programa de BECAS
DGTIC
de FORMACIÓN en TIC

Seguridad Informática

Programación con Perl

Cervantes Varela Juan Manuel
Vallejo Fernández Rafael Alejandro

Temario

- Introducción
- Conceptos
- Funciones
- Subrutinas
- Manejo de archivos
- Expresiones Regulares
- Estructuras de datos y referencias
- CPAN

Evaluación

- Ejercicios 10%
- Tareas/Prácticas 20%
- Proyecto 30%
- Examen teórico 10%
- Examen práctico 30%

Entrega

- **Formato de nombre de archivos:**

  [t|e|p]#_letraPrimerNombre_apellido.ext

```
Crear repo de GitHub:
Perl
|
|_Ejercicios:
|    |_ e1_rvallejo.pl
|    |_ e2_rvallejo.pl
|
|_Tareas
|    |_ t1_rvallejo.pl
|    |_ t2_rvallejo.pdf
|    |_ t3_rvallejo.pl
|
|_Practicas
     |_Practica1
     |    |
     |    |_p1_rvallejo.pl
     |
     |_ Practica2
          |
          |_p2_rvallejo.pl
```

Invitar usuario: **cursoPerl14g**

Introducción

Concepto de Perl

- Lenguaje de programación creado por Larry Wall
- Acrónimo de Practical Extraction and Report Language
- Más reconocido por ser un lenguaje práctico y rápido que estructurado o elegante
- Lema: "Hay más de una forma de hacerlo"
- Su potencial está en el procesamiento de texto (con archivos, cadenas y expresiones regulares)
- El estilo rápido e informal de Perl lo hace adecuado para todo tipo de programas pequeños

Historia (breve) de Perl

- A mediados de los 80's, Larry Wall trabajaba como sys-admin y se dio cuenta que tenía que ejecutar ciertas tareas una y otra vez.
- No le gustó ninguno de los lenguajes de programación que existían en aquel entonces y por eso inventó Perl.
- La versión 1 se lanzó alrededor de 1987 y han ocurrido muy pocos cambios desde esa versión a la más actual (5.32.0).
- Tiene características del lenguaje C, de bash, de awk, sed y en menor cantidad de otros lenguajes de programación

Ejecutar un programa en Perl

- Forma 1: Script ejecutable

1.- Dar permisos de ejecución al script (chmod 755 script.pl)

2.- Agregar en el script como primera línea un shebang, el cual contendría la dirección del ejecutable del intérprete.

```
1  #!/usr/bin/perl
2
3  # ...
4  # Código
5  # ...
```

3.- Se ejecuta como cualquier ejecutable (./script.pl)

- Forma 2: Mediante intérprete de Perl

```
1  # ...
2  # Código
3  # ...
```

1.- Se ejecuta en el intérprete bash de la siguiente forma:

perl script.pl

Estructura básica de un programa

De forma general, un script de Perl lleva la siguiente estructura.

```
1  #!/usr/bin/perl
2  use warnings;  # adevertencias para errores inesperados
3  use strict;  # obliga a que la declaración de variables sea con "my"
4
5  # declaracion e incialización de variables
6
7  # programa principal "main"
8
9  # subrutinas
```

Variables

- Los caracteres permitidos para los nombres de las variables son los alfanumericos y el guión bajo, y no debe empezar con números.
- Las letras mayúculas y minúsculas son diferenciadas (case sensitive).
- Van precedidas de un símbolo ($, @ o %) para indicar el tipo de dato (escalar, lista o lista asociada respectivamente).

Variables especiales

- Existen variables especiales que tienen un funcionamiento predefinido, y que son utilizadas y almacenan datos de forma automática al ejecutar ciertas funciones u operaciones, como por ejemplo:

```
1  $_ # Variable por defecto de operadores y funciones
2  @_ o @ARGV # Arreglo que almacena los argumentos recibidos
3  $#ARGV # Obtiene el número de argumentos recibidos - 1
4          # o el índice del último elemento
5  $! # Variable que almacena errores de ejecución
6  $| # Si no es cero, obliga a vaciar el búfer después de cada operación de escritura
```

- https://perldoc.perl.org/perlvar.html

Tipos de datos

El lenguaje Perl posee tres tipos de representaciones de datos:

- **Escalares ($):** representa el tipo básico en Perl. Permite representar enteros, reales y cadenas de caracteres..
- **Arrays o listas (@):** lista de datos de tipo escalar. Cada elemento de la lista es una variable escalar.
- **Hashes o listas asociadas (%):** lista asociativa que está indexada por cadenas (o valores escalares) en lugar de números, formando parejas del tipo *(llave, valor)*.

