Los Diez Mitos de Perl

http://www.perl.com/pub/2000/01/10PerlMyths.html

Traducido por Ramiro Encinas Alarza (enero - 2011)

Introducción

Una de las cosas que pasan desapercibidas cuando escuchas hablar de Perl es la gran desinformación que hay. Es realmente muy difícil separar el grano de la paja para alguien que no está familiarizado con Perl, y es muy fácil aceptar algunas de esas cosas como la pura verdad, a veces sin darse cuenta.

Lo que voy hacer es tomar los diez grandes mitos que hay por ahí y responderlos. No voy a tratar de convencer a nadie para que utilice Perl, porque la única forma de convencerse es probándolo. Pero espero demostrar que no es cierto todo lo que se oye.

Primero vamos a asegurarnos de lo qué entendemos por Perl y para qué sirve.

Perl es un lenguaje de programación de propósito general. La respuesta a la pregunta “¿Puedo hacer esto con Perl?” es muy probable que sea “si”. Se utiliza mucho para pequeñas tareas de administración de sistemas, CGI y web, pero eso no es todo, pronto veremos que también puedes utilizar Perl para proyectos a gran escala.

A veces Perl es llamado “lenguaje de scripting”, pero sólo por la gente que no le gusta o no lo entiende. Primero, no existe diferencia real entre programación y scripting, porque ambos dicen al ordenador lo que tú quieres que haga. Segundo, incluso si así fuera, Perl tiene de scripting lo mismo que Java o C. Aquí hablaré de programas en Perl, pero es posible que alguien diga por ahí que son scripts de Perl. Generalmente escriben mejor código los que los llaman “programas”.

Perl es difícil

Lo primero que escuchará de la gente es que Perl es difícil: difícil de aprender, difícil de usar, difícil de entender. Dado que Perl es muy poderoso, la lógica debe ser difícil, ¿no?

Pues no. Para empezar, Perl está basado en una serie de lenguajes conocidos por casi todos los programadores. ¿Sabes algo de C? entonces Perl es sencillo. ¿Estas aprendiendo a programar el shell de Unix? o ¿escribes programas con awk o sed? Si así es, de inmediato se sentirá como en casa con algunos elementos de la sintaxis de Perl.

Incluso si no conoces esos lenguajes, incluso si eres totalmente nuevo en la programación,

todavía puede decirse que Perl es uno de los lenguajes más fáciles para empezar a programar, por dos buenas razones.

Perl se adapta a lo que quieres. Uno de los lemas de Perl es “hay más de una forma de hacerlo”. La gente realiza tareas de formas muy distintas y a veces con soluciones muy distintas para el mismo problema. Perl es muy servicial y no te obliga a utilizar ningún estilo en concreto (a menos que se lo pidas) y te permite expresar tus intenciones de programación de forma que refleja lo que piensas al programar. Un ejemplo: supongamos que tenemos un archivo con dos columnas de datos separadas por dos puntos. Tenemos que intercambiar las columnas. Esto me ocurrió en una discusión el otro día, y así es como yo pensaba hacerlo: leer una línea, intercambio lo que está a los lados de los dos puntos y después muestro la línea.

```
while (<>)
{
      s/(.*):(.*)/$2:$1/;
      print;
}
```

No es un problema difícil de entender, por lo que no debería ser difícil de resolver. Sólo bastan pocas líneas, de hecho, si utilizas algunas opciones de la línea de comandos de Perl puedes quedarte sólo con la tercera línea. Pero no nos vamos a poner demasiado técnicos ahora.

Para aquellos que no saben mucho de Perl, los símbolos `<>` de la primera línea significan “lee línea a línea”, la “`s`” es “sustituye”, los paréntesis quieren decir “recuerda” y `.*` significa “cualquier cantidad de cualquier cosa”.

Por tanto, leemos una línea, realizamos la sustitución y escribimos otra línea como resultado. ¿Qué es lo que sustituimos? Tomamos algo que estamos recordando, seguido de dos puntos, y luego algo más que recordar. Después ponemos el segundo recuerdo, los dos puntos y el primer recuerdo. Esta es una forma bastante natural de pensar.

