Taller de procesamiento de textos con Perl: una mini-introducción

Rogelio Nazar
Pontificia Universidad Católica de Valparaíso
Versión:

5 de mayo de 2018

1. Introducción

Este documento tiene el propósito de servir como apoyo a la presentación oral del taller de Perl de tres sesiones que se dictó del 24 al 27 de abril de 2018:

http://www.tecling.com/tallerPerl

Se pretende preservar lo esencial de los contenidos presentados, obviando el proselitismo y la perorata ideológica que contextualizaron la presentación oral, en que se explicaba para qué sirve el programa y cómo se puede aplicar en lingüística.

Perl es un programa y a la vez un lenguaje de programación. Es un lenguaje porque escribimos 'scripts', es decir instrucciones utilizando una sintaxis que la computadora es capaz de entender. Estas instrucciones las guardamos en un documento de texto plano, como el .txt, y no como los documentos de word o pdf, que son documentos binarios. El lenguaje se parece al inglés, pero es un inglés controlado, es decir que no tiene la misma libertad y expresividad que ofrece una lengua natural. Y es un programa porque Perl es el 'compilador', es decir el programa que lee nuestro documento de texto con las instrucciones y lo 'interpreta' o traduce a otro lenguaje aun más abstracto que es el que en realidad es capaz de seguir la computadora. Aquí vamos a obviar esos detalles y tecnicismos porque escapan del foco de interés.

Las instrucciones del script se leen de manera secuencial y en ellas encontramos por un lado referencias a estructuras de datos, que son las que llamamos 'variables', y una serie de comandos para especificar operaciones a realizar con esas variables. Entre ellos destacan los que llamamos 'operadores de flujo'. El término presenta la imagen de los scripts como si fueran cañerías por las que pasan los datos. Todo proceso algorítmico puede ser visto como un flujo, porque tiene una entrada o 'input' y una salida o 'output', que son datos que entran y un resultado que sale.

Se describen además varios comandos que se pueden utilizar para el tratamiento de datos textuales. En la última sección se integra todo en un proceso relativamente complejo para contar las palabras de un documento. Hacia el final del taller se vio muy rápidamente el tema de las expresiones regulares, que se pueden aplicar tanto para la búsqueda de datos en corpus como para la transformación o anotación de textos. Este tema no se incluye en este resumen pero están disponibles los materiales de otro taller dedicado a esa materia específica:

http://www.tecling.com/tallerPerl/unidad4_2014.pdf

2. Hola Mundo

Para comenzar, vamos a presentar un script muy sencillo que no tiene ni variables ni operadores de flujo.

Antes de eso, un pequeño tono acusatorio: desaconsejo el uso de Windows porque este sistema operativo no es tan adecuado para Perl o el procesamiento de textos en general como los sistemas operativos que emulan al antiguo UNIX, es decir cualquiera de la familia de Linux, OSX o FreeBSD. Esto es porque estos sistemas representan toda una metodología de trabajo y ofrecen una gran cantidad de herramientas. En Windows no se da eso porque no es una comunidad de desarrolladores independientes y las herramientas son más limitadas. También es más complicada la integración de estas herramientas y entrañan el costo de sus licencias de uso. Los que, contra mi consejo, se empeñen en usar Windows para esto, tendrán que descargar el programa desde un sitio como este:

http://www.activestate.com/

Y también necesitarán un procesador de texto plano, como Notepad++:

https://notepad-plus-plus.org/

En el procesador de texto vamos a escribir nuestra primera línea de código:

```
print "\nHola Mundo \n";
```

Los `\n` simplemente representan un salto de línea, para dejar una línea en blanco antes y otra después de la leyenda. Tenemos que guardar este documento con un nombre como holamundo.pl y luego ejecutar ese script en la terminal o línea de comando. Quienes no son usuarios de Windows probablemente ya sepan cómo abrir una terminal. Caso contrario, se presiona la tecla Windows y se escribe 'cmd' en la entrada de texto para la ejecución de comandos que se aparece en la parte inferior del menú. Esto hará aparecer la terminal, un cuadrado negro con un cursor titilante.

