Guía Introductoria al Lenguaje Perl

versión 2.1

Fernando Bordignon, Gabriel Tolosa y Fernando Lorge

{bordi, tolosoft, florge}@mail.unlu.edu.ar

Laboratorio de Redes de Computadoras
División Estadística y Sistemas
Universidad Nacional de Luján

La presente guía es una versión “quick and dirty” -tal como dicen los anglosajones- con el objetivo de lograr un primer acercamiento al lenguaje. Se sugiere complementar el estudio con bibliografía adicional, como la recomendada.

Charles Babbage dijo *"se cometen muchos menos errores usando datos incorrectos, que no empleando dato alguno*". Es por esto que se pide perdón por los posibles errores que se hayan deslizado, y se solicita al lector que contribuya a mejorar este material de estudio, a partir de las correcciones y críticas que pueda aportar.

Fuentes de bibliografía adicional sobre el lenguaje Perl:

- Página oficial del lenguaje Perl. http://www.perl.com
- Curso de Perl (en inglés) http://fpg.uwaterloo.ca/perl/start.html
- Perl Mongers. http://www.perl.org/
- Editorial O´Reilly, página del lenguaje Perl. http://perl.oreilly.com/
- PerlDoc. http://www.perldoc.com/
- The Perl Journal. http://www.tpj.com/
- Perl Chile. http://www.perl.cl/
- En su <X>unix escriba: man perl

Lugares de descarga del lenguaje:

- Página oficial del lenguaje Perl. http://www.perl.com
- CPAN - Módulos Perl. http://www.cpan.org/
- Activestate. Perl para Windows. http://www.activestate.com/

Filosofía de Perl:

"Hay más de una forma de hacerlo" *según* ***Larry Wall,*** *su autor.*

Introducción al Lenguaje

"Si no noto la resistencia del papel frente a la pluma, me siento incapacitado" Jean Daniel

Perl significa *Lenguaje práctico de extracción y reporte (*Practical Extraction and Report Language) . Su creador fue Larry Wall, y su objetivo era simplificar las tareas habituales a realizar en el sistema operativo Unix. Hoy es un lenguaje de propósito general, de alta portabilidad y es la herramienta principal que utilizan los webmasters para implementar programación a través de la interfase CGI.

Perl es un lenguaje interpretado, aunque transparentemente compile los programas antes de ejecutarlos. Es por ello que los ortodoxos de la comunidad hablan de guiones (scripts) y no de programas. Existe una numerosa cantidad de programadores Perl distribuidos a lo largo del mundo, ellos día a día producen nuevos módulos de librería que extienden asombrosamente las capacidades del lenguaje. En Perl, practicamente, es posible programar casi cualquier problema; maneja: comunicaciones entre computadoras, bajo protocolos TCP/IP; implementa hilos de ejecución (threads); implementa -opcionalmente- el estilo de programación orientado a objetos, posee APIs con todos los protocolos populares de aplicación en Internet. Aunque su punto debil es la perfomance en la ejecución de guiones -no alcanza las velocidades ofrecidas por lenguajes como el C-, Perl es un excelente lenguaje destinado al aprendizaje y el desarrollo de prototipos de modelos de software.

PERL es gratuito, hay completa libertad de copiar el código fuente, compilarlo, imprimirlo, compartirlo, etc, pero todo esto es bajo la licencia GNU, que en términos generales ordena que cualquier programa que se desarrolle se debe liberar, o sea si Ud. puede utilizar el código hecho por otros; otros pueden usar el código hecho por Ud.

Como se mencionó anteriormente, la portabilidad es una características de Perl. Los guiones se pueden ejecutar sobre distintos sistemas operativos como ser: Linux, System de Macintosh, Microsoft Windows, Solaris de Sun, BSD, etc. En definitiva, un programa puede correr en cualquier sistema operativo sin tener que modificar el código fuente; solamente es necesario poseer el interprete de Perl para cada sistema y librerías adicionales que use.

Con Perl se pueden: construir pequeños programas destinados a ser usados como filtros para obtener información en base a la lectura de archivos, realizar búsquedas, etc. Es uno de los lenguajes más utilizados en la programación de la interfase CGI, que sirve para el intercambio de información entre aplicaciones externas y el espacio web. Por ejemplo programas de búsqueda, y procesamiento de formularios, entre otros.

El siguiente es el clásico ejemplo de primer programa Perl

```
#!/usr/local/bin/perl

print "Hola Mundo!!!\n";
```

En la primer linea se le indica al sistema operativo el camino del interprete Perl, para que lo pueda cargar y ejecutar, y le pase como parámetro el guión que se halla más abajo. Por norma, todo guión lleva la extensión pl y se lo escribe utilizando un editor ASCII normal. Inicialmente, para ejecutar un programa siga la siguiente sintaxis:

perl <nombre_del_programa.pl>

Para un mejor análisis de un programa, puede redireccionarse su salida a un archivo utilizando en la orden de ejecución el símbolo de redirección ">" y a continuación un nombre de archivo.

Tipos de Datos, Variables y Arreglos

"La comunicación se compone de quien habla, quien escucha y lo que se dice". Aristóteles

Perl permite representar los siguientes tipos de datos básicos: reales, enteros, cadenas de caracteres y booleanos.

Los tipos numéricos (reales y enteros).

Los valores numéricos se presentan en forma de valores reales codificados en doble precisión. Este formato interno se utiliza para todas las operaciones aritméticas.

```
$z = 0.127;          # real
$x = 3.22e-14;       # real
$c = 1567;           # entero
$d = -122;           # entero
```

Los valores enteros no pueden empezar por cero porque esto permite especificar un entero mediante su codificación octal o hexadecimal. El código octal comienza con cero 0; el código hexadecimal comienza con 0x.

```
$x = 0377;    # Representación octal, equivale a 255 decimal
$y = 0xff;    # Representación hexadecimal, equivale a 255
```

Las cadenas de caracteres.

Las cadenas de caracteres se representan por medio de un sucesión de caracteres delimitada por comillas ó apóstrofes. Cuando una cadena se delimitada por comillas, toda variable referenciada en el interior de la cadena se evalúa y se reemplaza por su valor. Por ejemplo:

```
$cadena = "Brothers";
$almace = "Blues $cadena";
```

Por el contrario, las cadenas de caracteres delimitadas por apóstrofes se dejan intactas.

Existe una sintaxis especial que permite delimitar una cadena de caracteres cuando contiene varias líneas y/o comillas o apóstrofes. El fin de la cadena se determina por la nueva línea que contiene únicamente el identificador. Éste no debe ir precedido por un espacio ni marca de tabulación. Por ejemplo:

```
$cadena = <<SALUDO;
hola,
buenos días,
adios,
SALUDO
```

El tipo booleano.

El tipo booleano existe de modo implícito, es decir, un número es falso si es igual a cero y verdadero en cualquier otro caso. Como el cero está asociado a la cadena vacía *("")*, ésta también equivale al valor falso.

Representaciones de datos.

Perl maneja datos escalares, siendo todos aquellos que contienen un valor; no importa si es real, entero, caracter o cadena; el lenguaje los reconoce automáticamente.El lenguaje posee tres tipos de representaciones de datos

Variables escalares.
Arreglos indexados escalares (listas).
Arreglos asociativos escalares (hash).

Las representaciones de datos son determinadas por la anteposición de un caracter especial.

$ Escalar.
@ Arreglo Indexado.
% Arreglo Asociativo.
& Función ó Procedimiento

Todas las representaciones tienen su propio nombre, así como las etiquetas, funciones, archivos y manejadores de directorios.

Variables escalares.

