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

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)[16] y bajarse la versión compilada.

> **Nota:** Aunque no es estrictamente necesario para programar Perl (y, por tanto, puedes evitarlo si tienes problemas de espacio), conviene bajarse también el paquete `perldoc`, que no sólo contiene documentación, sino también la orden **perldoc** para visualizarla.

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[17], 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 para hablarte de otras opciones. No es que haya muchas, pero hay alguna. Por ejemplo, puedes usar el conocido entorno Eclipse[18] con el plugin EPIC[19] para desarrollar proyectos en Perl, como se muestra en la figura siguiente.

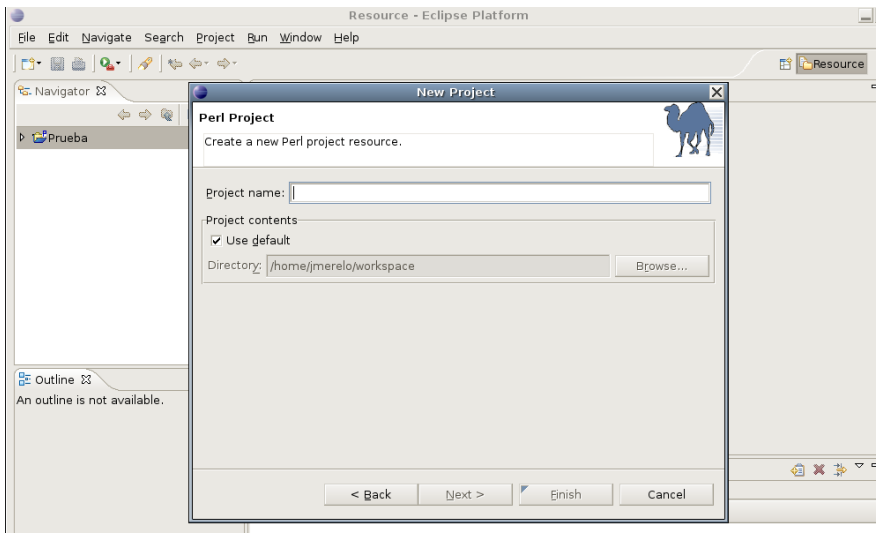

Figura 2. Iniciando un proyecto en Perl con EPIC/Eclipse

Otros entornos de desarrollo, como PerlIDE o Komodo, o bien no siempre funcionan o bien son de pago. Si consigues que te lo compre tu jefe, 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.

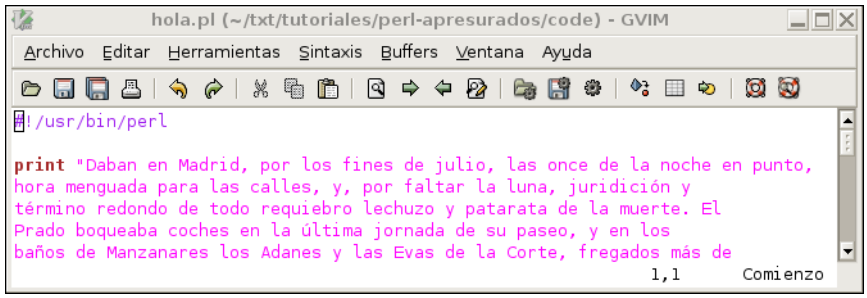

Figura 3. Editando un programilla con gvim

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? ¿El editor es *sensible al contexto* y tiene un modo específico para Perl? Si la respuesta a alguno de ellos es *no*, ¿a qué esperas para tenerlo? Venga, te espero.

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 se 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:

> **Nota:** Los programas ejemplo de este tutorial deberían estar en el fichero `perl-apresurados-ejemplos.tgz`[20]

```
#!/usr/bin/perl                                                    (1)
print "Daban en Madrid, por los fines de julio, las once de la noche en punto..."; (2)
print "\n";                                                        (3)
```

(1) 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 conveniente para 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.

Si piensas ejecutar el programa en diferentes entornos, también puedes usar **#!/usr/bin/env perl**, que ejecutará el primer intérprete de Perl que encuentre en el camino. También puedes usarlo si sabes que Perl está por ahí, el pillín, pero no quieres preocuparte de dónde.

(2) Aquí se imprime, con el *nihil obstat* obtenido previamente. Obsérvense las comillas y el punto y coma. Las órdenes en Perl se separan 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. Y no me refiero a la trapense.

Obsérvese también que el argumento se le pasa a `print` directamente, sin paréntesis. Los paréntesis son opcionales. Convencionalmente no se usan cuando se trata de funciones *nativas*, como esta, aunque se suelen usar para subrutinas y métodos definidos en módulo o por el usuario.