Desde allí tenemos que navegar por nuestro sistema de ficheros hasta el lugar donde hemos guardado nuestro script. Por ejemplo, si lo guardamos en el escritorio, tendremos que escribir allí:

```
cd Desktop
```

y a continuación darle a la tecla 'enter'. El comando 'cd' (*change directory*) sirve para cambiar de carpetas. El escritorio es también una carpeta que está en el 'home' del usuario. Para salir de una carpeta e ir a un directorio superior utilizamos: `cd ..` y para subir dos directorios `cd ../..`, etc. Para crear un directorio, por ejemplo, un directorio nuevo aquí en el escritorio, donde estamos, sería:

```
mkdir tallerPerl
```

Esto creará una carpeta llamada 'tallerPerl' en el escritorio. Allí podríamos guardar nuestros scripts. Si efectivamente lo hemos guardado al script en esa carpeta, tenemos que continuar navegando hasta esa carpeta, con

```
cd tallerPerl
```

Una vez que hemos llegado a la carpeta donde está el script, ejecutaremos el siguiente comando:

```
perl holamundo.pl
```

En realidad no hace falta escribir 'holamundo.pl' completo. Si no hay en la misma carpeta otro fichero que comience igual, basta con escribir las primeras letras y luego presionar la tecla 'tabulador' para que se autocomplete el nombre.

El enter hará que el compilador ejecute el script y aparezca en la pantalla la leyenda 'Hola Mundo'. De momento no es muy exitante pero en un rato, cuando incorporemos al script variables y comandos, se pondrá mejor.

3. Variables

Encontramos variables continuamente en la vida cotidiana. Los que tuvieron la suerte de ir a la escuela recordarán haber hecho ejercicios con ecuaciones para 'despejar la x'. Esa 'x' es una letra que representa un dato numérico que no que conocemos. En Perl las variables representan lo mismo, solo que el valor de x no se limita a datos numéricos sino también textuales o bien estructuras de datos como listas o tablas. Las variables sirven para retener información en la memoria RAM de la computadora, es decir sin escribir esta información en forma de ficheros en el disco.

Vamos a modificar nuestro script para definir en él una variable:

```
$x = "Sr. Patata";
print "\nHola $x \n";
```

Si los datos son numéricos no necesitan comillas, que son requeridas solo en el caso de datos alfanuméricos.

Al resultado de la ejecición ya no será 'Hola Mundo' sino 'Hola Sr. Patata'. Naturalmente, no es necesario definir la $x de esa manera. Tenemos otras opciones igualmente válidas como 'Sr. Sapo Pepe', 'Pepe the Frog' o similares. Lo natural, sin embargo, es que las variables no se definan en el mismo script, sino que provengan de fuera, cosa que se verá más adelante.

El nombre de la variable, a su vez, no necesariamente debe ser una sola letra. Puede estar compuesto por cualquier secuencia arbitraria de caracteres siempre y cuando no contenga tildes o espacios en blanco.

Lo esencial aquí es comprender que esa variable puede contener un dato cualquiera. Si ese dato es una sola entidad, como un número cualquiera o la secuencia de caracteres que conforman el nombre 'Sapo Pepe', entonces hablamos de un 'scalar', y lo representamos con el signo dólar: $.

Este es el tipo de variable más sencillo que tenemos en Perl. El siguiente tipo es la lista o 'array'. Lo representamos con una arroba:

```
@x = ("Sr. Patata", "Sapo Pepe", "Pepe the Frog");
print "\nHola $x[0] \n";
```

El resultado de la ejecución de este último script será muy similar al anterior. Aquí lo que ocurre es que en la primera línea definimos un array @x que tiene tres elementos separados por comas. En la segunda línea imprimos solo el primer elemento de la lista. Hay que notar el uso de corchetes en $x[0] y el cero que hace referencia al primer elemento, ya que algo esencial en toda la lista es que los elementos aparecen en un orden fijo. Si en lugar de un cero escribimos un 1, se imprimirá en cambio el segundo elemento. El otro detalle que hay que notar es que cuando mencionamos la variable para referirnos solo a un elemento, esta aparece precedida por el signo dólar, como en el caso del scalar, y no la arroba.

