

Perl: Expresiones Regulares Funciones Archivos

Expresiones Regulares

- Una de las grandes fortalezas de Perl:
 - Permiten realizar “pattern matching” y sustituciones
- Matching:
 - Lo más básico es buscar una pieza de texto dentro de un string
 - Tradicionalmente la expresión a buscar se encierra con “/”
 - Para aplicarla a una variable se utiliza:

```
$var =~ /exp/
```

- Retornará verdadero si “exp” está en \$var

Expresiones Regulares

- También se puede utilizar !~, que es la negación de =~
 - Será verdadero si “exp” no está en \$var
- ```
$var = "manzanas rojas y verdes";
print "hay rojas" if $var =~ /rojas/;
```
- Las expresiones regulares son case-sensitive, para evitarlo:
- ```
print "hay rojas" if $var =~ /rojas/i;
```

Expresiones Regulares

- Hay caracteres especiales para las expresiones regulares:
 - “^” indica que la evaluación se realiza al inicio del string
 - “\$” indica que la evaluación se realiza al final
 - “?” calza un carácter 0 o 1 vez
 - “*” calza un carácter 0 o más veces
 - /j*/ **siempre calza** aunque no existan “j” en la variable, pues calza 0 veces
 - “+” calza al menos una vez

Expresiones Regulares

- Ejemplos:

/shoo?t/ calza con shot o shoot
 /sho+t/ calza: shot, shoot, shoooot..
 pero no con sht
 /sho*t/ calza con sht, shot, shoot..

- Otros tipos especiales son:

- “\t” calza un tab
- “\n” calza un salto de linea
- “\s” calza cualquier cosa que parezca un espacio

Expresiones Regulares

- Tipos especiales:

- “\w” calza un carácter alfanumérico
- “\d” calza un dígito

- Todos los anteriores pueden ser negados por:

- \S: calza un no-espacio
- \W: calza algo que no sea alfanumérico
- \D: calza algo que no sea dígito

- Ejemplo:

/push\s*chair/ calza “push” seguidos por cero o
 más espacios o tabs y luego
 “chair”

Expresiones Regulares

- Ejemplos:

/number\s*\d+\s/ calza “number” con algún espacio opcional
 y uno o más dígitos seguidos por un
 espacio en blanco

/^e.*d\$/ calza “e” al comienzo de una linea, luego cualquier
 cosa y una “d” al final de la linea

/\sPerl\s/ calza la palabra “Perl” rodeada por un espacio
 /\bPerl\b/ Con \b delimitamos la Palabra que queremos
 calzar, no se permiten caracteres de palabras,
 solo espacios, signos de puntuación

Expresiones Regulares

- Utilizando paréntesis Perl coloca los calces en las variables \$1, \$2, etc, al estilo de sed

```
$test = "el dijo ella dijo";
$test =~ /\b(\w+)\b/ # $1 es "el"
$test =~ /(di.*o)/; # $1 es "dijo ella dijo"
```

- Para este último ejemplo, Perl busca el patrón “di” y sigue buscando el calce más grande
- Para que calce sólo “dijo”:

```
$test =~ /(di.*?o)/; # $1 es "dijo"
```

Expresiones Regulares

- Cuando se utilizan paréntesis, Perl retorna una lista:

```
$test = "el dijo ella dijo";
@calce = $test =~ /\s*(\w+)\s*(\w+)\s*(\w+)\s*(\w+)/;
# @calce es ("el", "dijo", "ella", "dijo");
```

- También se pueden utilizar los paréntesis para dar opciones:

```
/(boy|girl)/ Calza si la variable contiene "boy" o "girl"
```

Expresiones Regulares

- También se utilizan los paréntesis cuadrados, para indicar clases de caracteres
 - “-” indicamos un rango
 - [a-z] calza cualquier letra minúscula
 - [aeiou] calza cualquier vocal minúscula
 - Se pueden “negar” clases utilizando “^”
- [^a-zA-Z] Calza cualquier cosa que no sea una letra
- Para calzar caracteres especiales, se debe anteponer un “\”
 - Caracteres especiales: ?, ., (, /, etcétera.

