Curso Basico de Perl

November 14, 2018

Fernando Quintero A.
E-mail:*quintero@mx1.ibm.com*
Notes: *Fernando Quintero/Mexico/IBM*
Domino for AS/400
IBM Global Services
Ext. 7654
Tie Line: 877-7654

1 Introduccion

1.1 Historia de Perl

Perl es la abreviacion de "Practical Extraction and Report Language," su creador es Larry Wall y el creo Perl cuando estaba tratando de producir algunos reportes de archivos, donde awk fallaba en correr. El libero esta version a la comunidad de Usenet y esta fue la primera version que vio la luz del dia.

1.2 Proposito de Perl

Perl ha sido diseñado para asistir al programador con las tareas comunes que probablemente son muy pesadas para un shell o muy simples para un programa en C o algun otro lenguaje compilado.

Una vez que te familiarices con Perl encontraras mas sencillo escribirlo que un shell o una declaracion de C.

Una de las caracteristicas de Perl es que a veces puede ser un lenguaje muy criptico, pero es cuestion de tiempo para escribir programas totalmente legibles.

1.3 Conceptos bàsicos

Un script shell no es nada mas que una secuencia de comandos de shell en un archivo de texto. Cuando el archivo es hecho ejecutable al encender el bit (a través de chmod +x). En forma similar , un programa en Perl es como un monton de lineas y definiciones en un archivo, al encender el bit de ejecuciòn este se hace ejecutable, sin embargo, el archivo tiene que indicar que este no es un shell script, y esto se hace al adicionar la siguiente linea en la primera linea del archivo texto:

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

Perl, es un lenguage de formato libre, como C, pero aun cuando este puede escribirse asi, es necesario establecer cierto orden e indentar adecuadamente el texto con el objeto de hacerlo legible al programador.

Tal como un script shell, un programa de Perl consiste de todas las lineas del archivos tomadas colectivamente como una gran rutina para ejecutar. No existe el concepto de una rutina "main" como en C.

Los comentarios en Perl son como los comentarios de shell (#). No existen comentarios multilineas como en C.

A diferencia de otros shells , el interprete de Perl primero efectua una compilacion interna antes de ejecutar cualquier linea de codigo, esto significa que no obtendras un error de sintaxis una vez que el programa de Perl ha iniciado. Esta misma compilacion, asegura que las operaciones sean rapidas una vez que ha iniciado la ejecucion del programa.

Asi pues , perl es como un compilador y un interprete, lo primero porque primero lee completamente el programa antes de ejecutarlo y lo segundo porque necesita siempre el archivo de texto para ejecutar el programa.

1.4 El programa "Hello World"

Como no podia faltar, necesariamente debe de existir un "hello world" para Perl, y este es esencialmente nuestro primer programa en dicho lenguage, he aqui el mismo:

```
#!/usr/bin/perl -w
# File: pgm1.pl
# Programa 1
print ("Hello, world!\n");
```

La primera linea es una invocacion al interprete de perl, su proposito es indicar al sistema operativo que interprete va a usar para ejecutar el programa que se encuentra en el archivo texto.

Brevemente explicado, la primera linea es leida por el sistema operativo que asi ejecuta al interprete de perl, este, al ser ejecutado continua leyendo el archivo y ejecuta el primer comando: **print** el cual nos es familiar como un programa de C lo haria. Es importante notar que hay ocasiones que nos encontraremos con la misma sentencia sin parentesis, lo cual es igualmente valido, la regla es: los parentesis se pueden omitir en funciones integradas, y esto se deja a criterio del programador.

1.5 Preguntando y recordando el resultado

Necesariamente tenemos que aumentar la complejidad de este ejemplo y ahora modificaremos este programa para que recuerde nuestro nombre, y sea menos frio con nosotros.

Una manera de almacenar valores es a traves de las **variables escalares** nosotros las utilizaremos para tal objeto (posteriormente las veremos con mas detalle).