El tercer y último tipo de variable en Perl es el 'hash', que es una tabla de dos columnas, es decir que es una estructura de datos que organiza la información en forma de pares atributo-valor ('keys-values'). Vamos a comenzar también a utilizar números de línea para mostrar el código en este documento, ahora que los scripts comienzan a ser un poquito más extensos. Pero esto no quiere decir que haya que añadir números de línea, están solo para referirnos a cada línea específica. Toda línea que comience con este caracter que tiene tantos nombres como países (gato, almohadilla, numeral, hashtag) se considera un 'comentario', es decir que lo ignora el compilador, porque están destinados al lector humano.

```
1 # esto es un comentario
2 %x = ("Sr. Patata" => 60,
3   "Sapo Pepe" => 45,
4   "Pepe the Frog" => 15
5   );
6 print "\n Hola \$x{'Sr. Patata'}\n ";
```

Aquí, a diferencia del caso anterior, notamos que el nombre de la variable viene precedido del signo de porcentaje en la definición de la variable, y que los datos en un hash no tienen orden, ya que nos referimos a los elementos mediante el nombre del atributo. Otra diferencia es que no usa corchetes sino llaves. Si ejecutamos el script, lo que aparecerá ahora será el valor correspondiente a ese atributo, es decir el número 60.

Estos tres tipos de variables a su vez se pueden integrar y de manera recursiva, es decir que un array por ejemplo puede tener como valores hashes o viceversa, o bien otro array, etc. No hay límite alguno para la dimensionalidad de estas estructuras.

4. Operadores de flujo

En esta sección vamos a concentrarnos en dos operadores que son esenciales para cualquier proceso. Estos son el 'bucle' (*loop*) y la 'condición' (*if*). Comenzaremos con el primero.

Los bucles son fundamentales para la automatización de procesos. Con frecuencia tendremos operaciones que son rutinarias o cíclicas, como por ejemplo, descargar distintos documentos de internet, o procesarlos, convertirlos de pdf a texto o viceversa, etiquetarlos, etc. Es imposible que en el procesamiento de datos lingüísticos no tengamos que hacer un bucle alguna vez.

El comando más sencillo para hacer un bucle es el 'foreach', y su estructura es la siguiente:

```
1 @x = ("Sr. Patata", "Sapo Pepe", "Pepe the Frog");
2 foreach $i (@x) {
3   print "\n Hola $i";
4 }
```

Aquí, en la línea 1 simplemente definimos un array, tal como ya habíamos hecho antes. Lo que nos interesa está entre las líneas 2 y 4. En la línea 2 aparece el foreach y le sigue un scalar $i. Luego viene entre paréntesis el array @x y luego un par de llaves, una apertura y otra de cierra, que encierran el código que se debe ejecutar. Se puede interpretar como "por cada i de la lista x,

hacer lo siguiente". El código entre llaves se va a ejecutar tantas veces como elementos tenga el array @x, tres en este caso. Y en cada vez, en cada ciclo, el scalar $i irá cambiando su valor. Así, si ejecutamos el script comprobaremos que imprime "Hola" seguido de cada uno de los elementos de @x. Hay que notar, además, el tabulador que precede al print de la línea 3. Este tabulador no es necesario para el compilador, pero es de gran importancia para indentar el fragmento de código que aparece entre llaves. Es muy importante respetar esta convención para hacer nuestro código más legible, tanto para compartirlo con nuestros compañeros, como para nosotros mismos dentro de unos años, cuando hayamos olvidado completamente qué era lo que hacía el script.