Sin embargo, otra persona optó por hacerlo de otra manera:

```
while (<>)
{
      chomp;
      ($primero, $segundo) = split /:/;
      print $segundo, ":", $primero, "\n";
}
```

Un poco más largo, pero puede que un poco más fácil de entender, (o no, no sé) pero el punto es que así es cómo pensó resolver el problema esa persona. Así es cómo él lo visualizó y así

es cómo funciona su mente. Perl no impone ninguna forma en particular de pensar cuando programas con él.

Rápidamente: chomp elimina el fin de línea. Entonces separamos lo que hay a ambos lados de los dos puntos (utilizando obviamente y de forma razonable la función split) en dos variables. Por último, ponemos todo junto en orden inverso: el segundo, los dos puntos y el primero, y al final ponemos el fin de línea.

La segunda cosa que hace Perl fácilmente es que no necesitas entender todo para hacerlo bien. Podemos escribir el programa de antes en la línea de comandos, así:

```
% perl -p -e 's/(.*):(.*)/$2:$1/'
```

(-p hace que Perl haga un bucle con los datos de entrada y visualiza el resultado).

No teníamos porqué saberlo. Puedes hacer mucho sabiendo poco de Perl. Obviamente, es más fácil si sabes más, pero también es fácil empezar y utilizar lo que ya sabes para hacer el trabajo.

Tomemos otro ejemplo. Queremos examinar el archivo */etc/passwd* y mostrar algunos detalles de los usuarios. Perl incorpora algunas funciones para leer el archivo de contraseñas, por tanto podríamos utilizarlo, pero aunque no lo sepamos, podemos **hacer** el trabajo con lo que ya sabemos: sabemos cómo leer una entrada de un archivo y separar el contenido a ambos lados de los dos puntos, que es todo lo que necesitamos saber. Hay más de una forma de hacerlo.

Gracias a su similitud con otros lenguajes, el hecho de que no te obliga a pensar de una forma determinada, y el hecho de que saber un poco te lleva bien lejos, podemos decir felizmente que es un mito decir que “Perl es difícil”.

Perl parece una línea distorsionada

Lo siguiente y más común que puedes oir es que Perl es feo, o que es un lenguaje de sólo escritura. Para mi desgracia, hay muchos programas en Perl que son bien feos, pero escribir programas feos no quiere decir que Perl sea un lenguaje feo: los concursos de C ofuscado ya existían antes de los de Perl ofuscado.

Cada vez que veo un trozo de código Perl que parece EBCDIC sobre una línea distorsionada, me detengo y me pregunto “¿cómo puede alguien escribir tan mal?”. Con el tiempo me he dado cuenta de que es fácil abusar de Perl porque Perl es fácil.

Lo que creo que pasa es algo como esto: tienes un archivo de datos para convertirlo a otro formato, y lo necesitas para ayer. ¡Perl al rescate! y en cinco minutos tienes algo que sólo

entiendes tú en ese momento y funciona, no es bonito pero funciona. Lo guardas en algún sitio por si acaso vuelve el mismo problema. Antes o después aparece de nuevo el problema, pero en este caso el formato de entrada varía un poco y sólo necesitas una modificación rápida para conseguirlo. Al final tienes un programa bastante sofisticado. Es tan fácil modificar el que ya tenías que nunca necesitaste crear un enorme conversor. La reutilización del código y el desarrollo rápido se han unido para crear un monstruo.

El problema viene cuando la gente distribuye esto, porque funciona y es útil, y otros lo miran y dicen “Pero hombre, ¿cómo puedes escribir eso?”, y entonces Perl adquiere mala fama.

No tienes porque hacerlo así. Puedes anticipar la jugada y perder más tiempo volviendo escribir el programa, probablemente desde cero, para hacerlo más legible y más fácil de mantener. Puedes sacrificar tiempo de desarrollo por legibilidad, y viceversa.

Verás, es perfectamente posible escribir programas en Perl que son absolutamente claros, claros ejemplos del arte de la programación y mostrar sus inteligentes algoritmos en todo su esplendor. Pero Perl no hace eso, lo haces tú.