(3) 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?[21]

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 programa sería algo similar al que aparece en la captura siguiente

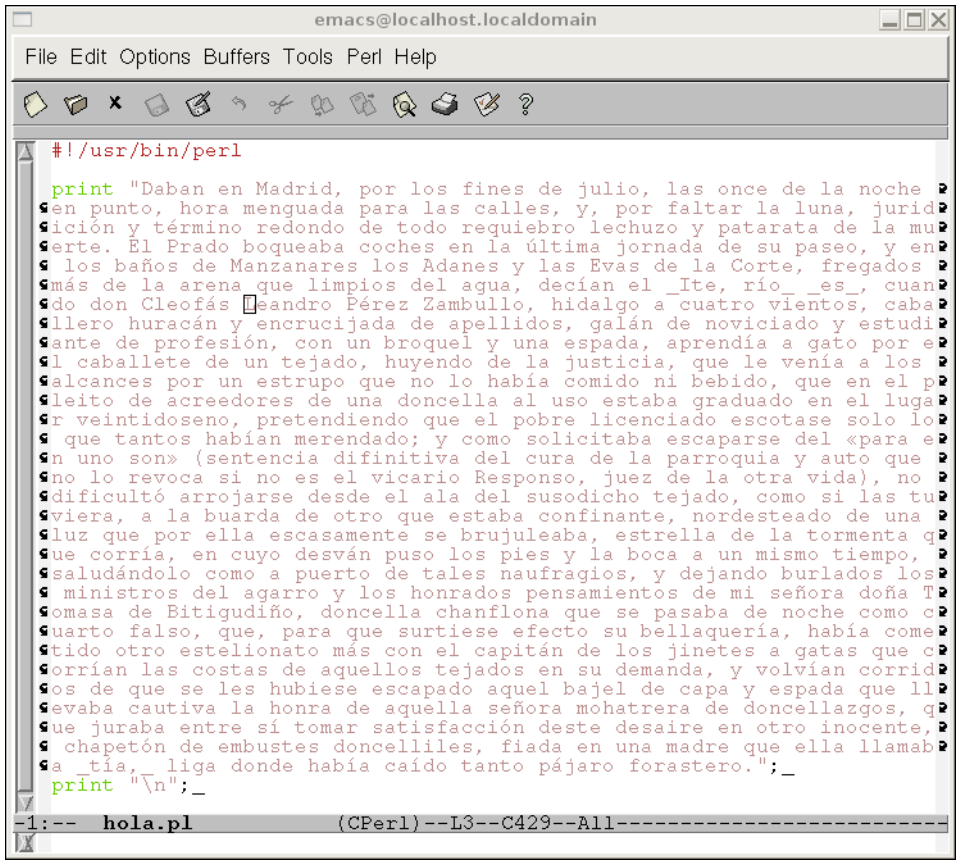

Figura 4. Editando hola.pl en emacs

> **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.

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 (de la eterna perdición) el fichero, y escribir **`perl hola.pl`**. Pero ya que estás puesta (o puesto), 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.