Expresiones Regulares

- Ejemplos:

```
/[0123456789]/ # cualquier dígito
/[0-9]/ # idem
/[0-9-]/ # calza 0-9 o “-”
/[a-z0-9]/ # cualquier minúscula o dígito
/[a-zA-Z0-9_]/ # cualquier letra, dígito o subr.
/[^aeiouAEIOU]/ # cualquier cosa que no sea vocal
```

Expresiones Regulares

- Clases predefinidas:
 - \d para [0-9]
 - \w para [a-zA-Z0-9_]
 - \s para [\t\nf]
- Se puede utilizar “{x,y}” para indicar entre “x” e “y” ocurrencias:

```
/r{3,5}/ # mínimo 3 y máximo 5 “r”
/r{4}/ # 4 “r”
/r{4,}/ # 4 o más “r”
/r{0,5}/ # 5 o menos “r”
```

Expresiones Regulares

- Sustitución:
 - Luego de “calzar” un patrón puede ser necesario sustituirlo por otro
 - Para realizar una sustitución se utiliza la sintaxis:

```
s/exp/repl/
```

- Ejemplo:

```
$test = "el dijo ella dijo";
$test =~ s/dijo/hablo/; # $test= "el hablo
                        #      ella dijo"
```

Expresiones Regulares

- Se buscó el primer calce y se reemplazo por el nuevo patrón
- Para realizar el reemplazo en todo el string se utiliza la opción “g”:

```
$test = "el dijo ella dijo";
$test =~ s/dijo/hablo/g; # $test= "el hablo
                        #      ella hablo"

$test = "123 456 7 890";
@a = $test =~ /\b(\d+)\b/g;
# @a = (123, 456, 7, 890)
```

Expresiones Regulares

- Utilizando las variables \$1 y \$2:

```
$test = "cambie rojo con verde.";
$test =~ s/cambie (.) con (.*)\./cambie $2 con $1./;
print $test; # "cambie verde con rojo"
```

- Otras variables especiales:
 - \$&: parte del string que cumple con el calce
 - \$`: parte del string anterior al calce
 - \$': parte del string que está después del calce

Expresiones Regulares

- Ejemplo:

```
$_ = "este es un ejemplo de texto";
/ej.*plo/; # calza "ejemplo"
          # `$ es "este es un "
          # $& es "ejemplo"
          # `$ es " de texto"
```

- Más de expresiones regulares: 'perlrope'

Estructuras de Control

- Evaluación:

```
if (){
}
else{
}
}
```

- Los paréntesis “{” y “}” **deben** utilizarse, aunque sólo encierren 1 línea

Estructuras de Control

- Ciclos:

```
for ($i=0; $i<10; $i++){print "contador: $i\n";}

for $i (0..9){print "contador: $i\n"; }

foreach (@a){ print $_ . "\n"; }
foreach $k (keys %h){ print "$k -> $h{$k}\n"; }

#filtro de comentarios y líneas en blanco
while(<>){next if /^#/; next unless /\S/; print}
```

Funciones

- Las funciones y subrutinas no son muy diferentes, ambas retornan un valor si se desea:

```
sub saluda{
    print "Hola\n";
}
```

- Se puede usar: `saluda()` ;
- Los parámetros son recibidos en el arreglo `@_`

Funciones

- Función con parámetros:

```
sub saluda{
    ($uno, $dos, $tres) = @_;
    warn "No hay parámetros!"
    unless $uno and $dos and $tres;
    print "Hola $uno, $dos, $tres\n";
}

saluda("Juan", "José", "Andrés");
```