También podemos hacer un bucle para imprimir todo el contenido de un hash, aunque en este caso tendríamos un par de pequeñas diferencias. En el código que aparece a continuación, en la línea 5 debemos ahora aclarar si el valor de $i en cada ciclo será el de los atributos (keys) o el de los valores (values) del hash. Lo que tiene más sentido, en general, es hacerlo por keys, porque es lo que nos permite hacer referencia luego a sus valores, que es lo que hacemos en la línea 6 con $x{$i}, que representaría la edad de estos personajes.

```
1 %x = ("Sr. Patata" => 60,
2   "Sapo Pepe" => 45,
3   "Pepe the Frog" => 15
4   );
5 foreach $i (keys %x) {
6   print "\n La edad de $i es $x{$i}. ";
7 }
```

Pasaremos ahora a las condiciones. Las condiciones tienen una estructura muy similar a la del bucle, en tanto que ambos usan paréntesis y llaves.

```
1 if ($x > 18) {
2   print "\n La variable es mayor que 18. ";
3 }
```

Este código muestra un condicional bastante simple. Entre paréntesis se defina la condición que se está evaluando. En este caso, si el contenido de la variable $x es mayor que 18. La evaluación puede tener dos resultados, positiva o negativa. Solo si la evaluación es positiva, entonces se ejecuta el código que aparece entre llaves.

También podemos enriquecer este código con el 'else', que especifica un nuevo par de llaves con el código que se tiene que ejecutar en caso de que la condición se evalúe como negativa.

```
1 if ($x > 18) {
2   print "\n La variable es mayor que 18. ";
3 } else {
4   print "\n Resulta que la variable no es mayor que 18. ";
5 }
```

También, entre ambos, podríamos tener un 'elsif' (¡así se escribe!), para especificar distintas posibilidades. Por ejemplo, si la variable menor, mayor o igual que 18. Hay que notar, en la línea 4, cómo se evalúa la igualdad con '==', que es distinto a la asignación de un valor a una variable, con '='. Además, notamos nuevamente los tabuladores en las líneas que están entre llaves.

```
1 print "\n La variable es ";
2 if ($x > 18) {
3   print "\n mayor que 18. ";
4 } elsif ($x == 18) {
5   print "\n igual a 18. ";
6 } else {
7   print "\n menor que 18. ";
8 }
```

Podemos tener un número ilimitado de comandos 'elsif' entre el 'if' y el 'else', pero siempre vamos a tratar de evitar la excesiva complejidad de los scripts. La concisión es una virtud en esto de escribir código.

Normalmente, en los scripts estaremos intercalando bucles y condiciones, como en el siguiente ejemplo:

```
1 %x = ("Sr. Patata" => 60,
2   "Sapo Pepe" => 45,
3   "Pepe the Frog" => 15
4   );
5 foreach $i (keys %x) {
6   if ($x{$i} >= 18) {
7     print "\n El $i es mayor de edad. ";
8   }
9 }
```

Algo que notar aquí es el signo >= en la línea 6, que significa 'mayor o igual que', y el doble tabulador en la línea 7, que significa que estamos dentro de un par de llaves, que a su vez está dentro de otro par de llaves.

Una síntesis entre bucles y condicionales es el bucle 'while', que ejecuta un código mientras se evalúe una condición. Con este bucle aprendemos de paso a incrementar el valor de una variable.

```
1 $x = 0;
2 while ($x < 10) {
3   $x++; # esto es equivalente a decir $x = $x + 1;
4   print "\n $i ";
5 }
```

Cuando lo que se comparan no son entidades numéricas sino alfanuméricas, entonces el signo que usamos en la comparación son los <>. De lo contrario, se utiliza `eq` para la equivalencia y `ne` para la desigualdad. Así, podemos tener:

```
1 $x = "Pepe";
2 if ($x eq "Pepe") {
3   print "\n La variable es igual a 'Pepe'"; # ya nos enteramos antes
4 }
```

5. Comandos varios

Hemos estudiado comandos varios. Estudiamos algunos que sirven para transformar un tipo de variable en otro. Un ejemplo de ello fue 'split', que divide un scalar para convertirlo en un array.

```
1 $x = "That which we call a rose by any other name would smell as sweet";
2 @x = split / /, $x;
3 foreach $i (@x) {
4   print "\n $i ";
5 }
```