Tipos de datos - Escalares

Dentro de los tipos de datos escalares, se aceptan valores numéricos (enteros, reales, octales y decimales), ademas de cadenas de caracteres y booleanos, aunque estos últimos de forma implícita.

- Numéricos

```
1  $numerico = 4.57;        # real
2  $numerico = 54;          # entero
3  $numerico = 0567;        # octal
4  $numerico = 0xe3;        # hexadecimal
```

Tipos de datos – Escalares

- Cadenas

Dentro de perl existe la "interpolación de variables" lo cual es, dentro de una cadena de caracteres agregar el valor de otra cadena de caracteres o variable.

```
1  $cadena_hola = "Hola ";
2  $cadena_mundo = 'mundo!';
3
4  print "Hola $cadena_mundo";          # Interpolaciónd de cadenas
5  print 'Hola $cadena mundo';
6  print $cadena_hola . $cadena_mundo;   # Concatenacion de cadenas
```

Tipos de datos – Escalares

- Cadenas

Otra sintaxis para definir cadenas con multiples saltos de línea, comillas y/o apóstrofes, es la siguiente:

```
1  # variable = <<ETIQUETA;
2  #     ...
3  # ETIQUETA
4
5  $cadena = <<FINAL;
6  hola,
7  mundo,
8  !,
9  FINAL
10 print $cadena;
```

Tipos de datos - Escalares

- Booleanos

Como tal, el tipo booleano no existe, pero se puede usar de forma implícita en variables escalares, donde una cadena vacía o el número 0 representa falso, y cualquier otro número o cadena, representa verdadero.

```
1   $bool = 1;       # verdadero
2   $bool = "a";     # verdadero
3   $bool = 0;       # falso
4   $bool = ""       # falso
```

Tipos de datos - Arrays

Admite valores escalares (ya sean de un solo tipo o diferentes) y existen diferentes formas de declararlos:

- Especificando cada valor de forma individual separados por comas.
- Especificando los valores de forma individual separados por espacios y utilizando qw (quote word), la cual hace un split de la cadena entre paréntesis y cada uno lo guarda como cadenas acotadas por comilla simple.
- Definiendo un rango de valores a almacenar usando ..

```
1  @arrays = ("hola", 23, "adios", 31.234);
2  @mezcla = qw(hola 1 adios 3.75);
3  @alfabeto = (a..z);
```

Tipos de datos - Arrays

Se puede acceder a los valores de los arreglos de diferentes formas:

- Especificando el indice de forma explicita.
- Generando y retornando otro arreglo especificando un rango de indices.
- Especificando varios indices separados por coma.
- Asignando los valores almacenados a variables infividuales de forma directa (en caso de especificar más variables de los valores almacenados, las excedentes quedarían vacías).

* NOTA: En el caso de querer obtener un único elemento de la lista, debe especificarse con el nombre de la variable tipo array pero con el simbolo $.

Tipos de datos - Arrays

```
1  print $alfabeto[2];          # c
2  @num1 = @alfabeto[1..3];     # @num1 = ("b", "c", "d")
3  @str = @alfabeto[10, 15];    # @str = ("k", "p")
4
5  @alfabeto = (a...c)
6  ($uno, $dos) = @alfabeto;
7  # $uno = a, $dos = b
8  ($uno, $dos, $tres, $cuatro) = @alfabeto;
9  # $uno = a, $dos = b, $tres = c, $cuatro = ""
```

Tipos de datos - Arrays

Se pueden combinar varios arrays para generar otro, en estos casos perl detecta que son del mismo tipo y los "aplana" insertando uno a uno todos sus elementos en la posición indicada del array que los contendrá.

```
1  @a = (1,2,3);
2  @b = (5,6,7);
3  @c = (@a,4,@b,8);
4  # @c = (1,2,3,4,5,6,7)
```

Tipos de datos - Arrays

Existen funciones para la manipulación de arreglos como lo son

- push(@arr, elem): agrega un elemento al final del arreglo.
- pop(@arr): extrae el último elemento del arreglo.
- shift(@arr): extrae el primer elemento del arreglo.
- unshift (@arr,elem): agrega un elemento al principio del arreglo.