En resumen, son las personas las que escriben Perl de forma ilegible, no Perl. Si puedes dejar de ser tú mismo cuando escribes Perl, entonces la mala fama que tiene Perl de parecerse a una línea distorsionada no será más que un mito.

Perl es difícil porque tiene expresiones regulares

Una de las partes en las que Perl ha contribuido al mito de que es un lenguaje ilegible es la forma en que utiliza las expresiones regulares. Como también pasa con Perl en sí, son cosas poderosas, y el poder corrompe. La idea básica es muy simple: lo que estamos haciendo es buscar ciertas cosas en una cadena. ¿Quieres buscar los tres caracteres “abc”?, escribes `/abc/`. Hasta aquí todo bien.

Lo siguiente es la habilidad de encontrar no sólo caracteres exactos, sino también ciertos tipos de caracteres: un dígito se puede encontrar con `\d`, y para encontrar un dígito y luego dos puntos dices `/\d:/`. Ya sabemos que . encuentra a cualquier caracter. También tenemos a `^` y a `$` para indicar el comienzo y el final de la línea respectivamente. Todavía te puedes acostumbrar, ¿no?. Para encontrar dos dígitos al principio de una línea, seguido de cualquier caracter y después una coma, se indica cada cosa en orden: `/^\d\d.,/`

Y así sucesivamente, tan complejo como quieras, dependiendo de la complejidad de lo que quieras encontrar. Lo importante a recordar es que la sintaxis de las expresiones regulares es como cualquier otro lenguaje: para expresarte en él necesitas el hábito de traducir entre él y tu lenguaje nativo hasta que puedas pensar en ese lenguaje. Aunque no lo entiendas a simple vista, puedes leer y entender `/^.,\d.$/`. Al principio de la línea queremos encontrar las siguientes cosas: cualquier caracter, una coma, cualquier dígito, cualquier caracter y el fin de

línea.

Si además de encontrar queremos sustituir, podemos hacer cosas realmente desagradables. Este es mi ejemplo favorito, que es tan relativamente simple como las expresiones regulares. Esto convierte “Mse” y “mse” en “Mes” y “mes” respectivamente:

```
s/\b([mM])se\b/$1es/g
```

Puedes sentarte y leerlo si puedes, pero Perl te permite romper la expresión regular con espacios en blanco y comentarios, y de esta manera no hay razón para no entender las expresiones regulares. A continuación, una versión completamente documentada. Una vez que hayas practicado la lectura y escritura de expresiones regulares, podrás hacer esta expansión en tu cabeza: (y tampoco te llevará mucho).

```
s/\b   # Comienza el límite de una palabra, y
  (    # comienza a recordar
  [mM] # puede ser “m” o “M”,
  )    # termina de recordar,
  se   # y el resto de la palabra
  \b   # fin del límite de la palabra
  /    # sustitúyelo con
  $1   # el caracter original recordado, “m” o “M”
  es   # el resto de la palabra correcta
 /g;   # afecta a toda la cadena dada.
```

Las expresiones regulares pueden parecer complicadas a simple vista, pero una vez aprendido el código y rompiéndolas y reconstruyéndolas como has visto, verás pronto que es una forma natural de expresar una búsqueda de texto como en tu propio lenguaje. Decir que las expresiones regulares hacen que Perl sea difícil, sería un mito.

Perl es difícil porque tiene referencias

Actualmente el siguiente mito son dos en uno: primero, el mito de que Perl no puede hacer frente a las estructuras complejas. Segundo, sólo tienes tres tipos de datos disponibles: el escalar, que son números y texto y se ve así: $foo; el array, que tiene una lista de escalares y se representa como esto: @bar; y el hash, que tiene correspondencias entre escalares y se escribe como: %baz. Los hashes normalmente se utilizan para guardar pares de clave-valor y lo veremos después.

Pelr no es suficiente, porque se dijo que para construir estructuras complejas se necesita “programación real”. Bien, esto ni siquiera es una verdad a medias. Desde que se publicó Perl 5, y esto fue hace cinco años, Perl es capaz de construir estructuras complejas sin referencias y lo veremos en un segundo.