El comando split admite dos argumentos. Primero, el tipo de caracter que se utilizará para segmentar la oración, que en este caso es el espacio en blanco representado entre dos barras / /, y el segundo argumento es el escalar que es sometido a este tratamiento. Aquí subrepticiamente entra el tema de las expresiones regulares, porque entre las barras puede ir también un par corchetes encerrando no solo espacios en blanco sino también otros elementos como comas, puntos y comas, dos puntos, y entre otros signos de puntuación, el punto, aunque este tiene que ir 'escapado' con una barra invertida, porque en las expresiones regulares, el punto es el caracter comodín, y significa cualquier caracter. Al escapar el caracter con la barra le especificamos que lo que nos interesa es literalmente un punto. Los corchetes definen así un determinado paradigma, en este caso de puntuación, aunque claramente está incompleto:

```
@x = split /[ ,;\.]+/, $x;
```

El contrario de split es 'join', con estructura similar al anterior solo que el primer argumento es el caracter que servirá de 'pegamento' entre los elementos (típicamente un espacio en blanco):

`$x = join ' ', @x;`

Estudiamos también comandos para llevar todas las letras a mayúsculas con el comando 'uc', por *upper case*:

`$x = uc $x;`

o a minúsculas con 'lc', por *lower case*:

`$x = lc $x;`

También nos interesamos por el comando 'sort', que sirve para ordenar los contenidos de una lista. Por defecto, ese orden es alfabético:

`@x = sort @x;`

Esto hará que los elementos de @x queden ordenados alfabéticamente, y el siguiente comando lo haría en el sentido inverso:

`@x = reverse sort @x;`

También es posible ordenar los elementos en tanto entidades numéricas y no por orden alfabético. En ese caso hay que especificarlo de la siguiente manera:

`@x = sort {$a <=>$b} @x;`

Con esto de $a <=>$b le estamos diciendo que tiene que ser en order creciente. Podríamos también hacerlo al revés invirtiendo el orden de estas dos variables ($b <=>$a).

6. Entrada y salida de datos

Hasta ahora hemos estado definiendo el valor de las variables dentro del mismo script, pero lo normal es que el valor de las variables sea un dato que provenga de una fuente externa al script. Esto es lo que llamamos la entrada o 'input'.

Una manera de pasarle un parámetro a un script es por la terminal, al momento de ejecutarlo. En el siguiente ejemplo, vemos una referencia a una variable (array) llamado @ARGV.

`print "\nHola $ARGV[0]!!";`

Podemos probar ejecutar un script `hola.pl` solo con esa línea, del siguiente modo:

`perl hola.pl Pepe`

El resultado será la leyenda "Hola Pepe". Ese $ARGV[0] hace referencia al primer elemento que se le pase al script como argumento mediante la línea de comando, "Pepe" en este caso. Podría así haber un segundo elemento (separados por espacios en blanco) de manera tal que podríamos referirnos a él como

`print "\nHola $ARGV[0] $ARGV[1]!!";`

ejecutable como:

`perl hola.pl Sapo Pepe`

Una manera de hacer salir datos es dirigir el output a un fichero desde la terminal, con el signo 'mayor que':

`perl hola.pl Sapo Pepe >SalidaProceso.txt`

Ejecutado de esta forma, el script no producirá resultado en pantalla sino que generará un nuevo fichero SalidaProceso.txt (si ya existía, borrará lo que había sin preguntar, y aquí no hay papelera de reciclaje).

Otra manera de hacer que el script reciba datos es hacerle leer un fichero. Podemos copiar y pegar una noticia del diario en un fichero de texto plano y guardarlo como `noticia.txt`.