Se utiliza el caracter $ para indicar un valor escalar. La asociación ó bind de una variable con valor se hace con el caracter =.

```
$foo = 55.23;
$foo = 'esta es una cadena';
$foo = "era $foo antes";          # variable interpolada.
$salida= `cd`;                    # Se invoca al sistema operativo y el resultado de la salida
                                  # se almacena en el escalar de la izquierda.
($a, $b) = ($b, $a)               # intercambio (swap)
```

Ejemplo:

Sumar dos números y luego aplicar el operador de post-incremento

```
#!/usr/local/bin/perl

$a = 10; $b = 5;
$suma = $a + $b;
print "Resultado:".$suma."\n";
$suma++;
print "Despues de incrementar: $suma \n";
```

Ejemplo:

Verificar el método de intercambio de variables

```
#!/usr/local/bin/perl
```

```
$a = "Mundo"; $b = "Hola";
($a, $b) = ($b, $a);
print "La famosa frase es $a $b!!! \n";
```

Las secuencias de escape se utilizan a los efectos de representar caracteres especiales ó modifcar un valor. Algunas son

\n	Nueva línea	
\t	Tabulador	Print "dato \t separado\n";
\0oo	oo valor octal de ASCII	
\xhh	hh valor hexadecimal de ASCII	
\\	Diagonal invertida	Print "la barra \\ sirve ...";
\"	Comillas dobles	Print "las comillas \" sirven ...";
\L	Minúsculas todas las letras hasta \E	Print "\LEN MINUSCULAS\E";
\U	Mayúsculas todas hasta \E	Print "\Uen mayusculas\E";
\E	Terminar \L o \U	

Ejemplo:

Ingresar dos cadenas y comparar si son iguales, idependientemente que sus caracteres esten en mayúsculas o minúsculas.

```
#!/usr/local/bin/perl

print "Ingrese 1er cadena:";
$cadenaa =<STDIN>;              #se pide ingreso por teclado
chop($cadenaa);                 #chomp elimina de cadenaa el enter
print "Ingrese 2da cadena:";
$cadenab=<STDIN>;
chop($cadenab);
$tmpa = "\U$cadenaa\E"; $tmpb = "\U$cadenab\E";

if ($tmpa eq $tmpb) {print "$tmpa y $tmpb son iguales \n";}
else {print "$tmpa y $tmpb NO son iguales \n";}
```

Arreglos

Arreglos Indexados

En PERL los arreglos son listas de datos sin importar su tipo, es decir que puede estar compuestos por enteros flotantes, cadenas, o incluso otros arreglos, cada elemento es considerado como una variable independiente. El caracter $ se utiliza para referirse a un elemento en particular y el caracter @ para todos. El primer elemento de un arreglo es el indicado con 0 (cero).

```
$arreglo[$i+2] = 3;          # Instancia el elemento $i+2 con el valor 3.
@arreglo = ( 1, 3, 5 );      # Inicializa el arreglo llamado arreglo
@arreglo = ( );              # Inicializa un arreglo vacío.
@foo = @bar;                 # copia el arreglo.
@foo = @bar[$i..$i+5];       # copia una parte del arreglo.
$numero = $#arreglo;         # el numero se almacena el valor de índice más alto
                             # definido. Es decir la cantidad de elementos del arreglo-1
```

Ejemplo:

```
#!/usr/local/bin/perl

@a = (95, 7, 'fff' );  # se instancia el arreglo a con 3 elementos
print $a[2];           # imprime el tercer elemento (fff)
print @a;              # imprime todo el arreglo 957fff
```

Ejemplo:

```
#!/usr/local/bin/perl

@arreglo1 = (10, 20, 30);
@arreglo2 = (100, 200);
@arreglo3 = (@arreglo1, @arreglo2, 8, "es una cadena");
$longitud_arreglo3 = @arreglo3;

# imprimir el arreglo
for ($n=0; $n<$longitud_arreglo3; $n++) {print $arreglo3[$n]."\n"};
```

Nota: Se a inicializado arreglo 3 con dos arreglos y dos variables escalares, por lo cual arreglo 3 paso a tener 7 elementos.

Ejemplo:

Se pueden definir valores de una lista por medio de la técnica de rangos:

```
#!/usr/local/bin/perl

@a = (2..7);     # $a se instancia con (2,3,4,5,6,7);
@b = ('a'..'e'); # $b instancia con ('a','b','c','d','e')

for ($n=0; $n<@a; $n++) {print $a[$n].” “;}; print “\n”;
$n=0; while($n<@b) {print $b[$n].” “;$n++;}; print “\n”;
```

Ejemplo:

La función join convierte un arreglo en un escalar y la función split hace lo inverso, convierte una variable escalar en una lista.

```
#!/usr/local/bin/perl

@a = ('a'..'e');
$a = join(":", @a);
print $a.”\n”; # se obtiene "a:b:c:d:e"

@lista = split(“:”, $a);
for ($n=0; $n<@lista; $n++) {print $lista[$n].” “}; print “\n”;
```

Ejemplo:

La función splice permite extraer un subarreglo y a la vez modificar el arreglo original

```
@a = ( 'a'..'e');
@b = splice( @a, 1, 2); # A partir del elemento 1 tome 2

for ($n=0; $n<@a; $n++) {print $a[$n].” “}; print “\n”;
     # @b queda con 2 elementos de @a: $a[1] y $a[2];
     # ( 'b', 'c')

for ($n=0; $n<@b; $n++) {print $b[$n].” “}; print “\n”;
     # @a queda sin esos 2 elementos:
     # ( 'a', 'd', 'e' );
```

```
# Se agrega un cuarto argumento (@b)y en @a se reemplaza el segundo elemento -
solamente (2, 1)- con el arreglo @b.

@a = ('a'..'e');
@b = (1..3);
splice( @a, 2, 1, @b);
for ($n=0; $n<@a; $n++) {print $a[$n].” “}; print “\n”;

     # Se imprime 'a', 'b', 1, 2, 3, 'd', 'e' dado que
     # el elemento de valor 'c' se reemplazo con (1, 2, 3)

# Es posible insertar sin eliminar elemento alguno

@a = ( 'a'..'e');
@b = ( 1..3);
splice( @a, 2, 0, @b);
for ($n=0; $n<@a; $n++) {print $a[$n].” “}; print “\n”;

     # Se imprime 'a', 'b', 1, 2, 3, c', 'd', 'e';
```

Arreglos n-dimensionales

En la siguiente instrucción se define un arreglo de 4 filas por 3 columnas -recordar que en Perl un arreglo comienza con el índice numérico 0-

```
@arreglo = ( [1, 2, 3], [4, -5, 6], [7, 8, 9], [10, 11, 12]);

print $arreglo[3][1]."\n"; # Imprime 11
```

Un arreglo no es más que una lista donde un elemento n puede ser a su vez otro arreglo, por lo tanto no es necesario que todas las filas tengan la misma longitud.

```
@m1 = ( 1 , "maria" );
@m2 = ( "pablo", "guillermo", "silvina" );
@m3 = ( "rosa", "agustin" , 3 );
@m  = ( [@m1], [@m2], [@m3] );

print $m[2][1]."\n";   # Imprime agustin
```

Para determinar la cantidad de filas y columnas se utiliza:

```
print $#arreglo."\n";         # Brinda la cantidad de filas-1 del arreglo
print $#{$arreglo[1]}."\n";   # Brinda la cantidad de elementos-1 de la fila 1
```

Ejemplo:

Programa que imprime un arreglo bidimensional.

```
@arreglo = ( [1, 2, 3], [4, -5, 6], [7, 8, 9, 99, 999], [10, 11, 12]);

for($i=0; $i<=$#arreglo; $i++) {

        for($j=0; $j<=$#{$arreglo[$i]}; $j++) {print $arreglo[$i][$j]."\t"};
        print "\n";

}
```

Un arreglo de tres dimensiones puede definirse de la siguiente manera

```
@arreglo = (   [[1,2,3],      [4,5,6],       [7,8,9]        ],
               [["a","b","c"],["d","e","f"], ["g","h","i"]  ],
```

```
                    [[-1,-2,-3],   [-4,-5,-6],    [-7,-8,-9]     ]
                );

        print $arreglo[0][1][2]."\n"; # Imrpime 6
        print $arreglo[2][2][1]."\n"; # Imprime -8
```

Pilas y colas con arreglos

La función shift extrae el 1er elemento de un arreglo y ademas lo elimina. La función pop extrae el último elemento de un arreglo y lo elimina. Por otro lado existen las funciones unshift y push que realizan lo inverso respectivamente. Unshift agrega un elemento o una lista de elementos al comienzo de una lista y push hace lo mismo, pero al final de la lista.

Ejemplo:

```
#!/usr/local/bin/perl

@a = ('a'..'e');
$b = shift(@a); # $b se instancia con 'a'
for ($n=0; $n<@a; $n++) {print $a[$n].” “}; print “\n”;
# Se imprimen los valores b, c, d y e

@a = ('a'..'e');
$b = pop(@a); # $b se instancia con 'e'
for ($n=0; $n<@a; $n++) {print $b[$a].” “}; print “\n”;
# Se imprimen los valores a, b, c y d

unshift(@a, 1); # agrega 1 al principio del arreglo
push(@a, 999); # agrega 999 al final del arreglo
for ($n=0; $n<@a; $n++) {print $b[$a].” “}; print “\n”;
```

Con las funciones descriptas es posible construir una estructura de datos denominada pila (stack); se caracteriza por que su acceso es LIFO, (last input first output) es decir que el último elemento añadido es el primero en ser leido, para lo cual se utiliza la función shift para extraer y la función unshift para insertar.

Toda estructura de datos llamada cola opera en modo FIFO (First Input First Output), donde el primer elemento en ingresar es el primer elemento en salir. Por ejemplo, se podría implementar una cola leyendo con shift y agregando con push. Alternativamente se podría leer con la función pop, y agregar con unshift.