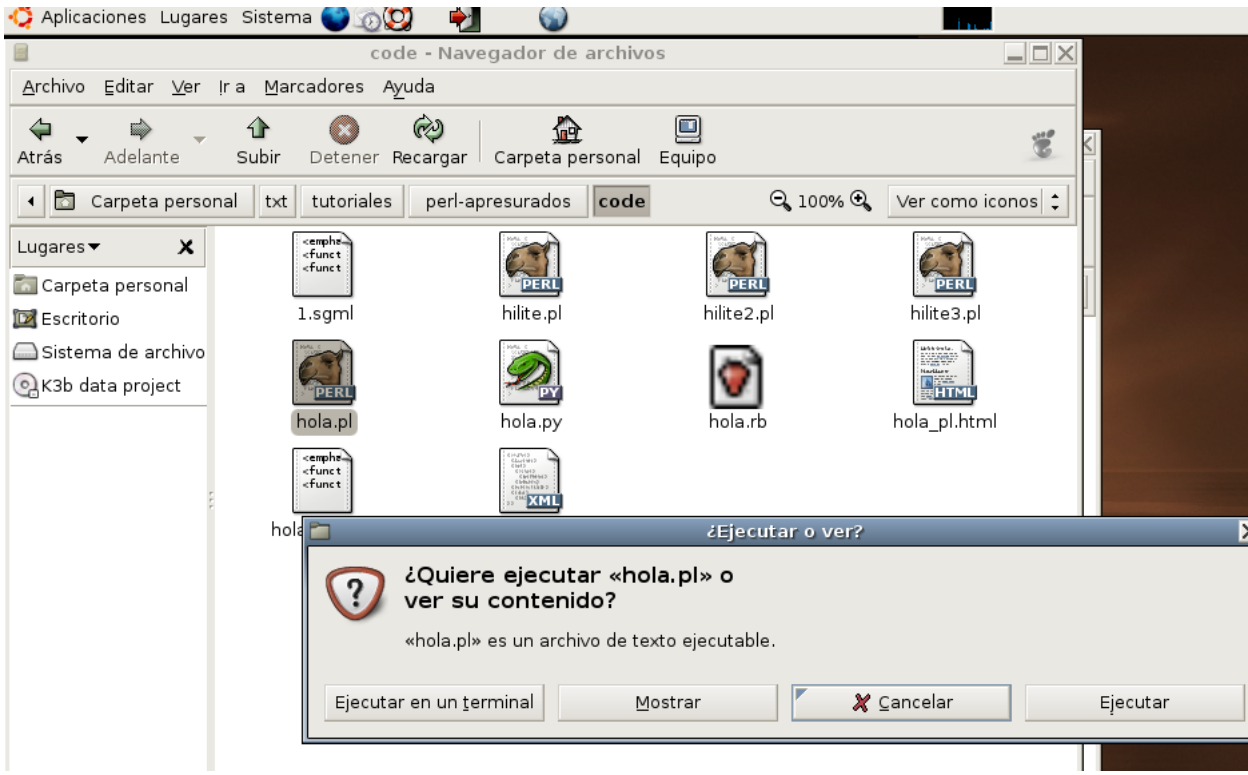

Figura 5. Ejecutando un programa en Perl desde Gnome

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: abrirá una ventana, escribirá el texto y saldrá como una exhalación.

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;  # Quieto parao ahí
```

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

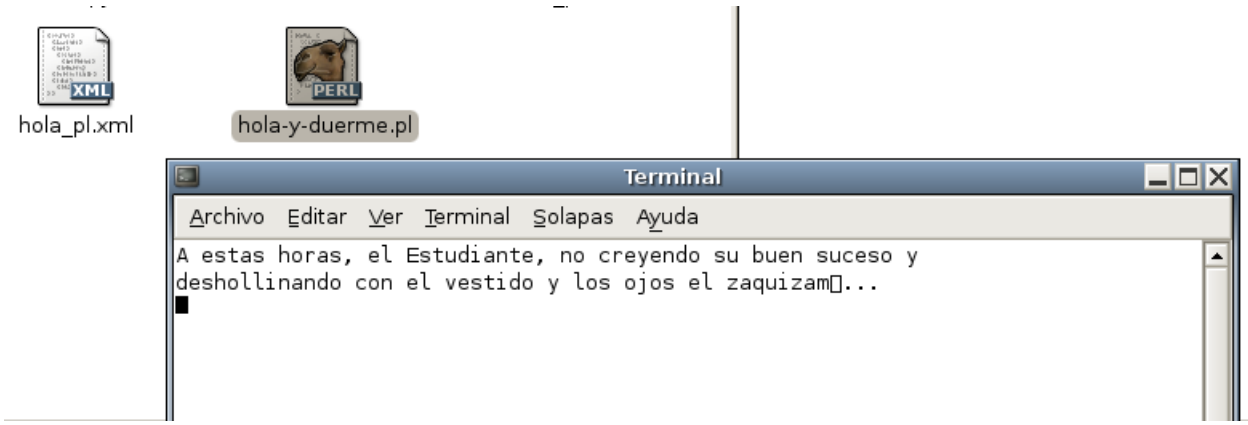

Figura 6. Terminal con el resultado de ejecutar el programa `hola-y-duerme.pl`

En otros sistemas operativos (o entornos), cambiará el icono y la apariencia del terminal donde está el resultado, pero por lo demás, el resultado será el mismo (y el sonriente camello, también). 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).

Por cierto que, de camino, hemos introducido nuestro primer comentario, que en Perl van precedidos por una *almohadilla* (#) y llegan hasta el final de la línea. Un buen programador debe ser un buen comentarista. Lo contrario no siempre es cierto.

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, que se ha cargado, interpretado y mirado a ver si hay errores de sintaxis; 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                                                                    (1)
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;
```

(1) **n**, de *next*, siguiente, ejecuta la línea ídem, es decir, justamente la que aparece al final de el ejemplo anterior

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

(3) 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; puedes usar, por ejemplo, la línea **perl -de0**, que te deposita directamente en el depurador sin ejecutar ningún programa. 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:

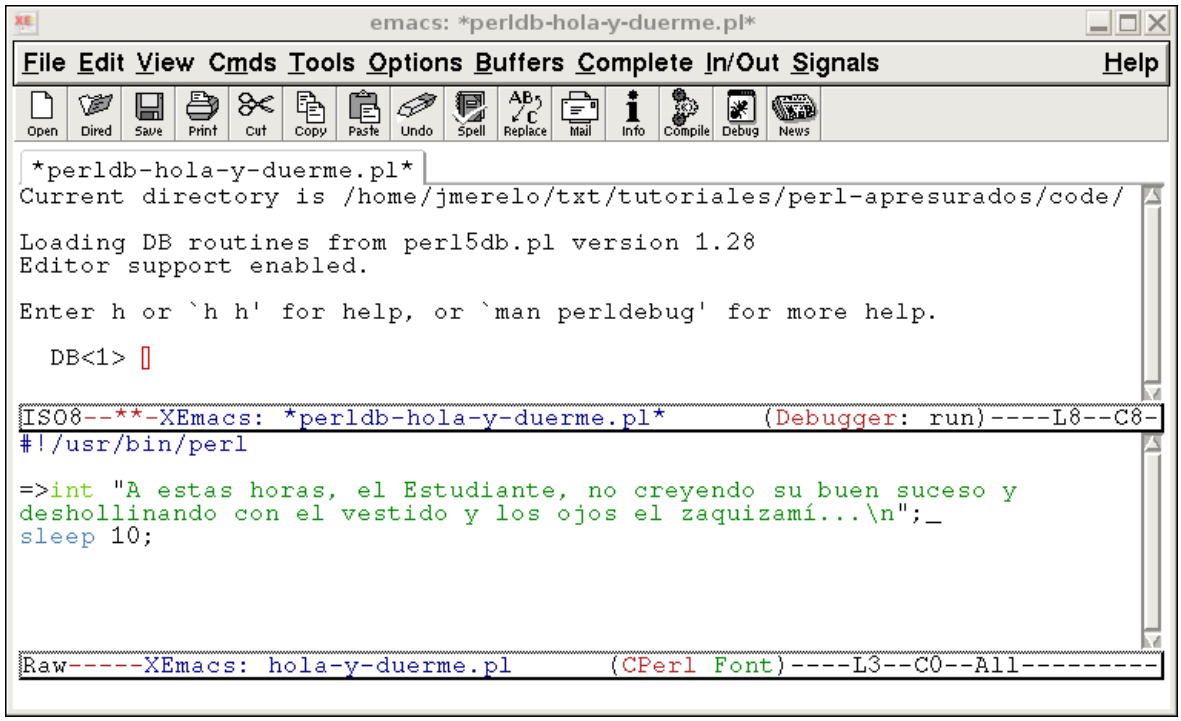

Figura 7. Depurando un programa en el mismo editor XEmacs. La flecha está situada sobre la siguiente línea a ejecutar.

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.

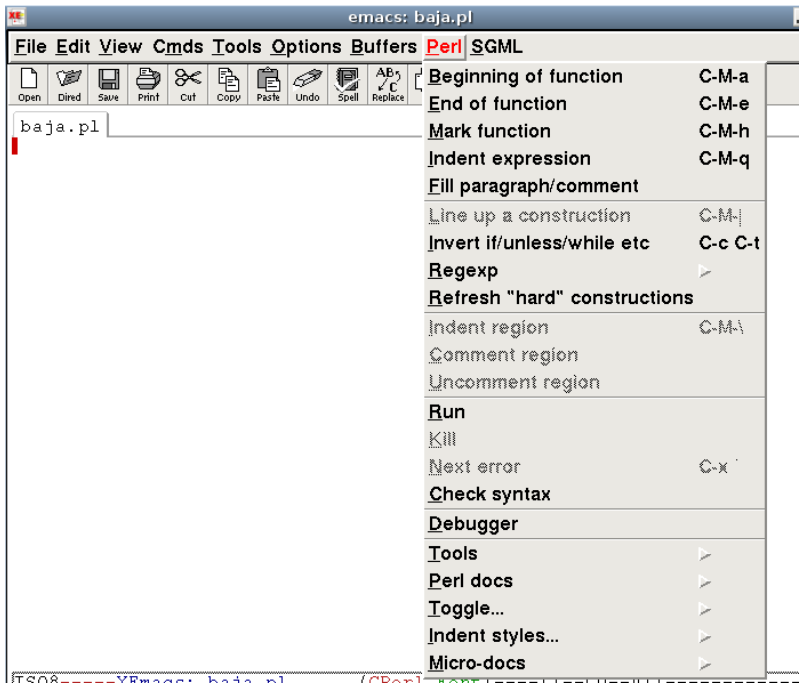

Figura 8. Menú específico del modo `CPerl` para XEmacs, mostrando las diferentes opciones, incluyendo la de depuración y otras cuantas que posiblemente no uses nunca.

Con esto, ya estamos listos para abordar empresas más elevadas 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. Recuerda, **h** es tu amiga.

Usando la sabiduría colectiva

Escribir está bien. Hay dos o tres personas que incluso se ganan la vida con ello[22]. 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[23]. 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[24], aprovecha para pensar en él con cariño. Antes de que la falta de calor humano lo convierta en un operador bastardo del infierno[25]. Para instalar módulos de CPAN para que sean accesibles para todo el mundo hace falta tener privilegios de administrador; 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[26]. 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.

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 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. Ten en cuenta que los basados en `ftp` puede que no le gusten a tu cortafuegos; lo más seguro es seleccionar primero los que usen el protocolo `http`.

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
  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.
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/.p
Appending installation info to /usr/local/lib/perl5/5.9.3/i686-linux-thread-multi-ld/perllocal.p
  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);
```

Todos los módulos en CPAN están documentados con un pequeño manual de referencia y ejemplos; es conveniente tener este manual abierto cuando se programe, o usar directamente los ejemplos incluidos en la SINOPSIS para comenzar a programar con él.

Vamos a ver ahora como se usa ese pozo de sabiduría para bajarnos una página web

```
use LWP::Simple;                                                    (1)

getprint('http://localhost:8020/');                                 (2)
```

(1) En esta línea lo único que se hace es cargar la librería que tendrá que estar instalada; no es el caso con esta, porque forma parte del núcleo o *core*. *Cargar* implica incorporar las funciones que ese módulo exporte; para saber qué funciones son esas habrá que consultar la página de manual, claro.

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 LWP::Simple;
```

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

(2) Sin dar muchas más vueltas, aquí está la orden que se baja la página web y la imprime. `getprint` es precisamente una de esas funciones que el módulo ha importado.