El programa necesita preguntar por nuestro nombre y almacenar el resultado, saludandonos de vuelta con nuestro nombre. La primera cosa que haremos es imprimir la pregunta y posteriormente leer de la terminal la informacion a traves de <STDIN>, luego asignaremos este a la variable $nombre:

```
#!/usr/bin/perl
# Programa 2
# File: pgm2.pl
print ‘‘cual es tu nombre?’’;
$nombre = <STDIN>;
print ‘‘Hola $nombre, como estas ?\n’’;
```

Digamos que tenemos ahora un saludo especial para “Fernando”, y vamos a verificar si es en realidad tal persona , el programa quedaria de la siguiente manera:

```
#!/usr/bin/perl
# Programa 3
# File pgm3.pl
print ‘‘Cual es tu nombre?’’;                          # Preguntamos por el nombre.
$nombre = <STDIN>;
if ($nombre eq ‘‘Fernando’’) {
       print ‘‘Hola Fernando , gusto en saludarte!\n’’;  # Fernando
} else {
       print ‘‘Hola $nombre, como estas?\n’’;            # Cualquier otro.
}
```

Aqui introducimos el operador **if** y dentro el mismo utilizamos la comparacion **eq** , la cual nos permite preguntar por un valor igual en la cadena que estamos comparando.

1.6 Algunos ejemplos mas

A continuacion , incrementaremos nuestro programa, agregando una palabra secreta, la cual solo sera preguntada si nuestro interlocutor no se llama Fernando.

```
#!/usr/bin/perl -w
# Programa 4
# File pgm4.pl
$secretword = "llama";                          # la palabra secreta
print "cual es tu nombre? ";
$name = <STDIN>;
```

```
chomp $nombre;
if ($name eq "Fernando") {
   print "Hola Fernando! gusto en saludarte!\n";
} else {
   print "Hola, $nombre!\n";                        # Saludo ordinario
   print "Cual es la palabra secreta? ";
   $guess = <STDIN>;
  chomp ($guess);
   while ($guess ne $secretword) {
      print "Error!, trata de nuevo, cual es la palabra secreta? ";
    $guess = <STDIN>;
    chomp ($guess);
   }
}
```

1.7 Ejercicios.

Teclea los programas mostrados en este capitulo y verifica que corran correctamente, no olvides que para hacerlos ejecutables tenemos que ejecutar el comando:

```
chmod a+x <archivo>
```

o ejecutarlo invocando directamente al interprete de perl ej:

```
perl -w <archivo>
```

2 Datos Escalares

2.1 Datos escalares?

Un **escalar** es la forma mas simple de datos que Perl manipula. Este puede ser un numero (como 4 o 3.25e20) o una cadena de caracters (como hola, etc). Las operaciones con escalares siempre llevan a un escalar como resultado.

2.1.1 Tipos de variables

FIZZLE - Un apuntador de archivo o un apuntador de directorio
$FIZZLE - Una variable escalar
@FIZZLE - Un arreglo indexado por numero
%FIZZLE - Un arreglo asociativo (hash)
&FIZZLE - Una subrutina
***FIZZLE** - Todo lo que se llame FIZZLE
Ejemplos de variables:

```
$answer =42; # Entero
$pi = 3.14159265 # numerico
```

```
$avocados = 6.02e23 # notacion cientifica
$mascota = 'perro'; # string
$sign = "Cuidado con el $mascota\n " # String con interpolacion
$curdir = `pwd`; # comando
```

2.2 Numeros

Aun cuando un numero o una cadena de caracteres son escalares ambos, es util ver los numeros y cadenas de caracteres separadamente, numeros primero, luego los caracteres.

2.2.1 Todos los numeros usan el mismo formato internamente.

Se pueden especificar ambos enteros (numeros como 17 o 25 o 342) y numeros de punto flotante (numeros reales con punto decimal com 3.14192 o 1.332). Pero internamente,Perl maneja solamente valores de numeros con doble precision de punto flotante. Esto significa que no existen valores enteros internos.

2.2.2 Literales flotantes

Un literal es la manera en que un valor es respresentado en el texto de un programa de Perl. Tambien se puede llamar a esto una constante en el programa, pero usaremos el termino literal. Los literales son la manera en que los datos son representados en el codigo fuente de tu programa como entrada al compilador de Perl.