A continuación se muestran invertidos una lista de números ingresados por teclado

Ejemplo:

```
#!/usr/local/bin/perl

@pila = ();                              # se crea la pila
print "Ingrese una lista de números  <terminar con -1> ";
$numero = <STDIN>;
while ($numero != -1) {
        unshift(@pila,$numero);          # se agrega un elemento a la pila
        $numero = <STDIN>;
}

$l = @pila;
for($i=1; $i<= $l; $i++) { print shift(@pila) };     # se extrae elemento de la pila
```

Arreglos Asociativos

Se caracterizan por que se puede definir cualquier tipo de dato escalar como índice. El caracter $ se utiliza para refererirse a un elemento escalar y el caracter % para todos. Para asignar valores a un arreglo asociativo, se utilizan las llaves ({ }) alrededor de la clave de índice. Para referirse a un arreglo asociativo en su totalidad, se usa su nombre precedido del símbolo "%"; si por el contrario se desea acceder solamente a un elemento se usa el nombre del arreglo anteponiendo el símbolo "$". Si lo que se quiere es manipular las llaves o los valores se usa el símbolo "@" antes del nombre.

Sobre los arreglos asociativos se se realizan tres operaciones principales:

1. obtener el valor asociado a una clave
2. cambiar el valor asociado a una clave.
3. Verificar si una clave existe.

En el siguiente ejemplo se define un arreglo asociativo llamado stock con sus pares de clave-valor,

```
%stock = (limones  => 6,  peras   => 3,   uvas  => 2);
```

A la estructura anterior se le pueden agregar ó modificar pares clave-valor de la siguiente forma

```
$stock{peras} = 9;
$stock{bananas} = 4;
```

Una forma de crear un arreglo asociativo es a partir de pares de datos clave-valor leidos de un archivo

Ejemplo:

```
open FILE, "frutas.txt" or die("Error $!");

while (<FILE>){
               $linea = chomp($_);           # se elimina el fin de linea
               @tmp   = split(/\t/);         # se separa el registo en campos
                                             # (el tabulador es el delimitador)

               $stock{$tmp[0]} = $tmp[1];    # se asigna par clave-valor
             }
close FILE or die("Error $!");
```

Un par clave-valor se borra con

```
delete $stock{uvas};
```

Un arreglo completo se borra

```
undef %stock;
```

Para eliminar todas los pares claves-valor, pero no la estructura, se utiliza

```
%stock = ();
```

Para navegar una estructura se pueden utilizar varias formas. La instruccion foreach permite recorrer la estructura de forma completa. La variable $clave se instancia con el valor de la clave.

```
foreach $clave (keys %stock) {
   print "$clave = $stock{$clave}\n";
}
```

También con el elemento map, es posible listar las claves.

```
print map "$_ = $stock{$_}\n", keys %stock;
```

Se puede utilizar while y each para leer una arreglo asociativo

```
while (($key,$valor) = each %stock){   print "$key = $valor\n"; }
```

La función keys crea un arreglo indexado con las claves de un arreglo asociativo

```
%a = ( x =>  5, y => 3, z => 'abc' );
@b = keys %a   #  @b se instancia con  ( 'x', 'y', 'z');
```

La función values devuelve un arreglo indexado con los valores de arreglo asociativo

```
%a = ( x =>  5, y => 3, z => 'abc' );
@v = values %a   #  @v se instancia con con  ( 5, 3, 'abc' );
```

La función exists prueba si existe una clave en un arreglo asociativo

```
%a = ( x =>  5, y => 3, z => 'abc' );
$b = exists $a{z};   #  $b se instancia con 1, es verdadero
$c = exists $a{w};   #  $c queda con "", es falso
```

Es posible ordenar una estructura. Para Perl existen tres formas: por orden ASCII, numericamente o alfabeticamente.

```
%hash = (Apples=>1, apples=>4, artichokes=>3,Beets=> 9,);
foreach my $key (sort keys %hash) { print "$key = $hash{$key}\n"; }
```

La salida será

```
Apples = 1
Beets = 9
apples = 4
artichokes = 3
```

A los efectos ordenar alfabeticamente se puede utilizar la función lc (convertir a minúsculas) y el operador de comparación cmp.

```
foreach my $key (sort {lc($a) cmp lc($b)} keys %hash) {
   print "$key = $hash{$key}\n";
}
```

Hash Slices: Un slice es un subconjunto de datos de una lista, arrreglo ó un hash. Utilizando slices es posible agregar o borrar pares de clave-valor en masa.

```
@meses = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
@num_meses{@meses}= 1..$#meses+1;

foreach my $key (keys %num_meses) {
   print "$key = $num_meses{$key}\n";
```

```
}
```

En el ejemplo anterior a partir de la lista meses se construyo el hash num_meses

El siguiente código permite listar los pares clave-valor ordenados por valor

```
foreach my $key (sort {$num_meses{$a} <=> $num_meses{$b}} keys %num_meses){
   print "$key = $num_meses{$key}\n";
};
```

Es posible definir distintas formatos de ordenamiento utilizando un puntero de reemplazo. El siguiente ejemplo muestra la técnica de uso de subrutinas para ayudar al ordenamiento.

Ejemplo:

```
foreach my $key (sort ascend_alpha keys %hash){
        print "$key = $hash{$key}\n";
}

        sub ascend_num          {$a <=> $b}
        sub descend_num         {$b <=> $a}
        sub ascend_alpha        {lc($a) cmp lc($b)}
        sub descend_alpha       {lc($b) cmp lc($a)}
        sub ascend_alphanum     {$a <=> $b || lc($a) cmp lc($b)}
        sub descend_alphanum    {$b <=> $a || lc($b) cmp lc($a)}
```

Referencias a hashes

```
$hashref = \%hash;

foreach my $key (sort ascend_alpha keys %{$hashref}){
   print "$key = $hashref->{$key}\n";
}
```

En el bloque se utiliza la referencia ($hashref) al hash

Hashes multidimensionales: Se utilizan para asociar mas de un valor a la clave, el ejemplo siguiente lo muestra

```
%hash = (
  Apples               => [4, "Delicious red", "medium"],
  "Canadian Bacon"     => [1, "package",       "1/2 pound"],
  artichokes           => [3, "cans",          "8.oz."],
  Beets                => [4, "cans",          "8.oz."],
  "5 Spice Seasoning"  => [1, "bottle",        "3 oz."],
  "10 Star Flour"      => [1, "package",       "16 oz."],
  "911 Hot Sauce"      => [1, "bottle",        "8 oz."],
);
```

Para extraer un valor particular, al conjunto de valores debe tratarselo como un arreglo.

```
print $hash{"Canadian Bacon"}[1];
```

La instrucción anterior mostrará por pantalla el valor package que es el segundo elemento. Es posible asignar un arreglo a un valor de clave.

```
@garlicstuff          = (4, "cloves", "medium");
$hash{"Garlic"}       = [@garlicstuff];
print $hash{"Garlic"}[1]; # Imprime  cloves
```

Es posible asociar arreglos de longitud variable a una misma estructura hash. El siguiente programa permite imprimir un hash con arreglos de distinta longitud asociados a las claves.

```
foreach my $key (sort ascend_alpha keys %hash){
   print "$key: \n";
   foreach my $val (@{$hash{$key}}){ print "\t$val\n"; }
   print "\n";
}
```

Operadores

" No puedo decir que no estoy en desacuerdo contigo". Groucho Marx

En Perl los valores falsos son:

En cadenas "" y "0"
En números, el número 0.
El valor "no definido", cuando una variable existe, pero no tiene valor ligado

Todos los demas valores son verdaderos

Los operadores clásicos son:

Operación	*Para aritmética*	*Para cadenas*
Mayor que	**>**	gt
Menor que	**<**	lt
Mayor o igual	**>=**	ge
Menor o igual	**<=**	le
Igual	**==**	eq
Diferente	**!=**	ne
Compara	**<=>**	cmp

El operador compara retorna un valor de -1, 0 o 1 dependiendo si el valor a la izquierda es menor, igual o mayor que el de la derecha del operador. Perl considera como verdaderos los valores no nulos (cadenas) y diferentes de cero.