Por otra parte, hay una forma más simple de hacer lo mismo, directamente desde la línea de comandos

```
perl -MLWP::Simple -e 'getprint("http://localhost:8020")'
```

La opción **-M** carga el módulo que la sigue, y **-e** ejecuta el código Perl que le sigue. Es conveniente para programas cortos, o si se quiere hacer un alias del shell, por ejemplo.[27]

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*.

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) o 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 siguiente 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 escribas un fichero con errores y digas *Um, parece que tiene un error*. Pero acuérdate de lo leído, te será útil. O lo será para quien llegue aquí desde Google[28].

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 = "diablocojuelo.txt";                                    (1)
open my $fh, "<", $leyendo                                            (2)
  or die "No puedo abrir el fichero $leyendo por $!\n";
while (<$fh>) {                                                       (4)
  print "$.$_";                                                       (5)
}
close $fh;                                                            (2)
```

(1) 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 el ámbito de una variable puede ser cualquier bloque (que están encerrados entre llaves (`{}`), como uno que hay 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*.

(2) 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[29], 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.

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.

(3) Si hay algún problema al abrir el fichero, el programa irremisiblemente muere (`die`). Pero no muere por las buenas, sino que te da un mensaje que te demuestra que hay una buena razón para hacerlo:

```
./escrito.pl
No puedo abrir el fichero diablocojuelo.txt porque 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 documentación sobre las variables predefinidas o implícitas se puede consultar escribiendo **`perldoc perlvar`**.

Pero no hemos dicho nada del `or` 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 la anteriores viene a decir "Voy a intentar abrir el fichero, *pero si no* pudiera o pudiese, me muero y dejo este mensaje" Ese *pero si no*, `or`, que representa a la función lógica `O`[30]hace que se ejecute su parte derecha solo si la parte izquierda tiene como resultado un valor falso, es decir, si `open` devuelve `undef`, lo que ocurrirá si con cualquier problema a la hora de abrir el fichero.

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: una variable dentro de una cadena se sustituirá por su valor al ejecutar el programa.

(4) Esto es un bucle `while`. Un pirulí 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 sigue 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.

(5) 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 `$_`, la variable por defecto por excelencia, donde van a parar un montón de cosas que sabemos y otras muchas que ignoramos. Muchas funciones actúan sobre esta variable por defecto sin necesidad de explicitarla y si hay un bucle huérfano sin variable de bucle, allí que te va esa variable por defecto para ayudarla. Y para que, a su vez, no se quede solita, le ponemos delante la `$.`, 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 en medio y no acordarse de que están ahí, en la mayor parte de los casos).

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 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 dé a `Ctrl-D`, que equivale al carácter `EOF`, fin de fichero. Lo que equivale exactamente a leer de la entrada estándar, que se puede hacer de alguna de las múltiples formas posibles: **`cat diablocojuelo.txt| ./escrito2.pl`** o **`./escrito2.pl < diablocojuelo.txt`**, sin ir más lejos. . Pero es que todavía se puede simplificar más, usando argumentos de línea de comandos:

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

que se quita de en medio 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 = "diablocojuelo.txt";
if ( ! -r $leyendo ) {                                          (1)
  die "El fichero $leyendo no es legible\n";
}
open my $fh, "<", $leyendo
  or die "No puedo abrir el fichero $leyendo porque $!\n";
while (<$fh>) {
  chop; chop;
  print "$." if $_;                                             (2)
  print "$_\n";
}
close $fh;
```

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

(2) 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 carácter 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. Por eso no tenemos que limitarnos a abrir ficheros y escribir porquería en pantalla, que luego se ponen los terminales perdidos y parecen el salvapantallas de Matrix. Así que vamos a escribir la salida en un fichero, como hacemos en el siguiente programa