Perl acepta el juego completo de literales de punto flotante disponibles a los programadores de C. Se permiten numeros con o sin punto decimal (incluyendo el signo menos o mas), asi como elevando a la potencia de 10 o notacion exponencial. Por ejemplo.

```
1.25        # uno y un cuarto
7.25e45     # 7.25 veces 10 a la 45th potencia
-6.5e24     # negativo 6.5 veces 10 a la 24th
-12e-24     # negativo 12 veces 10 a la  -24th
-1.2E-23    # lo mismo de otra manera
```

2.2.3 Literales enteros

Los literales enteros no tienen la gran explicacion, como por ejemplo:

```
12 15 -200 4 3485
```

No se debe de iniciar el numero con 0 porque Perl soporta literales octal y hexadecimal. Estos se especifican con un cero al principio como por ejemplo 0x o 0X. Los numeros hexadecimales de A ala F representan numeros convencionales de 10 al 15. Por ejemplo:

```
0377    # 377 octal , lo mismo que 255 decimal
-0xff   # FF negativo, lo mismo que -255 decimal.
```

2.3 Cadenas

Las cadenas son secuencias de caracteres (como **hola**). Cada caracter tiene un valor de 8 bits de un set de caracteres de 256, (no hay nada especial en el caracter NULL como en algunos lenguajes).

La cadena mas corta no tiene caracteres. La cadena mas larga puede llenar toda la memoria disponible, esto se basa en el principio de "no limites internos" que perl sigue en cada oportunidad. Las cadenas tipicas son secuencias imprimibles de letras y numeros y puntuaciones en el rango de ASCII 32 y ASCII 126. Sin embargo , la facilidad de tener un caracter de 0 a 255 en una cadena significa que podemos crear, buscar y manipular datos tanto binarios como cadenas - Lo que algunas otras utilerias tendrian dificultad en hacer.

Como los numeros, las cadenas tienen dos tipos: cadenas de comillas simples como ('Hola') y cadenas de comillas dobles (como "Hola").

2.3.1 Cadenas de comillas simples.

Una cadena de comilla simple es una secuencia de caracteres entre comillas simples. Las comillas simples no son parte de la cadena; estas estan para indicarle a Perl el principio y el fin de la cadena. Cualquier caracter entre comillas. Notese que el caracter \n dentro de una cadena de comillas simples no es interpretado como salto de linea, sino como los caracteres backslash y n.

2.3.2 Cadenas de comillas dobles

Una cadena de comilla doble actua como una cadena en C. Esta es una secuencia de caracteres, pero esta vez encerrada en comillas dobles. La diferencia es que en este tipo el backslash tiene una representacion importante al especificar caracteres de control, o aun un caracter octal. He aqui algunos ejemplos:

```
"Hola mundo\n" # Hola Mundo, y un salto de linea
"coca\tsprite" # Coca tabulador y sprite.
```

El backslash puede preceder muchos caracteres diferentes para significar diferentes cosas (tipicamente llamadas secuencias de escape) he aqui una tabla indicando los caracteres:

Construccion	Significado
\n	Salto de linea
\r	Retorno
\t	Tabulador
\f	salto de forma
\b	Backspace
\a	Bell
\e	Escape
\007	Cualquier valor ASCII octal
\x7f	Cualquier valor ASCII hex
\cC	Cualquier carater de control
\\	Backslash
\"	Comillas dobles
\l	Minusculas (el sig. caracter)
\L	Minusculas (todo el texto hasta \E)
\u	Mayusculas (el sig. caracter)
\U	Mayusculas (todo el texto hasta \E)
\Q	Backslash quotas hasta \E

2.4 Operadores escalares.

Un operador produce un nuevo valor (resultado) de uno o mas valores (los operandos). Por ejemplo: + es un operador porque toma dos numeros (los operandos, como 5 y 6) y produce un nuevo valor (11, el resultado).

Los operadores de Perl y expresiones son un superset de los proveidos en otros lenguages tipo ALGOL/Pascal, como C y Java. Un operador espera operadores numericos o de cadena (o posiblemente una combinacion de ambos). Si proporcionas un operador de cadena donde un numero es esperado, o viceversa, Perl automaticamente convierte el operando utilizando algunas reglas intuitivas. Por ejemplo, si dices:

```
$camellos = "123"; print $camellos +1, "\n";
```

El valor original es una cadena , pero para efecto de hacer la suma es convertido automaticamente en un numero, sumado, y entonces convertido de vuelta en cadena de caracteres.

2.4.1 Operadores para numeros.

Perl provee los operadores tipicos de suma, resta, multiplicacion y division, por ejemplo:

```
2+ 3            # 2 mas 3
5.1 - 2.4       # 5.1 menos 2.4
```

Adicionalmente, Perl provee un operador tipo FORTRAN, este operador es el representado por el doble asterisco, como 2**3.