Funciones de arrays

```perl
@array = qw(uno dos tres cuatro cinco);
print "@array\n";           # uno dos tres cuatro cinco
push(@array, "seis");
print "@array\n";           # uno dos tres cuatro cinco seis
$aux = pop(@array);
print "@array\n";           # uno dos tres cratro
$aux = shift(@array);
print "@array\n";           # dos tres cuatro
unshift(@array, "cero");
print "@array\n";           # cero dos tres cuatro
```

Tipos de datos - Hashes

Las listas asociadas o hashes se pueden declarar de dos formas:

- Separando los valores con comas, almacenandose dentro del hash en pares, donde el primero es la llave y el segundo el valor.
- Especificando de forma explicita la llave seguido de "=>" y el valor, separando cada par por comas.

```
1  %usuarios = ("root",1000,"luis",5230,"pedro",6000);
2
3  %usuarios =(
4      "root"=>1000,
5      "luis"=>5230,
6      "pedro"=>6000
7  );
```

Tipos de datos - Hashes

Para acceder, modificar o añadir nuevos elementos, se debe especificar la variable y entre llaves la clave (llave) deseada.

* NOTA: Para obtener o modificar el valor de una llave, debe especificarse con el nombre de la variable tipo hash pero con el simbolo $.

```
1  $id = $usuarios{"root"};        # $id = 1000
2  $usuarios{"pedro"} = 500;
3  print $usuarios{"pedro"}        # 500
```

Funciones con hashes

Las funciones creadas predeterminadamente para los hashes son las siguientes

- keys(%hash): devuelve un arreglo con todas las llaves de un hash
- values(%hash): devuelve un arreglo con todos los valores de un hash.
- delete $hash{‘val’}:Elimina el elemento del hash de la llave dada.

Funciones con hashes

```
1  %hash = ("uno", 1, "dos", 2, 3, "tres");
2  @llaves = keys(%hash);
3  print "@llaves\n";           # uno dos 3
4  @valores = values(%hash);
5  print "@valores\n";          # 1 2 tres
6  delete $hash{3};            # ("uno", 1, "dos", 2)
```

Referencias

- Son equivalentes a los punteros en C
- Referencian a datos contenidos en otra entidad.
- Son tratados como tipo de dato escalar.
- La definición de referencia se usa mediante el operador de referencia backslash (\)
- Tiene un mecanismo de *"papelera de reciclaje"* que consiste en un registro que posee las diversas referencias a cada elemento en memoria y las destruye cuando descubre que nadie hace referencia a él.
- Se pueden referenciar variables (creando un nuevo punto de acceso a su valor) o a valores (creando un objeto anonimo al que solo se puede acceder por la referencia).
- Para referenciar de forma anonima se deben delimitar las cadenas entre "" e iniciando con \, las listas entre [] y los hashes entre {}

Referencias

```perl
$escalar = 5;
@arreglo = qw(a b c);
%hash = ("uno", 1, 2, "dos")
$rescalar = \$escalar;  # Referencia a variable escalar
$rarreglo = \@arreglo;  # Referencia a variable array
$rhash = \%hash;        # Referencia a variable hash
```

```perl
$rescalar = \"hola";              # Referencia anonima a variable escalar
$rarray = [1,2,3];                # Referencia anonima a variable array
$rlista = {"llave1" => "dato1"};  # Referencia anonima a variable hash
```

```perl
print $rescalar, "\n";  # SCALAR(0x5618bf01a738)
print $rarreglo, "\n";  # ARRAY(0x5618bf00d9a0)
print $rhash, "\n";     # HASH(0x5618bf01a5d0)
```

Referencias

Para acceder o modificar los valores de una referencia se puede hacer mediante dos formas:

- Añadiendo el identificador a la variable que aloja la referencia (similar a un cast, quedando el $ diplicado)
- Mediante el operador "->"

```
1  $rarray = [1,2,3,4];          # Creacion de array anonimo
2  print $$rarray[2], "\n";      # 3
3  $rarray->[2] = "tres";         # o $$rarray[2] = "tres";
4  print $rarray->[2], "\n";     # tres
5
6  @$rarray=();                 # Limpia el arreglo anonimo
```

Referencias

```
1  $val = hola;
2  print $val, "\n";    # hola
3  $rval = \$val;
4  $$rval = "mundo";
5  print $val, "\n";     # mundo
```

```
1  @array = (1, 2, 3, 4, 5);
2  $rarray = \@array;
3  print @array, "\n";      # 12345
4  $rarray->[2] = 3000;
5  print @array, "\n";      # 12300045
6  $$rarray[3] = 6;
7  print @array, "\n";      # 12300065
```

```
1  %hash = ("uno", 1, "dos", 2, "tres", 3);
2  print %hash, "\n";           # uno1do2tres3
3  $rhash = \%hash;
4  $rhash->{"dos"} = 2000;
5  print %hash, "\n";           # uno1dos2000tres3
6  $$rhash{"cuatro"} = 4;
7  print %hash, "\n";           # uno1dos2000tres3cuatro4
```