Algunos operadores lógicos que permiten determinar el valor de verdad de una expresión son:

Operador	*Función*
!	Negación Lógica
-	Inverso aditivo (cambia el signo de un numero)
~	Negación Bit a bit de un valor
&&	AND - Conjunción lógica de los operandos
||	OR - Disyunción lógica de los operandos
&	AND - Conjunción bit a bit de los operandos
|	OR - Disyunción bit a bit de los operandos
^	XOR - Or exclusiva bit a bit
..	Si evaluado como escalar regresa alternativamente verdadero y falso. Como arreglo genera un arreglo a partir de los limites de un rango.
not	Negación lógica de la expresión a su izquierda (sinónimo de baja prioridad)
and	Conjunción lógica (sinónimo de baja prioridad)
or	Disyunción (sinónimo de baja prioridad)
xor	Disyunción exclusiva lógica (sinónimo de baja prioridad))

Ejemplo:

```
if ( $a and $b ) { print "A y B son verdaderos"; } # $a y $b han sido definidas
if ( $a or $b )  { print "A o B son verdaderos"; } # $a y/ó $b han sido definidas
```

Existen los operadores **&&** y **||** (como en el lenguaje C) y poseen mas prioridad que and y or.

comparación de números

```
$x = $a <=> $b

       # $x queda con -1 si $a < $b
       # $x queda con 0 si $a == $b
       # $x queda con 1 si $a > $b
```

comparación de strings

```
$x = $a cmp $b

       # $x queda con -1 si $a lt $b
       # $x queda con 0 si $a eq $b
       # $x queda con 1 si $a gt $b
```

Así como los operadores aritméticos fuerzan a la variable a comportarse como de tipo entero o real, los operadores de cadena fuerzan a los escalares a comportarse como cadenas, de modo que los números son automáticamente convertidos en cadenas al aplicarles estos operadores.

Si una variable está indefinida, al ser evaluada da una cadena vacía, lo cual significa falso. Para definirla, se puede aplicar la siguiente técnica:

```
$costo = 100 unless $costo;  # Si costo no esta definido, inicializarlo con 100
```

La función defined se utiliza para averiguar si una variable esta definida

```
$a = 5;
print "variable a definida" if (defined $a);
print "variable b no definida" if (!defined $b);

$costo = 100 unless defined($costo);  # Acuerdese que lo que está a la izquierda
                                      # de unless se ejecuta si la expresión de
                                      # la derecha da falso.
```

Nota: la función undef lo que hace es desligar un valor de una variable.

Las operaciones a realizar con cadenas, por lo regular se manejan como funciones en vez de como operadores, con la excepción de las siguientes:

Operador	*Función*
.	Concatenación, todas las cadenas operadas por "." forman una sola cadena. $a = "Hola"; $b = "mundo"; $c = $a." ".$b;
x	Operador de repetición, en contexto escalar toma cadena de la izquierda y la repite el numero de veces que el numero de la derecha indica. En contexto de una lista, toma la lista a su izquierda y la repite las veces que indica el operador a su derecha en una nueva lista. $bar = '-' x 72; # devuelve una cadena de 72 guiones.

++	Autoincremento, modifica el valor de los elementos de una cadena de modo que se incrementen sin cambiar su naturaleza. Por ejemplo ++"zzz" da "aaaa"; ++"99" da "100" , ++"a9" da "b0".
..	Operador que permite definir rangos. Por ejemplo: @a = "ay" .. "bb"; @a queda con ("ay", "az", "ba", "bb") @a = 1..5; @a queda con (1, 2, 3, 4, 5)

Estructuras de Control

"¿Por qué las cosas son como son y no de otra manera?" Blaise Pascal

Las estructuras principales de Perl y su sintaxis son:

IF:

if (EXPRESIÓN) BLOQUE
if (EXPRESIÓN) BLOQUE else BLOQUE
if (EXPRESIÓN) BLOQUE elsif (EXPRESIÓN) BLOQUE ... else BLOQUE

WHILE:

while (EXPRESIÓN) BLOQUE
do BLOQUE until | while (EXPRESIÓN)

FOR:

for (EXPRESION;EXPRESION;EXPRESION) BLOQUE

FOREACH:

foreach VARIABLE (LISTA) BLOQUE

Donde:

BLOQUE es un conjunto de comandos agrupados por los caracteres {}
EXPRESION es una proposición lógica que evaluada devuelve un valor de verdad.
LISTA es un conjunto de valores separados por coma.

Ejemplo if

Ejemplo:

```
$objeto = "tiza";
$requerido = "tiza";
$cantidad = 22;

if ($objeto eq "tiza") {

        print "la variable objeto esta instanciada con  tiza\n";

} else {

        print "la variable objeto NO esta instanciada con  tiza\n";
}

if ($cantidad <= 25) {print "Hay que reponer ".((25-$cantidad)*3). " elementos\n";}
```

Ejemplo if-elsif-else

Si la primer expresión es verdadera ejecutar bloque, de lo contrario evaluar las expresiónes elsif, si alguna es verdadera ejecutar bloque correspondiente, si finalmente se obtiene falso ejecutar bloque definido en else

Ejemplo:

```
#!/usr/local/bin/perl
print ("Ingrese el dividendo:\n");
$dividendo = <STDIN>; chop ($dividendo);
print ("Ingrese el divisor:\n");
$divisor = <STDIN>; chop ($divisor);

if ($divisor == 0) { print ("Error: no puedo dividir por cero!\n"); }
elsif ($dividendo == 0)
        { $resultado = $dividendo; }
elsif ($divisor == 1)
        { $resultado = $dividendo; }
else { $resultado = $dividendo / $divisor; }

if ($divisor != 0)
        { print ("El resultado es ", $resultado, "\n"); }
```

Ejemplo unless

La entructura unless, equivale a la negación de if, es decir que si la condición de unless da por resultado falso se ejecutan las intrucciones del bloque.

Ejemplo:

```
#!/usr/local/bin/perl

print "Ingrese el nombre de un sistema operativo: ";
chop($nombre=<STDIN>); # no toma el ultimo caracter ingresado

if($nombre=="linux")
{ print "Ese si es un buen producto "; }
else
{ print "Le pedi que ingresara un nombre de sistema operativo"; }


#!/usr/local/bin/perl

print "Escriba un numero menor o igual a 10: ";
chop($numero=<STDIN>);

unless($numero <= 10) { print "Error $numero no es menor igual a 10"; }
```

Ejemplo while

Ejecutar bloque siempre que el resultado de la evaluación de la expresión sea verdadero

Ejemplo:

```
#!/usr/local/bin/perl

$i=0;
$suma=0;
while($i<10) { $suma = $suma+$i; $i++ }

print "El resultado de la suma es: $suma";
```

Ejemplo do while

Ejemplo:

```
#!/usr/local/bin/perl

do {

        $linea = <STDIN>;

} while ( $linea ne "\n" );
```

Ejemplo until

Ejemplo:

```
#!/usr/local/bin/perl

$i=0;
$suma=0;
until($i != 10) { $suma= $suma + $i; $i++ }

print "El resultado de la suma es: $suma";
```

Ejemplo do until

Ejemplo:

```
#!/usr/local/bin/perl


do {

        $linea = <STDIN>;

} until ($linea eq "\n");
```

Ejemplo for

Itere mientras la condición no se cumpla.

Ejemplo:

```
#!/usr/local/bin/perl
```

```
$suma=0;
for($i=0; $i<10; $i++) { $suma = $suma + $i }

print "El resultado de la suma es $suma \n";
```

Ejemplo foreach

Itera la cantidad de elementos que haya en la lista de datos, en cada vuelta toma el enésimo elemento y lo asigna a una variable.

Ejemplo:

```
#!/usr/local/bin/perl

@lista = (7, 11, 22, 5, 6, 7, 45);
foreach $elemento (@lista) { print $elemento." "; }
```

Ejemplo last y next

En Perl, la instrucción last se utiliza para forzar la finalización de un ciclo while . En cambio la instrucción next fuerza inmediatamente a la evaluación de la condición, saltando las siguientes instrucciones del bloque.