```
my $leyendo = "diablocojuelo.txt";
if ( ! -r $leyendo ) {
  die "El fichero $leyendo no es legible\n";
}
open my $fh, "<", $leyendo
  or die "No puedo abrir el fichero $leyendo por $!\n";
open my $fh_out, ">", "$leyendo-out";
while (<$fh>) {
  chop; chop;
  print $fh_out "$." if $_;
  print $fh_out "$_\n";
}
close $fh;
close $fh_out;
```

Este programa es igual que el anterior, salvo por las líneas resaltadas, que abren el fichero y escriben en él. La única diferencia entre abrir y cerrar un fichero es el símbolo de menor (<) que ahora mira para el otro lado. El nombre del fichero de salida lo hemos creado a partir del de entrada añadiéndole `-out`. Y en cuanto a escribir un fichero, se diferencia de escribir en pantalla (en salida estándar, en realidad) sólo en la inserción del filehandle (**print $fh_out**) tras la orden. Sin coma detrás, ojo.

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, y lo escriba en un fichero de salida cuyo nombre se cree a partir del nombre del fichero original, con la extensión `lc`.

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;                                                    (1)

@ARGV || die "Uso: $0 <fichero a dividir trancos>\n";
my $text = read_file( $ARGV[0] ) ;                                  (1)
```

```
my @trancos=split("TRANCO", $text);                                    (2)

for (@trancos[1..$#trancos]){                                          (2)
  print substr($_,0,40), "\n", "-"x40, "\n";
}
```

(1) 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[31].

(2) 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[32] son variables y es aconsejable 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`) 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 y se puede trabajar con ellos elemento por elemento o 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`); el operador `..` genera en Perl una vector que contiene una secuencia ascendente del primer elemento al último (incluidos). Es decir, `1..5 == qw( 1 2 3 4 5)`

> **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.

(3) 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 y precedencia.

Pero no hemos visto cómo se asignan valores directamente a los vectores. El programa siguiente, que modifica el anterior poniendo nuestro propio ordinal a los capítulos, usa una de las formas posibles:

```
use File::Slurp;

my @ordinales = qw( primero segundo tercero cuarto quinto
      sexto séptimo octavo noveno décimo );

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

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

```
}
```

El vector se declara al principio del programa, usando la forma menos incómoda, que se ahorra comillas y espacios y cosas. Siempre que cada uno de los elementos del vector sea un *bloque* sin espacios en medio, se puede usar `qw()`. La forma más general, sin embargo, sería

```
my @ordinales= ('Primero','Segundo',...)
```

Para ir recorriendo ese vector y escribiendo sus contenidos por orden, usamos `shift`, que extrae el primer elemento de un vector, es decir:

```
DB<1> @vector = qw( primero segundo tercero );

  DB<2> print shift @vector
primero
  DB<3> x @vector
0  'segundo'
1  'tercero'
```

. **x** muestra el contenido de una variable en el depurador de Perl. Por tanto, en cada iteración del bucle el vector `ordinales` perderá su primer elemento. Por lo demás, `print` escribirá todo lo que le pasemos, separado por comas. Es así de obediente.

> **Nota:** Esta forma de recorrer un vector es más eficiente que usar un índice sobre los elementos del mismo, como suele hacerse en C. Y sólo es incomprensible si no estás acostumbrado.

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).

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                                                    (1)
  || die "Uso: $0 <nombre de fichero>n";
open my $fh, "<", $fichero_a_procesar  || die "No puedo abrir el fichero. Error $!\n";

my %indice;                                                                       (2)
while(<$fh>) {
  if ( /[Dd]on ([A-Z][a-záéíóúñ]+)/ ) {                                                (
    $indice{$1} .= "$.";                                                          (2)
  }
```

```
}

for (sort {$a cmp $b} keys %indice ) {                    (2)
  print "*Don $_\n\t$indice{$_}\n";
}
```

(1) He aquí una forma alternativa de leer los argumentos de la línea de comandos. Sin dejar de usar `@ARGV`, en este caso echamos mano de `shift`, que, si no tiene argumento al que hincarle el diente, lo hace directamente sobre él. O séase, la línea anterior sería exactamente igual que:
`my $fichero_a_procesar = shift @ARGV`
.

(2) 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 para encontrar lo que buscas, si no sabes su índice. 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[33]. 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 derecha 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`**

(3) 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 estructura 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* `//`. En la expresión regular siguiente (a veces abreviada como *regexp* o *regex*):
`/[Dd]on ([A-ZÁÉÍÓÚ][a-záéíóúñ]+)/`
La primera parte concidirá con cualquier cadena que empiece por `don` o `Don`; `[Dd]` coincide con un solo carácter 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 con `[A-ZÁÉÍÓÚ]`, que es una serie de caracteres que comienzan por A y terminan por Z (más las mayúsculas acentuadas, aunque en este texto no sirven de mucho), lo que vienen siendo las mayúsculas y luego cualquier letra minúscula (inclusive letras con acento y la ñ, que no están dentro del alfabeto anglosajón), 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 coincidirá 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, aunque no se vean). Coincidirá precisamente con lo que queremos que coincida.

Una forma fácil de visualizar las expresiones regulares es el programa `kregexpeditor`, que aparece abajo con la expresión regular anterior.