El siguiente código leerá ese documento e imprimirá el contenido línea por línea.

```
1 open( $fh, '<', "noticia.txt" ) or die "\n No encuentro el fichero $filename!";
2 while ( $line = <$fh> ) {
3   print $line;
4   sleep 1;
5 }
6 close $fh;
```

En la línea 1 está el comando 'open', que es con el que abrimos el fichero, y que require tres argumentos: un scalar con el cual referirse al contenido del documento, una 'dirección' que especifica si abrimos para leer (representado con el signo 'menor que') o para escribir (con el signo 'mayor que') y un tercer agumento es el nombre del fichero, que también hubiéramos podido pasar como $ARGV[0] según lo expuesto anteriormente.

Esto que sigue con 'or die...' es por si algo fallara. El 'or' sirve para dar una instrucción para el caso de que falle otra. Y el 'die' es para interrumpir la ejecución dando un mensaje de error. Lo que sigue, entre las líneas 2 y 5, es un bucle en el que vamos leyendo las líneas de $fh y las vamos imprimiendo.

Ponemos un 'sleep' en la línea 4, que es un comando que hace 'dormir' a la máquina una cantidad determinada de segundos, para que vayan apareciendo las líneas poco a poco. Lo de ralentizar un proceso con sleep no es algo que uno vaya a querer hacer muy a menudo. Más bien todo lo contrario. Está aquí con propósito meramente didáctico.

A la inversa, si lo que se desea es escribir un fichero, cambiamos la dirección del segundo argumento. El el código siguiente, en la línea 2, imprimimos una leyenda pero el print viene seguido de $fh, lo que indica que tiene que escribir eso en el fichero y no en la pantalla.

```
1 open( $fh, '>', "output.txt" );
2 print $fh "\n Hola Sapo! ";
3 close $fh;
```

7. Integrando todo en un script más complejo

Luego de haber comentado varios comandos, podemos ahora intentar integrar todo en un script que realice alguna tarea compleja, como por ejemplo contar las palabras de un documento. En el mismo script aparecen los comentarios explicativos de lo que vamos haciendo paso a paso.

```
$filename = "noticia.txt";
open( $fh, '<', $filename ) or die "\n No encuentro el fichero $filename!";
while ( $line = <$fh> ) {
  chomp $line; # este chomp sirve eliminar el caracter de retorno de carro
  @tokens = split /[ .,;\.\?]+/, $line; # con esto dividimos en palaras.
  # Naturalmente faltan varios signos, dejamos esos para simplificar
  foreach $i (@tokens) {
    # con este bucle interno recorremos los tokens
    # y registramos su frecuencia en este hash:
    $types{$i}++; # con esto se incrementa en 1 el valor de cada token
  }
}
close $fh;
foreach $i ( sort { $types{$b} <=> $types{$a} } keys %types) {
  print "\n$i\t$types{$i}";
  # con esto imprimimos el listado de palabras junto a su frecuencia
  # separando con un tabulador
  # y en orden decreciente de frecuencia, mediante $types{$b} <=> $types{$a}
}
```

Tampoco es tan compleja la operación de contar palabras, pero el script ya permite hacerse una idea del potencial de la herramienta, y cómo se pueden combinar las piezas para realizar tareas más interesantes.

8. Reflexiones finales

Lo que hemos visto en este taller es apenas el principio de lo que hay y lo que se puede hacer. La idea aquí ha sido solo comenzar a entender la lógica. Muchas cosas quedaron afuera por falta de tiempo, y en verdad son tan complejas que más vale tenerlas en otro taller. Ha sido tan laxo y suave –apenas un paseo por el campo– que ni hemos llegado nunca a hablar de `use strict;` que hace que el compilador se ponga 'estricto' y no tolere ninguna vaguedad en el código.

Quienes puedan dominar estos conceptos y técnicas, van a adquirir de a poco un gran poder, y es muy satisfactorio comprobar que mediante la automatización de procesos complejos se pueden conseguir grandes cosas. También, es preciso aceptar que con grandes poderes vienen también grandes responsabilidades, y que es necesario resistir a la tentación de atacar servidores o casillas de correo con envíos masivos y ese tipo de maldades. Como reflexión final me gustaría decir: imagínense todo lo que se puede hacer.