Operadores aritméticos

Significado	Operador
Suma	+
Resta	-
Producto	*
División	/
Residuo (módulo)	%
Potencia	**

```
1  $a = 7
2  $b = 5
3  print $a + $b;    # 12
4  print $a - $b;    # 2
5  print $a * $b;    # 35
6  print $a / $b;    # 1.4
7  print $a % $b;    # 2
8  print $a ** $b;   # 16807
```

Operadores de asignación

Significado	Operador
Asignación	=
Incremento en 1	++
Decremento en 1	--
Asignación combinado con operadores aritmeticos	+=, -=, *=, /=, %=, **=
Asignación combinado con operadores lógicos (a nivel de bits, excepto la negación)	&=, |=, ^=, !=

```
1   $a = 5;
2   print $a++, " ", ++$a; # 5 7
3   print $a--, " ", --$a; # 7 5
4   $a += 4;          # $a = $a + 4;
5   $a &= 3;          # $a = $a & 3;
```

Operadores de comparación

Significado	Operador numérico	Operador de cadena
Igual a	==	eq
No igual a	!=	ne
Menor que	<	lt
Mayor que	>	gt
Menor o Igual	<=	le
Mayor o Igual	>=	ge

```
1  $a = 7;
2  $b = 5;
3  print $a == $b; #
4  print $a != $b; # 1
5  print $a < $b;  #
6  print $a > $b;  # 1
7  print $a <= $b; #
8  print $a >= $b; # 1
```

Operadores lógicos

Significado	Operador numérico	Operador de cadena
Conjunción (AND)	&&	and
Disyunción (OR)	||	or
Negación (NOT)	!	not

```
1  $a = 5;
2  $b = 0;
3
4  print $a && $b;    # 0
5  print $a || $b;    # 5
6  print !$a;         #
```

Operadores lógicos a nivel de bits

Significado	Operador numérico	Operador de cadena
Conjunción (AND)	&	
Disyunción (OR)	|	
Negación (NOT)	~	
Disyunción exclusiva (XOR)	^	xor

```
1  $a = 5;
2  $b = 0;
3
4  print $a & 3;      # 1
5  print $a | $b;     # 5
6  print ~$a;         # 18446744073709551610
7  print $a ^ 3;      # 6
```

Otros operadores

- Cmp: Compara dos cadenas (carácter a carácter), devolviendo –1 si cadena de la izquierda es menor que la de la derecha, 0 si es igual y 1 si es mayor.

```
1  print 'abc' cmp 'abc', "\n";  # 0
2  print 'abc' cmp 'cba', "\n";  # -1
3  print 'cba' cmp 'abc', "\n";  # 1
```

- <=>: Compara dos numeros, devolviendo –1 si el número de la izquierda es menor, 0 si es igual al de la derecha y 1 si es mayor.

```
1  print 10 <=> 100, "\n";  # -1
2  print 10 <=> 10, "\n";   # 0
3  print 10 <=> 9, "\n";    # 1
```

Otros operadores

- Selección condicional (ternario): Evalúa una condición, eligiendo un valor si se cumple u otro si no.

```
# condicion ? valor_si : valor_no

$res = (100<3 ? "Si" : "No"); # $res = No
```

- , : Evalúa varias expresiones donde la sintaxis solo permite una, el valor resultante es la última (de izquierda a derecha)

```
$i = (10+5, 30*8, 4/2);
print $i;                    # 2
```

Estructuras de control

Existen diferentes estructuras de control que nos permiten modificar o cambiar el flujo de ejecución de nuestros programa. Existen 3 categorías:

- Condicionales

Ejecutan un bloque de código si se cumple una condición.

- Cíclicas

Ejecutan repetidamente un bloque de código si se cumple (o hasta que se cumpla) una condición.

- Comandos de control de flujo

Permiten cambiar el flujo de ejecución de forma directa.

Estructuras de control - condicionales

Dentro de esta categoría se encuentran:

- If: Ejecuta el bloque de código si se cumple la expresión dada, se puede implementar por sí sola, con else y/o con elsif.
- Unless: (a menos que, si no existe..., si no es...) Ejecuta el bloque de código si no se cumple la expresión dada. De igual forma puede implementarse por sí sola o con else y elsif, aunque este último conserva el mismo funcionamiento que con If.