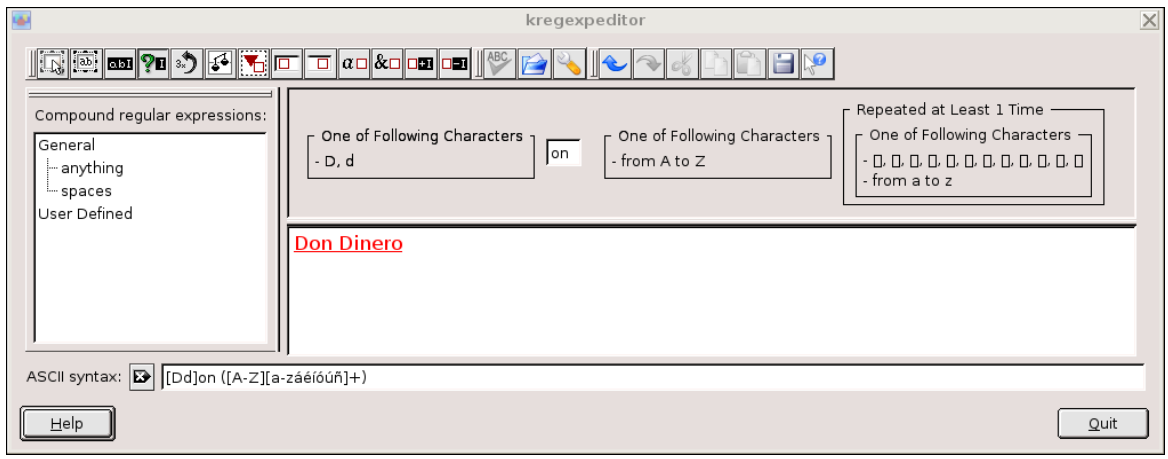

Figura 9. Programa `kregexpeditor` presentando la expresión regular `([A-Z][a-záéíóúñ]+)`, y mostrando subrayado en rojo una cadena que cumple la expresión. En el panel superior se muestra una explicación de los diferentes elementos de la regex.

> **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
*Don Cristóbal
 2894 3459 8950
```

, donde queda bastante claro quién es el prota (aparte del Diablo Cojuelo, que por ser diablo no tiene Don). Y que no poner un acento pasa hasta en las mejores familias.

Pero la utilidad de matrices asociativas y expresiones regulares no acaba ahí. Ni la del dominio público: vamos a modificar *El Diablo Cojuelo* para sustituir unos cuantos nombres, a gusto del consumidor. Y lo haremos con el siguiente programa:

```
use File::Slurp;

my %roles = (
       Madrid => 'el foro',
       'Cleof[áa]s' => 'El prota',
       '[Ee]l [Dd]iablo [Cc]ojuelo' => 'Su Satánica Majestad' ,
      );

my $fichero_a_procesar = shift
  || die "Uso: $0 <nombre de fichero>n";

my $texto=read_file($fichero_a_procesar);

for ( keys %roles ) {
  $texto =~ s/$_/$roles{$_}/g;                                    (2)
}
```

```
print $texto;
```

(1) Aquí definimos los cambios. Más claro no puede estar: lo que está a la izquierda de la flechica, se convertirá en lo que hay a la derecha de la flechica. Y lo definimos usando un hash, que usan un paréntesis tal como los vectores. Las claves están a la izquierda y si son amazacotadas (sin espacios ni caracteres raros en medio; de hecho, si tienen la misma sintaxis que un nombre de variable en Perl), se les pueden quitar las comillas (como pasa con `Madrid`. La última coma, por cierto, no es necesaria, pero si conveniente.

La *coma gorda (=>)* sustituye a la coma normal cuando se definen expresiones regulares; adicionalmente, se pueden omitir las comillas. Por supuesto, también se pueden definir como las matrices normales, sin más que poner a cada clave seguida por su valor o definición.

(2) Las sustituciones se hacen precisamente en esta línea, usando el escueto comando `s///`, que tiene una sintaxis un tanto curiosa, procedente del ignoto comando de Unix **sed**. Por lo menos usa la `s` de *sustituir*. `s/esto/aquello/xyz` vendría a ser algo así como `sustituye(esto,aquello,xyz`, donde el último argumento son una serie de opciones que modifican su comportamiento. El primer argumento puede y debe ser una expresión regular; el segundo puede ser cualquier cosa, pero si ponemos una expresión que incluya variables, tenemos que añadir al final la opción `e` para que la evalúe. Y la `g` es para que no se pare en la primera sustitución, sino que siga hasta que llegue al final del texto.

Y el resultado (parte de él), sería algo así como esto:

```
--¿Quién es aquí Su Satánica Majestad? Que he tenido soplo que está aquí en
este garito de los pobres, y no me ha de salir ninguno deste aposento
hasta reconocellos a todos, porque me importa hacer esta prisión.