Ejemplo:

```
#!/usr/local/bin/perl

$total = 0;
while (1) {

        $linea = <STDIN>;             # Ingresar n números por línea, el
                                      # separador es el tabulador

        if ($linea eq "\n") {last}    # last indica que se debe finalizar el
                                      # ciclo while en caso de que se ingrese
                                      # una línea vacia

        chop ($linea);
        @numeros = split("\t", $linea); # Extraer los n números ingresados

        foreach $num (@numeros) {     # validar si cada elemento es un número

                if ($num =~ /[^0-9]/) {print "$num no es un número\n"}
                $total += $num;

        }

}

print "La suma de los números ingresados es $total \n";
```

Ejemplo:

```
#!/usr/local/bin/perl

@arreglo = (0..9);
```

```
print("@arreglo\n");

for ($indice = 0; $indice < @arreglo; $indice++) {

    if ($indice == 3 || $indice == 5) {
        next;
    }
    $arreglo[$indice] = "*";

}

print("@arreglo\n");
```

Funciones y subrutinas

" No creo en una vida posterior, pero por si acaso me he cambiado de ropa interior". Woody Allen

Perl no hace diferencias entre funciones y subrutinas. Una función se define:

```
sub mifunción {
        print "Soy una función \n";
}
```

y se la invoca:

```
&mifuncion;
```

Si se quieren enviar y recibir parámetros:

```
sub maximo {

        if ($_[0] > $_[1])            # Por defecto en el arreglo $_ se almacenan los
                { return($_[0]); }    # parámetros recibidos
        else
                { return($_[1]); }
}

$a = 22; $b = 55;
$mayor = &maximo($a, $b);
```

Por defecto, si no se especifica el valor a devolver con return, el valor retornado por la función es el último valor evaluado dentro de su bloque.

```
sub saludar
  {
    print "¡Hola $_[0]!\n";
  }

saludar("Cesar");
saludar("Sandra");
```

El pasaje de parámetros, realizado en el siguiente ejemplo es por referencia, es decir que se envían un punteros indicando la dirección de memoria donde están los datos. Si dentro de la función se modificara algún valor sobre los parámetros pasados, se modificarían las variables originales. Luego de ejecutar el siguiente programa, muestre los valores de $a, $b y saque sus propias conclusiones.

```
sub maximo {

        ($_[0], $_[1]) = ($_[1], $_[0]); # se realiza un swap
        if ($_[0] > $_[1])
                { return($_[0]); }
        else
                { return($_[1]); }

}

$a = 22; $b = 55;
$mayor = &maximo($a, $b);
```

Variables locales

Una variable declarada con "my" es local en el ámbito de la subrutina. Se aconseja la práctica de copiar los argumentos a variables locales. Con esto se consiguen dos efectos: darles nombres más significativos, y hacer que el traspaso sea por valor (el default es por referencia).

```
sub suma
  {
    my($a,$b) = @_;   # @_ es la lista de los parámetros recibidos

    $a+$b;            # se podría haber devuelto el resultado
  }                   # con la función return

print &suma(2,3)."\n";
```

Si se tiene un número variable de parámetros se los podría recibir en una lista local.

```
  print &suma(3,35,22,5)."\n";
  print &suma(1..8)."\n";

sub suma
  {
            my @lista = @_;
            my $x,
            my $s = 0;

            foreach $x (@lista)  { $s += $x; } # += suma a $s y asigna }
            return($)
  }
```

Ejemplo:

Multiplicación por medio de sumas sucesivas

```
#!/usr/bin/perl

# programa principal

if(!defined($ARGV[1])) {die "Uso: multiplica.pl x y"}
$x = $ARGV[0]; $y = $ARGV[1];

$z = &multiplica ($x, $y);

print "El resultado de $x * $y es $z\n";


# subrutina

sub multiplica {

    local( $x ) = @_[0];
    local( $y ) = @_[1];

    local( $z ) = 0;
    local( $i );

    for ( $i = 1; $i <= $y; $i++ ) {

        $z = $z + $x;

    };

    return $z;
};
```

Ejemplo:

Verificar si un número es primo

```
#!/usr/bin/perl

# programa principal

if(!defined($ARGV[0])) {die "Uso: primo.pl x"}

$x = $ARGV[0];

if ( &primo( $x ) ) {
    print "$x es un numero primo\n";
} else {
    print "$x no es un numero primo\n";
};


# subrutina

sub primo {

    local( $x ) = @_[0];

    local( $primo ) = 1;
    local( $i ) = 2;
    local( $j );

    while ( ( $i < $x ) && $primo ) {

        $j = int( $x / $i );
        $j = $x - ( $j * $i );

        if ( $j == 0 ) {
            $primo = 0;
        } else {
            $i++;
        };
    };

    return $primo;
};
```

Recursividad

Ejemplo:

Cálculo de la potencia

```
#!/usr/bin/perl

# programa principal

if(!defined($ARGV[1])) {die "Uso: potencia.pl x y"}

print $ARGV[0]." a la ".$ARGV[1]." = ".&potencia($ARGV[0],$ARGV[1])."\n";


# subrutina

sub potencia  {

          local($a,$n) = @_;

            if( $n == 0 )
              {
```

```
                1;
              }
            elsif( $n%2 == 1 )
              {
                $a*potencia($a,$n-1);
              }
            else
              {
                potencia($a*$a,$n/2);
              }

}
```

Factorial

```
$x = $ARGV[0];

$fatt = &fatt( $x );

print "$x! = $fatt\n";

sub fatt {
    local( $x ) = @_[0];

    if ( $x > 1 ) {
        return ( $x * &fatt( $x - 1 ) );
    } else {
        return 1;
    };

};
```

Paquetes y Módulos

En ciertas ocasiones, los programas que se construyen son demasiado grandes ó por una cuestión de organización interesa poder dividir el código en fragmentos más pequeños, que se puedan reutilizar. Perl proporciona dos soluciones:

Paquete: Es el equivalente a la instrucción #include del lenguaje C, se pueden tener funciones por separado, pero al ejecutarse se juntará todo en un único script. `Un dato` importante que debe tenerse en cuenta al construir un archivo tipo paquete es no olvidar incluir como última línea una sentencia cuyo valor lógico sea **cierto**. La forma más común de conseguirlo es que figure al final del mismo la línea 1;

Paquete suma

```
sub suma
{
local($a,$b)=@_;
$a+$b;
}
1;
```

Programa ejemplo.pl

```
require "suma";           # Se carga el paquete suma
$resultado = &suma(5, 2);
print "El resultado es $resultado \n";
```

Módulos: Los módulos son elementos más avanzados que los paquetes, son una especie de librerías y pueden cargarse de forma dinámica. Este tema no es objetivo de este documento.

Referencias

"No debe haber barreras para la libertad de preguntar. No hay sitio para el dogma en la ciencia.
El científico es libre y debe ser libre para hacer cualquier pregunta,
para dudar de cualquier aseveración, para buscar cualquier evidencia, para corregir cualquier error.
Mientras los hombres sean libres para preguntar lo que deben; libres para decir lo que piensan;
libres para pensar lo que quieran; la libertad nunca se perderá y la ciencia nunca retrocederá"
Glenn Theodore Seaborg, físico estadounidense

Las referencias son escalares que almacenan la dirección de memoria de otra variable. Para visualizar tal concepto pruebe:

```
$a = "Hola Mundo";
print $a."\n";  # se muestra el contenido de $a
print \$a."\n"; # se muestra el puntero a $a, es decir la dirección de memoria de $a
```

Otras referencias:

```
$refa = \$a;   # referencia a escalar
$refb = \@b;   # referencia a arreglo
$refc = \%c;   # referencia a arreglo asociativo
$refx = \$rb;  # referencia a referencia
```

Arreglos anónimos construidos por referencias

Se crea un arreglo sin nombre (observe el parentesis recto)

```
$refb = [ 'd1', 'd2', 'd3'];
       # en refb se almacena la referencia al arreglo sin nombre
```

Idem, pero con un arreglo asociativo

```
$refc = { c1 => 'd1', c2 => 'd2' };
       # en refc se almacena la referencia al arreglo asociativo sin nombre
```

Utilizando una referencia a una referencia se obtiene el dato real

```
       $ra = \$a;  # referencia a escalar
       $rb = \@b;  # referencia a arreglo
       $rc = \%c;  # referencia a hash
       $rx = \$rb; # referencia a referencia

       ${$ra}  es la desreferencia de $ra... el valor de $a
       @{$rb}  es la desreferencia de $rb... el valor de @a
       @{$ra}  es un error porque $ra apunta a un escalar
       %{$rc}  es la desreferencia de $rc... el valor de %c
```

Ejemplo:

Búsqueda secuencial de un entero en una lista

```
#!/usr/bin/perl

# Programa principal

if(!defined($ARGV[0])) {die "Uso: buscarseq.pl x"}

$x      = $ARGV[0];
@lista  = (0..100);
```

```
$i = &busqueda_secuencial(\@lista, $x);

print "\nEl elemento $x estaba en la posición $i\n";

# Procedimiento &busqueda_secuencial( <puntero_a_lista>, <numero_a_buscar> )

sub busqueda_secuencial {

    local($lista)      = @_[0];
    local($x)          = @_[1];
    local($i);

    for ( $i=0; $i<=$#{$lista}; $i++ ) {

print $listas[$i]." ".$xx;
        if ( $x == ${$lista}[$i] ) {
            return $i;
        };
    };

    # El número no estaba en la lista
    return -1;
};
```

Veamos una manera de accesar los elementos de un hash usando una referencia

```
$rc = { a => 1, b => 2 };
       # $rc es una referencias a un hash anónimo

print  $rc->{a};
       # imprime: 1
```