Estructuras de control – condicionales

```
1  if (condicion) {
2      # Codigo a ejecutar si se cumple la condicion
3  }
```

If

```
1  if (condicion) {
2      # Codigo a ejecutar si se cumple la condicion
3  } else {
4      # Codigo a ejecutar si no se cumple la condicion
5  }
```

If - else

```
1  if (condicion_1) {
2      # Codigo a ejecutar si cumple la condicion_1
3  } elsif (condicion_n) {
4      # Codigo a ejecutar si no se cumple(n) a(s) condicion(es) anterior(es)
5  } else {
6      # Codigo a evaluar si no cumple ninguna condicion
7  }
```

If – elsilf - else

Estructuras de control – condicionales

```
unless(condition) {
   # Codigo a ejecutar si no se cumple la condicion
}
```

Unless

Unless - else

```
unless(condition) {
   # Codigo a ejecutar si no se cumple la condicion
} else {
   # Codigo a ejecutar si se cumple la condicion
}
```

Unless – elsif - else

```
unless(condition_1) {
   # Codigo a ejecutar si no se cumple la condicion_1
} elsif (condition_n) {
   # Codigo a ejecutar si se cumple la condicion_n
} else {
   # Codigo a ejecutar si se cumple la condicion_1 y no se cumple(n) la(s) condicion_n
}
```

Estructuras de control - ciclicas

Dentro de esta categoría se encuentran:

- while: Ejecuta el bloque de código de forma ciclica mientras se cumpla la expresion dada. En cada iteración evalua la condición y a continuación ejecuta el codigo (si se cumple) o se sale del ciclo (si no se cumple).
- Until: Ejecuta el bloque de código de forma ciclica hasta que se cumpla la expresion dada. En cada iteración evalua la condición y a continuación ejecuta el codigo (si no se cumple) o se sale del ciclo (si se cumple).
- Continue: Estructura de control dependiente a otras, el cual ejecuta el bloque de instrucciones al terminar la iteración actual (similar a la tercera parte del for (actualización)).

Estructuras de control - ciclicas

- Do...while: Ejecuta el bloque de código de forma cíclica mientras se cumpla la expresión dada. En cada iteración ejecuta el código y a continuación se evalúa la condición y continúa las iteraciones (si se cumple) o se sale del ciclo (si no se cumple). Tambien se puede usar con unless.
- For: Ejecuta el bloque de código de forma ciclica mientras se cumpla la expresion dada. Se pueden proporcionar 3 variables:
  - Inicialización: Se asignan valores a las variables a usar.
  - Condición: Condición a evaluar en cada iteración.
  - Actualización: Se actualizan los valores de las variables en cada iteración para que haya un progreso.

Estructuras de control - ciclicas

- Foreach: Itera sobre los elementos de un arreglo, asignando cada uno a una variable especificada (o $_ si no se especifica ninguna) en cada iteración.

```
while(condicion) {
    # Bloque de instrucciones si se cumple la condicion
}
```

```
until (condicion) {
    # Bloque de instrucciones si no se cumple la condicion
}
```

Estructuras de control - ciclicas

```
1  while(condicion) {
2      # Bloque de instrucciones si se cumple la condicion
3  } continue {
4      # Bloque a ejecutar antes de la siguiente iteración
5  }
```

```
1  do {
2      # Bloque de instrucciones a ejecutar si se cumple la condición
3  } while (condicion);
```

Estructuras de control - ciclicas

```
for(inicializacion ; condicion ; actualizacion) {
    # Bloque de instrucciones a ejecutar si se cumple la condicion
}
```

```
foreach $variable (@lista) {
    # Bloque de instrucciones
}
```

Estructuras de control – control de flujo

Dentro de esta categoría se encuentran:

- Goto: Salta a una etiqueta especificada en otra parte del código, ignorando el código intermedio. No es recomendable su uso y es considerado una mala práctica debido a que puede generar código espaguetti.
- Next: Su uso es dentro de estructuras de control cíclicas, y lo que hace es continuar con la siguiente iteración del ciclo ignorando al código posterior de esta instrucción.
- Last: Su uso es dentro de estructuras de control cíclicas, y lo que hace es terminar y salirse del ciclo y continuar con las siguientes instrucciones.

Estructuras de control – control de fluio

```
1  print "Instruccion 1", "\n";
2  print "Instruccion 2", "\n";
3  goto salida
4  print "Instruccion 3", "\n";
5  salida: print "Instrucciones ejecutadas";
```

```
1  for ($i=0 ; $i<100 ; $i++) {
2      if ($i<50) {
3          next;
4      }
5      print $i;
6      if ($i>60) {
7          last;
8      }
9  }
```