Los pobres y las pobras se escarapelaron viendo la justicia en su
garito, y el verdadero Diablo Cojuelo, como quien deja la capa al toro,
dejó a Cienllamas cebado con el pobrismo, y por el caracolillo se
volvieron a salir del garito él y don El prota.
```

que no es que sea perfecto, pero es un comienzo.

Ejercicios

Las humildes palabras, que no tienen título, también pueden y deben ser contadas, para llevar una contabilidad exacta. Así que contemos todas las palabras en minúscula que aparecen en un texto, y hagamos un ránking de las 50 palabras más comunes. Por ejemplo.

A dónde vamos desde aquí

Lo que distingue a Perl de los otros lenguajes es la comunidad, y la humildad y accesibilidad de los *gurus*; no es extraño encontrárselos en los foros o listas de discusión, ni que te acepten un parche para sus módulos. Por eso nunca va a faltar información. Si no manejas la lengua del imperio, sin embargo, no hay tanta.

Libros

Multitud de libros te ayudarán a penetrar en el maravilloso mundo del Camello. Puedes empezar por Learning Perl, seguir por Programming Perl, continuar con el

Perl Cookbook, y acabar con High Order Perl o con Perl Hacks. La mayoría de los buenos libros de Perl están publicados por O'Reilly[34], y todos ellos están disponibles como e-libros en el servicio Safari[35]. Puede que tu universidad o tu empresa haya comprado acceso a algunos de ellos. Casi cualquier cosa escrita por Damien Conway merece la pena ser leída.

Por otro lado, si puedes encontrarlo, Professional Perl Programming es bastante completo, aunque de estilo un tanto más disperso. Sin duda, el capítulo sobre *locale* e internacionalización es el mejor.

Tutoriales

Tampoco faltan los tutoriales en Internet, entre ellos el decano en castellano, hecho por un servidor, un tutorial de Perl[36] cuyo estilo no difiera mucho de este pero que, por su edad (fue escrito cuando Perl iba todavía por la versión 4), quizás haya quedado un tanto obsoleto.

Listas de correo y foros

Hay muchas listas de correo en inglés dedicadas a aspectos específicos: Perl y XML, Perl e Inteligencia artificial, Perl y el Zen; en español la lista `perl-es`[37] está alojada en Yahoogroups y tiene un tráfico moderado (y un nivel de spam relativamente indeseable). Pocas preguntas se quedan sin respuesta. También hay un foro bastante activo Perl en español[38].

En inglés, PerlMonks[39] actúa como un foro, y tiene también un chat incorporado. Sirve tanto para plantear cuestiones como para proponer ideas. Abierto 24/7.

Esto es todo

Bueno, esto es todo, diablillos. Este tutorial tiene licencia GFDL, o sea que puedes hacer con él lo que te dé la gana, grabarlo en piedra, dárselo a tu abuela, traducirlo al telugu, con las condiciones que te marca la licencia. Si encuentras algún fallo, o tienes alguna petición o ruego o pregunta, envíame un correo[40], o cuéntalo en alguno de los foros indicados más arriba; suelo pasar por ahí, o ya pasará alguien que me lo cuente.

Agradecimientos

A Joaquín Ferrero[41], por una lectura y múltiples sugerencias (y corrección de errores), y a jamarier[42], por ser mi beta tester y bug quasher number one. También al proyecto Gutenberg[43], por la copia de El Diablo Cojuelo[44] que he usado y de la que he abusado en este tutorial.

Notas

1. http://gugs.sindominio.net/licencias/gfdl-1.2-es.html
2. perl-apresurados-1.2-src.tgz
3. perl-apresurados.pdf
4. index.html
5. perl-apresurados-ejemplos.tgz
6. http://www.python.org
7. http://www.php.org

8. http://www.perlmonks.org/index.pl?node=Perl%20Poetry
9. http://es.wikipedia.org/wiki/Dia_de_la_Toalla
10. http://www.cpan.org/src/latest.tar.gz
11. http://www.ebb.org/PickingUpPerl/pickingUpPerl.html#The%20Slogans
12. http://www.ruby-lang.org
13. http://www.java.com
14. http://es.wikipedia.org/wiki/llama
15. http://www.windows.com
16. http://www.perl.com
17. http://www.gentoo.org
18. http://eclipse.org
19. http://e-p-i-c.sourceforge.net
20. perl-apresurados-ejemplos.tgz
21. Lo que es recuerdo también de aquellos mismos tiempos en que `STDOUT` era una abadía, en la 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.
22. 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
23. http://search.cpan.org
24. http://www.sysadminday.com/
25. http://es.wikipedia.org/wiki/Bofh
26. http://search.cpan.org/~marcos/Lingua-ES-Silabas-0.01/
27. En cuanto que yo averigüe cómo diablos se hace, lo comunicaré oportunamente.
28. http://google.com
29. Lo que viene siendo un *filehandle* de toda la vida.
30. También se usa `||`, pero la precedencia es diferente.
31. http://search.cpan.org
32. Serían, en realidad, vectores o matrices unidimensionales, aunque no es difícil crear matrices multidimensionales en Perl.
33. 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
34. http://www.oreilly.com
35. http://safari.oreilly.com
36. http://merelo.net/tutoperl
37. http://yahoogroups.com/groups/perl-es
38. http://perlenespanol.baboonsoftware.com/foro/index.php
39. http://perlmonks.org
40. mailto:jjmerelo (en) gmail.com
41. http://aprosi.com
42. http://barbacana.net
43. http://gutenberg.org

44. http://www.gutenberg.org/etext/12457