La función ref devuelve un string que indica el tipo del referenciado

```
$ra = \$a;  # referencia a escalar
$rb = \@b;  # referencia a arreglo
$rc = \%c;  # referencia a hash
$rx = \$rb; # referencia a referencia
$rf = \&f;  # referencia a función

ref ( $ra ); #  devuelve "SCALAR"
ref ( $rb ); #  devuelve "ARRAY"
ref ( $rc ); #  devuelve "HASH"
ref ( $rx ); #  devuelve "REF"
ref ( $rf ); #  devuelve "CODE"
```

Si el operando de ref no es una referencia, ref devuelve falso ó "no definido"

Diferencia entre trabajar con variables y trabajar con referencias...

```
$a = 5;
$b = $a;
       # $b recibe una copia del valor de $a

$a = 7;
       # cambio el valor de $a

print  "$a  $b\n";
       # imprime: 7  5
       # un cambio en $a no afecta a $b

$a = 5;
$ra = \$a;
$rb = $ra;
$a  = 7;
       # cambio el valor de $a

print  "$a  $$ra  $$rb \n" ;
       # imprime  7  7  7
```

```
# las referencias si comparten en mismo valor...
# el valor 7 aqui tiene 3 usuarios...
```

Expresiones Regulares

" Claro que lo entiendo. Incluso un niño de cinco años podria entenderlo. ¡Que me traigan un niño de cinco años!"
Groucho Marx

Una expresión regular es una forma de expresar gramaticalmente la estructura de cualquier cadena alfanumérica. Una expresión regular es un patrón que se compara con una cadena de caracteres. Esta comparación es conocida con el nombre de *pattern matching* o reconocimiento de patrones, permite identificar las ocurrencias del patrón en los datos tratados.

Las expresiones regulares en Perl se utilizan para la búsqueda, modificación y extracción de cadenas de caracteres. De esta manera se pueden dividir las expresiones regulares en varios tipos que son: expresiones regulares de de comparación, sustitución y de traducción.

m//	Coincidencia (match) (la m puede obviarse)
s//	sustituir
tr//	traducir

Ejemplos

```
if($texto =~ m/unlu/) { print "el escalar texto contiene la cadena unlu"}

$copia =~ s/copy/copia/g; # Si en la cadena $copia aparece la palabra
                          # copy se reemplaza por copia

$entrada =~ tr/A-Z/a-z/; # Convierte a minúsculas la cadena $entrada
```

Metacaracteres

\	Caracter de escape (Ej. \t tabulador, \" comillas)
^	Coincidencia al inicio
$	Coincidencia al final
|	O lógico

Constructores

Contrucción	Significado	Contrucciones equivalentes
\d	Un dígito	[0-9]
\D	Cualquier cosa que no sea digito	[^0-9]
\w	Un carácter alfanumérico	[a-zA-Z0-9]
\W	Cualquier cosa que no sea un carácter alfanumérico	[^a-zA-Z0-9]
\s	Un espacio, retorno de carro, tabulador, etc	[\n\r\t\f]
\S	Cualquier cosa que no sea un espacio, etc	[^ \n\r\t\f]
[a|b|c]	Cualquier linea que contenga la letra 'a', 'b' o 'c'	[a-c]

Cuantificadores

.	Unnico caracter
?	Coinicidencia de 0 ó 1 vez.
*	Coinicidencia de 0 ó más veces
+	Coinicidencia de 1 ó más veces

Operadores

g	Reemplaza todas las ocurrencias
i	Chequeo no sensitivo
o	Evalua la expresión a reemplazar una sola vez

Ejemplos

Verificar si una cadena está contenida en otra

```
$_ = "Hola Mundo!!!";
print "Si ola esta contenida\n" if (/ola/);

if($texto =~ /unlu/) { print "el escalar texto contiene la cadena unlu"}

if($texto =~ /unlu/i) { print "el escalar texto contiene la cadena unlu"}
# comparación no sensitiva

if($texto !~ /unlu/i) { print "el escalar texto NO contiene la cadena unlu"}
```

Verificar si una subcadena está al comienzo de otra cadena

```
$a = "Universidad Nacional de Luján";

if ($a =~ /^Uni/){# código a ejecutar en caso de que la expresión sea verdadera}
```

ó al final

```
if ($a =~ /ján$/)

        {# código a ejecutar en caso de que la expresión sea verdadera}
```

Reemplaza copy con copia a lo largo de la cadena $copia

```
copia ="copy, xcopy, diskcopy";
$copia =~ s/copy/copia/g;

# El contenido de la variable $copia será: "copia, xcopia, diskcopia"
```

Elimina caracteres

```
$cadena = "abcdef";
$cadena = s/cd//;
print $cadena;  #  Imprime: abef
```

El patrón de traducción (tr) permite modificar el contenido de una variable. En el siguiente ejemplo se reemplaza la lectra c por 1, la o por 2, la p con 3, la y con 4 y el 5 se ignora.

```
$copia ="copy, xcopy, diskcopy";

$copia =~ tr/copy/12345/;

# El contenido de la variable $copia será: "1234, x1234, disk1234"
```

Conversión a minúsculas

```
$entrada = "La Casa de Ana";

$entrada =~ tr/A-Z/a-z/; # Convierte a minúsculas

print $entrada."\n";
```

La siguiente expresión regular acepta solo números

```
$variable = "1a2b3c";
$variable =~ tr/0-9//cd;
print $variable."\n";
```

Reemplaza el caracter = con un espacio

```
$a = "f1=abc test=on";
$a = tr/=/ / ;  # $a queda "f1 abc test on"
```

La siguiente expresión reemplaza cada "a" por "e", cada "b" por "d" y cada "c" por "f" en la cadena $cadena:

```
$cadena = "cabeza";
$cabeza =~ tr/abc/edf/;
print $cadena."\n"; Imprime fedeze
```

La siguiente expresión retorna el número de sustituciones hechas. Se cuentan el número de asteriscos en la cadena $cadena:

```
$cadena = "Si no *** fuera * por la * suerte *";
$contador = ( $cadena =~ tr/*/*/ );
```

Otros ejemplos

d.l	una "d" seguida de un caracter cualquiera y una "l" (del, dal, dzl, d5l, etc)
^f	una "f" al principio de la cadena (fofo, farfolla, f35, etc)
^hol	"hola" al principio de la cadena (hola, holita, etc)
e$	una "e" al final de la cadena (este, ese, etc)
te$	"te" al final de la cadena (este, paquete, etc)
ind*	"in" seguido de cero o más caracteres "d" (in, ind, indd, etc)
.*	cualquier cadena, sin retorno de carro
^$	una cadena vacia
[qjk]	una "q", o una "j" o una "k"
[^qjk]	no sea "q", o una "j" o una "k"
[a-z]	cualquier letra entre la "a" y la "z"
[^a-z]	no sean letras minúsculas
[a-zA-Z]	una letra minúscula o mayúscula
[a-z]+	una secuencia no vacia de letras minúsculas
f.*ca	coincide con "fca", "foca", "flaca", "flor vaca", etc
f.+ca	coincide con los anteriores salvo con "fca"
fe?a	coincide con "fa" y "fea"
^[\t]*$	una linea en blanco, o combinaciones de espacios y tabuladores
[-+]?\d*\.?\d*	lo mismo que [-+]?[0-9]*\.?[0-9]* (números decimales)
pepe|juan	o "pepe" o "juan"
(pe|hue)cos	o "pecos" o "huecos"
(da)+	o da o dada o dadada ...
[01]	un "0" o un "1"
fia|fea|fua	coincida con "fia", "fea" o "fua"
f(i|e|u)a	coincida con "fia", "fea" o "fua"

Archivos

"La ciencia son hechos; de la misma manera que las casas están hechas de piedras, la ciencia está hecha de hechos; pero un montón de piedras no es una casa y una colección de hechos no es necesariamente ciencia."
Alfred North Whitehead, matemático inglés.

Entrada estandard

Para leer una linea desde la entrada estandard se debe asignar una variable escalar a <STDIN>

```
$a=<STDIN>;     # guarda en $a la linea leida
chop($a);       # se saca el enter que tiene la linea ingresada
```

Como procesar toda la entrada estandar:

```
while ( $a=<STDIN> )
        {
                bloque de procesamiento ;
        }
```

Se leen una serie de líneas ingresadas por teclado y se las almacenan como elementos de un arreglo.

```
print "Ingrese varias líneas de texto <Ctrl-Z para finzalizar>\n";

@l = <STDIN>;  # las líneas se almacenan como elementos de un arreglo
print "\n"; foreach $i (@l) {print "*".$i}
```

Archivos perl

Perl accede a los archivos por medio de manejadores de archivos. Los archivos se abren mediante la instrucción *open* que admite dos argumentos, un manejador de archivo y un nombre de archivo:

```
open(PUNTERO, "<modo de acceso><nombre de archivo>");
```

El argumento nombre puede estar asociado a una serie de prefijos que indiquen la modalidad de acceso, por ejemplo: >> indica modo añadir (append). La siguiente tabla muestra los modos existentes

Modo	Descripción
<	Abrir para lectura (por defecto)
>	Abrir en modo grabación. Si el archivo existe se borrará todo su contenido.
>>	Abrir en modo añadir, a los efectos de grabar al final del archivo
|	La salida de una script se redirecciona al programa.
+>	Abre un archivo en modo lectura-escritura, si el archivo no existe lo creará.
+<	Abre un archivo en modo lectura-escritura, si el archivo no existe reportará error.