Asi que una vez que lo sepas todo, escucharás lo contrario: las referencias son muy complicadas y se termina con lío de símbolos de puntuación. Curiosamente, las personas que encuentran más complicadas las referencias son las que utilizan C donde las referencias son una especie de punteros que dejan al programador ocupado con la gestión de memoria y otras cosas irrelevantes. Con Perl no tienes que preocuparte de la gestión de la memoria, pues tú y tu tiempo teneis mejores cosas que hacer. Normalmente los programadores de C se confunden haciendo cosas más complicadas de lo que necesitan.

Las referencias son básicamente paquetes planos de datos. Pueden tomar cualquier tipo de dato: hashes, arrays, escalares, hasta subrutinas y ponerlo todo en un escalar que lo represente. Por lo tanto, digamos que tenemos algunos hashes así:

```
%billc =   (
                nombre => "Bill Clinton",
                puesto => "Presidente de US"
           );

%billg =   (
                nombre => "Bill Gates",
                puesto => "Presidente de Microsoft"
           );
```

Efectivamente es molesto tener una variable para cada persona, y además hay muchos Bills en el mundo, yo tengo cuatro al mes y eso ya es bastante. Lo ideal es poner a todos juntos en un array. ¡Pero hay un problema!, los arrays sólo pueden contener escalares, no hashes. No hay problema: utilizamos una referencia a un paquete plano donde cada hash es un escalar. Esto lo hacemos añadiendo una barra invertida antes del nombre:

```
$billc_r = \%billc;
$billg_r = \%billg;
```

Ahora tenemos dos escalares que contienen todos los datos en nuestros arrays. Para desempaquetar los datos y llegar al hash, le quitamos la referencia: dile a Perl que quieres un hash. Sabemos que los hashes comienzan con `%`, por tanto pongamos un `%` delante del nombre de nuestra referencia:

```
%billc puede accederse via %$billc_r
%billg puede accederse via %$billg_r
```

Y ahora podemos guardar estas referencias en un array:

```
@bills = ( $billc_r, $billg_r )
```

o

```
@bills = ( \%billc, \%billg )
```

¡Listo! un array con dos hashes: una estructura compleja de datos.

Por supuesto que hay un par de trucos más: crear referencias a arrays y a hashes directamente en lugar de tomar una referencia a una variable existente, y llegar directamente a los datos en una referencia en lugar de desreferenciar mediante una variable nueva. (¿Ves la simetría?).

Como antes, para hacer cosas no necesitas saber todo sobre el lenguaje. Es verdad que el código es más claro y corto en caso de no utilizar variables temporales innecesariamente, pero en caso de no saber cómo hacerlo, no te impide utilizar referencias.

Por supuesto que no tienes porqué entender completamente ahora las referencias. No tienes porqué saber si son útiles o no; de hecho, si sólo escribes programas simples que tiran texto, es posible que nunca las utilices.

Esperemos que ahora puedas navegar entre Scila que dice que no puedes tratar con datos complejos y Caribdis que dice que sí puedes pero es irremediablemente confuso, y ver que al igual que el resto de la Odisea, es sólo un mito.

Perl es sólo para Unix

¿Sólo para Unix? He escuchado esto y me resulta difícil responder de forma seria. La distribución estándar de Perl soporta 70 sistemas operativos. Existen proyectos por separado para Windows y para Macintosh y muchos otros sistemas más. Es complicado encontrar en estos días un ordenador donde Perl no funcione: Perl ahora corre en Psion Organiser y está cerca de ser incluido en el Palm Pilot.

Esto significa que Perl es uno de los más, sino **el** lenguaje más portable que hay. Un programa correctamente escrito no necesita ningún cambio desde Unix a Windows NT y tampoco a Macintosh, y uno escrito incorrectamente necesitará posiblemente cambiar unas tres o cuatro líneas.

Definitivamente Perl no es sólo para Unix. Esto es simplemente puro mito.