Funciones

- La función “warn” imprime un mensaje de error pero permite que el programa continúe
- Existe la función “die” que imprime el mensaje pero termina inmediatamente la ejecución del programa
- Como los parámetros están en @_, se pueden obtener utilizando “shift”

```
sub saluda{
    $nombre = shift;
    print "Hola $nombre\n";
}
```

Funciones

- Se puede retornar dependiendo del contexto de la función:

```
sub funcion{
    # operaciones....
    # retornar si se llama en contexto "void"
    return unless defined wantarray;
    if (wantarray){ #se espera un arreglo
        return @resultado;
    }# se espera un escalar
    return $valor
}
```

Funciones

- Las variables son consideradas “globales”

```
$var = "externa";
sub a{ $var = "modificada en a";}
a();
print $var; # "modificada en a";
$var = "externa";
sub b{ my $var = "modificada en b";
    print "En b: $var";
}
b(); # "En b: modificada en b"
print $var; # "externa"
```

Funciones

- Utilizando “my” se indica que la variable existirá en el bloque actual
 - Forma de tener variables privadas
 - Se puede utilizar con una lista de variables
- ```
my ($var, @lista, %fonos);
```
- Este tipo de variables no es “vista” desde otras rutinas que se llaman dentro del bloque
  - Para tener variables que sean alcanzables para todas las funciones llamadas desde un bloque se utiliza “local”

## Funciones

```
$var = "original";
imprime();
cambia();
imprime();

sub cambia{
 local $var = "temporal";
 imprime();
}

sub imprime{
 print "Valor actual: $var\n";
}
```

## Archivos

- La entrada y salida con archivos se realiza por medio de "filehandles"
- Los filehandles son un tipo de variable especial, cuando un programa comienza su ejecución existen:
  - STDIN: entrada estándar
  - STDOUT: salida estándar
  - STDERR: salida de errores
- Se puede escribir en ellos utilizando "print":

```
print STDERR "Ocurrió un error!\n";
```

## Archivos

- Los archivos se acceden con "open" (tal como en C), dando como parámetro el filehandle y el nombre de archivo
- Se puede utilizar la sintaxis estilo shell para acceder los archivos:
  - "filename" o "<filename" para leer
  - ">filename" para truncar o crear y escribir
  - ">>filename" para agregar al final

```
open(LOG, ">error.log")
 or die "No puedo escribir en error.log: $!\n";
```

## Archivos

- Perl se encarga de cerrar todos los archivos abiertos con "open" al finalizar la ejecución
  - Es recomendable utilizar "close(filehandle)" para cerrar los archivos abiertos
- Se recomienda utilizar mayúsculas para los filehandles:
  - Se podría tener: \$var, @var, %var, sub var, filehandle var
- Al reabrir un archivo, automáticamente se cierra el archivo abierto anteriormente
- Se puede averiguar información del archivo utilizando los "File test"

## Archivos

- Para saber si un archivo existe:

```
if (-e $filename){
 print "$filename existe!\n";
} else {
 print "No existe $filename\n";
}
```

- Con “-r” se puede averiguar si el archivo existe y se puede leer
- Con “-w” se puede saber si el archivo se puede escribir
- Para revisar todo el listado de “file test”:  
perldoc perlfunc

## Archivos

- Los file test entregan bastante información de los archivos, pero no toda:
  - ¿Como saber la cantidad de enlaces que tiene un archivo?
  - Para obtener toda la información relacionada a un archivo se utiliza stat:
 

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
 $atime,$mtime,$ctime,$blksize,$blocks)
 = stat($filename);
```
  - Más información: perldoc -f stat

## Archivos

- Se puede utilizar:

```
($uid, $gid) = (stat("tarea4.pl"))[4,5];
```

- “stat” no hace diferencia entre un nombre de archivo y un enlace
  - Si recibe un enlace como argumento, entregará la información de lo que apunta el enlace
- “lstat” opera igual que stat para archivos, pero si recibe un enlace entregará la información del enlace