Existen dos archivos que no requieren abrirse ni cerrarse, y son la entrada (manejador STDIN) y la salida standar (manejador STDOUT). El lenguaje Perl no necesita que se mencionen explícitamente, en lugar de ello, los crea y les asigna canales automáticamente.

Leer registros

EL operador <manejador_de_archivo> permite leer registros de un archivo. El siguiente ejemplo indica como acceder al contenido del archivo *datos.dat, pasar a minúsculas sus letras* y guardar el resultado en el archivo salida.dat

```
open(IN, "datos.dat");        # se abre para lectura el archivo datos.dat,
                              # el manejador IN lo referencia
open(OUT, ">salida.dat");     # se abre para grabación el archivo salida.dat,
                              # el manejador OUT lo referencia

while(<IN>) {                 # mientras no sea fin de archivo lee un registro

       $linea = $_;           # el registro leido se guarda en la variable linea
       $linea =~ tr/A-Z/a-z/; # se pasan a minúsculas todas las letras

       print OUT $linea;      # se graba la linea en el archivo de salida

}

close(IN);
close(OUT);
```

Nótese que el comando close se utiliza a los efectos de cerrar un archivo.

Función die

La función die es útil para finalizar la ejecución de un programa ante la existencia de errores de ejecución. En el siguiente ejemplo si fallará la apertura del archivo el operador or (||) disparará la ejecución de la función die que interrumpirá la ejecución del programa y mostrará un mensaje de error.

```
$archivo = "datos.dat";

open(INPUT, $file)     || die "no se puede abrir el archivo\n";

@lineas  = <INPUT>     # Notese que leerá todo el archivo. y lo almacena
                       # en un arreglo
close(INPUT);
```

Pipes

En el modo de apertura de un archivo, el caracter | (pipe) es utilizado a los efectos de ejecutar un comando externo y redireccionar su salida al programa Perl. Cuando el nombre del archivo lleva como prefijo el carácter |, este nombre se trata como una orden de ejecución. Tal orden se ejecuta y su salida es tomada como un archivo. Ejemplo:

```
open(WHO, "|who");            # determina los usuarios de un sistema
while($who = <WHO>) {

       chop ($who);           # elimina el caracter de retorno de carro
       ($user, $tty, $junk) = split(/\s+/, $who, 3);
       print "El usuario $user está en la terminal $tty\n";

}
close(WHO);
```

Operadores de comprobación de archivos

Aportan distintas informaciones sobre un archivo, por ejemplo el operador -A devuelve la última fecha de acceso. La siguiente tabla muestra los operadores más comunes:

Operador	Descripción
-A	Fecha de último acceso
-e	El archivo o directorio existe
-r	El archivo existe y se tienen permisos para acceder
-w	El archivo es modificable
-x	El archivo es ejecutable
-M	Fecha de última modificación

Por ejemplo:

```
$archivo = "datos.dat";
if (e- $archivo)       { # El archivo existe }
else                   {print "el archivo $archivo no existe.\n"; }
```

Grabación

A los efectos de grabar sobre un archivo secuencial se usa la instrucción print y a continuación el manejador de archivo que indica el archivo sobre el cual se desea grabar. Ejemplo

```
open(SALIDA, ">>datos.dat");  # se abre el archivo en modo añadir
print SALIDA "346\tPerez, Jorge\tSan Luis 347\n";
close(SALIDA);
```

Fijese que en el ejemplo anterior se grabó un registro (tres campos separados por tabulador) sobre el archivo datos.dat. Como en C se puede utilizar la instrucción printf a los efectos de aplicarle máscaras de formato a los datos a grabar.

```
Programa que chequea  si un archivo existe y está disponible

#!/usr/local/bin/perl

unless (open(FILE, "miarchivo.txt")) {
      if (-e "file1") {
                die ("El archivo existe, no puede ser abierto.\n");
      } else {
                die ("El archivo no existe.\n");
      }
}

$linea = <FILE>;

while ($linea ne "") {
      chop ($linea);
      print ("\U$linea\E\n");
      $line = <FILE>;
}
```

El siguiente programa es un ejemplo de lectura y grabación sobre archivos secuenciales. La script determina que se debe leer el archivo secuencial(catalg.lin) y copiar su contenido al archivo secuencial sucopia.lin, traduciendo los caracteres alfabéticos de mayúsculas a minúsculas.

```
open(IN, "catalg.lin");       # se abre para lectura el archivo catalg.lin
                              # el manejador IN lo referencia
open(OUT, ">sucopia.lin");    # se abre para grabación el archivo sucopia.lin
                              # el manejador OUT lo referencia

while(<IN>) {          # mientras no sea fin de archivo lee un registro
```

```
$linea = $_;           # el registro leido se guarda en la variable linea
chop($linea);
$linea =~ tr/A-Z/a-z/;  # se pasan a minúsculas todos los caracteres
alfabéticos

print OUT $linea."\n"; # se graba la linea en el archivo de salida

}

close(IN);
close(OUT);
```

Funciones para manejo de archivos

read(manejador, variable, longitud, [desplazamiento])

Esta función lee desde un archivo, especificado en el parámetro *manejador*, un número de bytes, especificado por el parámetro *longitud,* y lo introduce en una variable de tipo escalar representada por el parámetro *variable*. El parámetro *desplazamiento*, si se especifica, indica desde que posición del archivo se empieza a leer. Por defecto, es la posición donde se encuentra el puntero. He aquí algunos ejemplos:

```
open(INPUT,"<datos.dat");
read(INPUT, $var, 60);     # $var almacena los primeros 60 bytes del
archivo
```

Programa destinado a leer un archivo relativo (catalg.rel) y mostrar por pantalla sus registros. La estructura del archivo es la siguiente:

```
Longitud de registro: 50 bytes

 Campos
              ID-articulo     4 bytes (de byte  0 a  3)
              articulo       18 bytes (de byte  4 a 21)
              marca           8 bytes (de byte 22 a 29)
              pais origen     8 bytes (de byte 30 a 37)
              meses garantia  2 bytes (de byte 38 a 39)
              stock           3 bytes (de byte 40 a 42)
              precio          7 bytes (de byte 43 a 49)
```

```
$linea = '-'x57; print "$linea\n";

open(DAT,"<catalg.rel");

while(not(eof(DAT))) {

  read(DAT, $buffer, 50);      #Lee 50 bytes del archivo DAT y almacena su
                               #contenido en una variable de memoria
                               # denominada buffer

       &parsing_e_imprime();  #Extrae contenido de los campos a variables
                               # de memoria e imprime


}
print "$linea\n";

close(DAT);

sub parsing_e_imprime {

       $id_articulo = substr($buffer,0,4 );
       $articulo    = substr($buffer,4,18);
       $marca       = substr($buffer,22,8);
```

```
        $pais        = substr($buffer,30,8);
        $garantia    = substr($buffer,38,2);
        $stock       = substr($buffer,40,3);
        $precio      = substr($buffer,43,7);

        print   "|".$id_articulo."|".
                $articulo."|".
                $marca."|".
                $pais."|".
                $garantia."|".
                $stock."|".
                $precio."|\n"; # Imprime registro

        return;
}
```

seek(manejador, posición, referencia)

Seek mueve pone el puntero de archivo a la posición especificada (valores positivos avanzan y negativos retroceden) El desplazamiento indicado por el parámetro posición se puede hacer desde el principio del archivo, desde la *posición* del puntero actual del archivo o desde el fin del archivo. Esto lo determina el parámetro referencia con los valores 0,1 y 2 respectivamente. Ejemplo:

```
open(INPUT,"<datos.txt");
read(INPUT, $variable, 22);
seek(INPUT,-23,1);                # se retorna al inicio
read(INPUT, $variable, 22);       # se vuelve a leer lo mismo
```

```
seek( MIOFILE, 0, 2 );
Posiciona el puntero al final del archivo

seek( MIOFILE, 0, 0 );
Posiciona el puntero al inicio del archivo
```

tell(manejador)