Perl sólo es para scripting, no puedes hacer programas “de verdad” con él

La misma clase de gente que dice que Perl es sólo un lenguaje de scripting probablemente te diga que Perl no es viable para la “programación seria”. Nunca escribirías un sistema operativo con Perl, por tanto eso no puede ser bueno.

Bueno, es posible que no necesite escribir un sistema operativo con Perl. Conozco a una o dos personas que lo están intentando, pero son unos freaks. En cualquier caso, esto no quiere decir que no puedas construir programas grandes, importantes y sofisticados con Perl. Perl sólo es un lenguaje de programación.

La gente ha escrito cosas muy grandes con Perl: por ejemplo, Slashdot, demuestra que Perl puede hacer frente a una buena cantidad de carga, los datos del Proyecto del Genoma Humano, y un innumerable de servidores y clientes de red. Son muy comunes los programas escritos en Perl que tienen cientos o miles de líneas.

Además, puedes ampliar Perl para utilizar cualquier librería de C. Cualquier cosa que puedas hacer con C lo puedes hacer con Perl y mejor. Sí, es un buen lenguaje de scripting y manipulación de datos, pero eso no es todo lo que Perl sabe hacer.

Decir que Perl no está preparado para la “programación seria” denota desconocimiento sobre Perl o habría que hacerse la pregunta “¿qué es la programación seria?”, en todo caso, un mito.

Perl es sólo para CGI

Ah, la gran confusión de CGI/Perl. Dado que Perl es el mejor lenguaje para crear contenido web dinámico, la gente ha creado más confusión con las diferencias entre Perl y CGI.

CGI es sólo un protocolo, una forma de que un servidor web y un programa hablen el mismo lenguaje. Puedes utilizar Perl para hablar ese protocolo, que es lo que hace mucha gente. Puedes utilizar C para hablar CGI, si quieres. He escrito programas que hablan CGI con INTERCAL, porque me gusta. No hay nada concreto de Perl para CGI.

Tampoco hay nada concreto de CGI para Perl. Debido a que Perl está especialmente adaptado a la web, mucha gente lo utiliza, pero como hemos visto, la gente con Perl puede hacer y está haciendo muchas más cosas a parte de la web. Perl ya daba saltos cuando la Web estaba en pañales.

¿CGI es Perl? ¿Perl es CGI?. Es todo un montón de mitos.

Perl es demasiado lento

Lo mismo has escuchado alguna vez que Perl es lento para cualquier cosa.

En algunos casos, puede ser bastante lento en comparación con C: C puede ser hasta 50 veces más rápido que su programa equivalente en Perl. Pero depende de muchas cosas.

Depende de cómo escribas tu programa; si lo escribes con el estilo de C, será más lento que su

equivalente escrito con expresiones de Perl. Por ejemplo, podrías mirar un string caracter por caracter, como lo harías en C. En este caso posiblemente haces más trabajo que si lo hicieras con una expresión regular. Cuanto menos le pides a Perl, más rápido lo ejecuta.

A veces y sin embargo, hay cosas que C las encuentra complicadas y Perl no; la manipulación de cadenas es una de ellas, porque Perl te permite pensar en cosas a nivel de cadenas, en lugar de obligarte a verlas caracter por caracter.

Sin embargo, existen ocasiones donde C gana en términos de tiempo de ejecución. Pero si escribes software para tí mismo, tienes que considerar otro factor: el tiempo de desarrollo. La cantidad de tiempo y energía emocional que gastas programando es importante para tí, y, como los programadores cuestan bastante dinero actualmente, el que te paga también le cuesta.

Tomemos un ejemplo muy simple para ver cómo funciona: tenemos un documento con una serie de puntos numerados. Hemos añadido otro punto en la línea 50, y después de este punto, cada número en el comienzo de la línea se tiene que incrementar en uno. Prefiero gastar unos cuantos segundos en cocinar un poco de Perl como esto:

```
% perl -p -e 's/^(\d+)/1+$1/e if $. > 50'
```

en lugar de pasar media hora rompiendome la cabeza con C.

Por supuesto, es posible que no necesitemos la velocidad de C para un ejemplo simple como éste, pero el principio se extiende también a grandes programas: **puedes** escribir un programa que se ejecute rapidamente en C, pero **siempre** podrás escribir programas más rápido en Perl.

Perl no es seguro

¿La seguridad? Tiene que haber algunas grietas en la armadura. La gente puede leer el código fuente de tus programas, por tanto ¡eres vulnerable a todo tipo de problemas!

Es cierto que el código fuente de un programa escrito en Perl tiene que estar disponible para que el intérprete de Perl lo ejecute, pero esto no quiere decir que Perl sea inseguro. Eso podría significar que todo lo que escribes es inseguro y que puedes pensar que sería mejor ocultar las deficiencias, pero en eso sólo cree Microsoft y algunos otros. La seguridad por oscuridad no es muy segura.

Al igual que la legibilidad del código y el maravilloso bug Y2K, no puedes pedirle a Perl que escriba por tí. Perl no hace que escribas programas seguros de forma mágica, bueno, el modo antienvenenamiento “taint mode” te protege si escribes código inseguro, pero eso no sustituye el escribir adecuadamente.

Si quieres de verdad, puedes intentar ocultar el código fuente; puedes utilizar filtros, puedes

intentar compilarlo con el compilador de Perl, pero nada de esso te asegura que no pueda ser descifrado o decompilado y nada de eso te solucionará ningún problema. Lo mejor siempre es escribir código seguro.

Por lo tanto, puedes escribir de forma insegura, pero ¿Perl es inseguro en sí?. No, es otro mito.

Perl no es comercial, y eso no puede ser bueno

Para terminar, puedes ser uno de los que se quejan de que, como Perl no es un software comercial, no puede ser bueno. No hay soporte, la documentación la dan voluntarios, y todo eso.

Me sorprende que en el mundo de Linux, Apache y Samba y otras tecnologías líderes, la gente todavía pueda pensar así. Pero no debería sorprenderme, porque los vendedores quieren que la gente piense de esa forma, los vendedores invierten un montón de dinero para generar miedo, para que la gente siga pensando así.

Podría perder mi tiempo diciendo que Perl es un buen producto porque el soporte lo dan voluntarios que les gusta Perl y no cobran por ello, pero no voy a entrar en filosofías sobre la naturaleza del desarrollo de los voluntarios. Vayamos a lo hechos.

La distribución estándar de Perl contiene más de 70.000 líneas de documentación, lo que realmente debería ser suficiente para cualquiera. Si no, en la red hay innumerables tutoriales disponibles. A esto hay que añadir todas las líneas de documentación de los módulos CPAN, y con todo esto tendremos una buena base. Y eso es sólo el material de libre disposición. Actualmente hay unos 250 libros dedicados a Perl, y es posible que muchos más.

La documentación no es algo que sea un problema.

¿Y el soporte? Bien, si has leido toda la documentación y todavía tienes un problema, hay por lo menos cinco grupos de noticias en Usenet dedicados a Perl, y al menos un canal IRC. Todos ellos integrados por voluntarios, por tanto, no serán amables contigo si antes no has leído los FAQs. Esto no quiere decir que no sean útiles: algunos de los grandes nombres de Perl están colgados por ahí. Es posible que encuentres respuestas a tus preguntas si muestras el suficiente sentido común.

Por supuesto, es posible que necesites más: miles de empresas dan cursos de Perl y puedes contratar soporte real, comprar paquetes de Perl y todo lo que quieras para que tu jefe se sienta cómodo. Ser libre no quiere decir que no sea comercial, y la idea de que no vale la pena porque es libre no es más que un mito.

Conclusión

Estos no son todos los mitos que puedes escuchar de Perl; no tengo tiempo para mencionarlos a todos, pero hay mucha desinformación ahí fuera. Si has escuchado alguna de estas cosas que he mencionado, dale otro vistazo a Perl; es más fácil de lo que crees, es más rápido de lo que piensas, y es mejor de lo que imaginas. No hagas caso a los mitos, ni siquiera creas en mi palabra. Sal ahí fuera y compruébalo tú mismo.