Función que retorna la posición del puntero en el archivo especificado en el parámetro *manejador*. Ejemplo:

```
open(INPUT,"<datos.txt");
read(INPUT, $variable, 30);
$puntero = tell(INPUT);    # $puntero almacena  31;
```

Script que modifica un registro del archivo relativo catalg.rel. Se accede al registro 3 y se modifica el contenido del campo pais de origen

```
open(DAT,"+<catalg.rel");

$ptro_archivo = tell(DAT);
print "Al abrir el puntero de archivo esta en la posicion $ptro_archivo\n";

seek(DAT,100,1);
$ptro_archivo = tell(DAT);
print "Se movio el puntero de archivo al inicio del 3er registro
($ptro_archivo)\n";

read(DAT, $buffer, 50); #Lee el 3er registro y almacena su contenido
                         #en una variable denominada buffer

&parsea_e_imprime();

$pais   = "TURKIA  ";
$buffer = $id_articulo.$articulo.$marca.$pais.$garantia.$stock.$precio;
seek(DAT,100,0);
syswrite(DAT, $buffer, 50); #Regraba el 3er registro
```

```
# Lee nuevamente e imprime el contenido del 3er registro

seek(DAT,100,0);
read(DAT, $buffer, 50);
&parsing_e_imprime();

close(DAT);
```

eof(manejador)

Función que retorna verdadero (1) si el archivo especificado por el parámetro *manejador* tiene el puntero en el final del mismo.

```
open(INPUT,"<datos.txt);
while (!(eof(INPUT))) {
       $linea = <INPUT>;
       print $nombre;
}
```

getc(manejador)

La función getc lee caracteres, de a uno por vez, de un archivo. Ejemplo:

```
$caracter = getc(INFILE);
```

```
#!/usr/local/bin/perl

while (1) {
  $char = getc(STDIN);
  last if ($char eq "\*\");
  $char =~ tr/a-zA-Z0-9/b-zaB-ZA1-90/;
  print ($char);

}

print ("\n");
```

binmode(manejador)

Función que distingue entre archivos de texto y binarios. Normalmente en ambientes unix no hace falta distinguir entre estos tipos de archivos, pero en ambientes Microsoft si.

```
binmode (DAT);
```

syswrite(manejador,variable,largo)

Función que graba sobre un archivo relativo el contenido una variable, de longitud especificada, sobre la posición actual en la que se halle el puntero de archivo.

```
syswrite(DAT, $buffer, 50);
```

Script que añade un registro al final del archivo relativo catalg.rel

```
open(DAT,"+<catalg.rel");


seek(DAT,0,2);
$ptro_archivo = tell(DAT);
print "Se movio el puntero de archivo al final del archivo ($ptro_archivo)\n";
```

```
$buffer="0442Disco Zip 1GB     Iomega  Japon   360130322.00";
syswrite(DAT, $buffer, 50); #Regraba el 3er registro

# Lee e imprime el contenido del registro añadido

seek(DAT,-50,2);
read(DAT, $buffer, 50);
&parsing_imprime();

close(DAT);
```

Miscelaneas Utiles

"Fuera del perro, un libro es probablemente el mejor amigo del hombre, y dentro del perro, probablemente, esta demasiado oscuro para leer". Groucho Marx

Leer parámetros de ejecución de un programa

El arreglo @ARGV almacena los argumentos de la línea de comando. No incluye el nombre con el cual fue ejecutado el programa.

```
>perl leeparametros.pl 33  2parametro -123 "hola mundo"

#!/usr/bin/perl

foreach $parametro (@ARGV) { print "Parámetro: ".$parametro."\n"  }
```

Otras funciones

rand(numero)

Retorna un número al azar entre 0 y *numero*

srand(número)

Establece una semilla para rand

time

Retorna el tiempo en segundos desde el 1/1/1970.

length(cadena)

Retorna el largo en caracteres del string *cadena*

```
$a = "112244";
print  length($a); # imprime 6
```

sleep(n)

Hace que el programa tome una pausa durante *n* segundos

chop(cadena)

Elimina el último caracter del valor de cadena

```
$cadena = "aabcdef";
chop($cadena) ; # $cadena almacena "aabcde";
```

index(expr1, expr2)

Retorna la posición de expr1 en expr2

```
$a = "abcdef";
$b = index($a, "cd");
print  $b;  #  imprime: 2
```

uc(cadena)

Retorna los caracteres de cadena en mayusculas

lc(cadena)

Retorna los caracteres de cadena en minusculas

substr($a, $pos, $len)

extrae un subcadena de la cadena $a
el 2do parámetro es la posición inicial
el 3er parámetro es la longitud de la subcadena a extraer

```
$a = "abcdef";
print  substr ($a, 2, 3);
# imprime: cde
```

Si la función substr se situa al lado izquierdo de una asignación su efecto es de reemplazo de caracteres.

```
$a = "abcdef";
substr ( $a, 2, 3 ) = "xy";    # cambia "cde" por "xy"
print $a                       # imprime: abxyf
```

grep(expresion, arreglo)

Retorna un subarreglo que contiene los elementos del arreglo que satisfacen una determinada condición.

```
@a = ("a1", "a2", "b1", "b2", "c1", "c2");

@b = grep /^b/, @a;    # el patrón /^b/ indica elementos que comienzen
                       # con b
print  "@b";           # imprime: b1 b2
```

sort(arreglo)

Retorna un arreglo ordenado

reverse(arreglo)

Retorna un arreglo invertido

Ejecución de un programa Perl

En la ejecución de un programa Perl, alternativamente, se pueden utilizar modificadores ó switchs. Adicionalmente es posible colocarlos en la primer línea de carga del interprete. (Ej. #!/usr/bin/perl -w)

perl -v # muestra la version del intéprete Perl

perl -V # muestra información acerca de la compilación del intérprete.

```
perl -w  car.pl    # muestra warnings antes de la ejecución del programa car.pl

perl -e '....'     # ejecuta las instrucciones entre comillas
                   # perl -e '$a=78*3.14; print $a."\n"'
```

Trabajando con directorios

Perl dispone de las funciones opendir, readdir y closedir, que permiten leer el contenido de un directorio. Readdir devuelve un arreglo con los nombres de todos los archivos. El siguiente programa busca en el directorio actual el nombre de un archivo determinado:

Ejemplo:

```
#!/usr/local/bin/perl

die "Uso: busca_en_directorio nombre_archivo\n" unless($ARGV[0]);
# se chequea que exista el parámetro nombre de archivo

opendir(MANEJADOR,".") || die "ERROR al leer el directorio actual\n";

foreach (readdir(MANEJADOR)){
       print $_."\n";
       print "\t\t\tSE HA HALLADO $_\n" if (/$ARGV[0]/io);
}

closedir MANEJADOR;
```

En la expresión regular io significa que la búsqueda ignora mayúsculas y minúsculas, y que la expresión regular se compila sólo una vez.

Apéndice: Ejemplo de instrucciones usuales para manejo de vectores de bits

```
$cadena1 = "10101010";
$cadena2 = "11110000";

$vector1 = pack("b*", $cadena1); # $vector es un escalar que contiene a
$vector2 = pack("b*", $cadena2); # cadena1 en formato binario
                                 #(vector de bits)

$and = $vector1 & $vector2;      # Se realiza un and (*)
$or  = $vector1 | $vector2;      # Se realiza un or (+)

# Se mueven los escalares a un arreglo

@resu1 = split(//,unpack("b*",$and)); # al arreglo resu1 se transfiere el
@resu2 = split(//,unpack("b*", $or)); # vector de bits

foreach $ele (@resu1) { print $ele }; print "\n"; # resultado del AND
foreach $ele (@resu2) { print $ele }; print "\n\n"; # resultados del OR

# Con la funcion vec es posible setear un bit

vec($vector1,0,1) = 0;  # se setea a 0 el primer bit (a la izquierda)
vec($vector2,7,1) = 1;  # se setea a 1 el ultimo bit de cadena2

@resu1 = split(//,unpack("b*",$vector1));
@resu2 = split(//,unpack("b*",$vector2));

foreach $ele (@resu1) { print $ele }; print "\n";
foreach $ele (@resu2) { print $ele }; print "\n\n";

# Con la funcion vec es posible testear el estado de un bit particular

  for($i=0;$i<=7;$i++)
  {
      $val1 = vec($vector1, $i, 1); # se testea el bit iesimo
      print $val1."\t"
  }
      print "\n";

# A los efectos de expandir una cadena que contiene un vector de bits
# simplemente se lo setea con vec. La expansion siempre se realiza de
# a byte. Ejemplo> Se tiene un cadena de 8 bits en 1, y con vec se
# setea el bit 10 a 1 [vec($cadena,9,1) = 1] entonces la cadena resultado
# sera: 1111111101000000
```