2.4.2 Operadores de comparacion.

Los operadores de comparacion son aquellos utilizados en las pruebas de los dos tipos de escalares, a saber:

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

2.4.3 Sumario de operadores escalares.

Perl tiene muchos operadores, una buena cantidad es derivada del lenguage C. De hecho los unicos operadores que no tiene son los de casting (*), los de direccionamiento (&) y operadores de miembros de estructura (. y ->) he aqui algunos de ellos.

busqueda de patrones

```
$a =~ /pat/ concordancia verdadero si $a contiene un patron.
$a =~ s/p/r  substitucion  Reemplaza ocurrencias de p con r en $a
$a =~ tr/a-z/A-Z/ Traslacion translada los caracteres correspondientes.
```

operadores logicos

```
$a && $b  And  verdadero si $a es verdadero y $b es verdadero
$a || $b  Or $a si $a es verdadero,de otra manera $b
! $a  Not  Verdadero si $a es falso.
```

Operadores aritmeticos.

```
$a + $b  Suma  la suma de $a y $b
$a - $b Resta  la diferencia de $a y $b
$a * $b Multiplicacion el producto de a$ veces $b
$a / $b division el cociente de a$ dividido entre $b
$a % %b modulo  el residuo de $a dividido entre $b
$a ** $b exponente $a elevado a la potencia de $b
++$a,$a++ autoincremento suma 1 a $a
--$a,$a-- autodecremento resta 1 a $a
rand($a) random  un numero aleatorio en el rango de 0..$a
```

Operadores de cadena.

```
$a . $b  concatenacion valores de $a y $b como una sola cadena.
$a x $b  repeticion el valor de $a concatenado $b veces.
substr($a,$o,$l) subcadena subcadena en el offset $o del largo $l
index($a,$b) indice  Offset de la cadena $b en el string $a
```

Operadores de asignacion

```
$a = $b  asignacion $a obtiene el valor de $b
$a += $b suma   incrementa $a por $b
$a -= $b  resta de  decrementa $a por $b
$a .= $b agregacion agrega la cadena $b a $a
```

Operadores de archivo.

```
-r $a  legible  verdadero si el archivo $a es legible
-w $a  escribible verdadero si el archivo $a es escribible
-d $a   directorio verdadero si el archivo $a es un directorio
-f $a   archivo  verdadero si el archivo $a es un archivo
-T $a  archivo texto verdadero si el arcihvo $a es un archivo texto
```

2.4.4 Variables escalares.

Una variable es un contenedor que puede tener uno mas valores, los ejemplos son muy sencillos:

```
$esta_es_una_variable_escalar
$esta_es_otra
```

El limite de tamaño es de 255, que la mayoria de las veces es suficiente.

> tip: trata de utilizar nombres descriptivos para tus variables.

2.4.5 Operadores escalares y funciones

La operacion mas sencilla en una variable escalar es la asignacion, la cual es la manera de dar un valor a una variable. Las operaciones de asignacion de efectuan de la siguiente manera:

```
$a = 17; # a=17
$b = $a + 3; # da a$ el valor de a$ mas 3.
```

Es facil darse cuenta que las variables tiene el operador $ precediendo a la variable, este operador es tambien utilizado en el shell, pero para efecto de asignacion se omite y se usa solo para obtener el valor de la variable, es conveniente entonces no olvidar que en Perl es necesario usarlo siempre.

2.4.6 Operadores de asignacion binaria

Las expresiones tales como $a = a$ + 5 ocurren tan frecuentemente en Perl que existe una via corta, tal como en C, el equivalente seria:

```
a$+=5;
```

asimismo, existen otro tipo de operadores adicionales tales como:

```
a$-=5;              # resta 5
a$*=5;              # multiplica por 5
a$/=5;              # divide entre 5
a$**=5;             # eleva a la potencia de 5
$a.= '' ''         # agrega la cadena '' ''
```

Indicando operadores de resta,multiplicacion y division y otros.

2.4.7 Autoincremento y Autodecremento

Como si no fuera suficientemente facil, existen otros operadores adicionales, los de autoincremento y autodecremento,

los cuales se expresan de la siguiente manera:

```
$a++; # Incrementa $a en 1
$a--; # Decrementa $a en 1
```

El autoincremento funciona aun en cadenas.

2.4.8 Las funciones chop y chomp.

Una funcion integrada muy util es chop. Esta funcion toma un solo argumento dentro de sus parentesis, el nombre de la variable escalar, y remueve el ultimo caracter de la cadena. Por ejemplo:

```
$x = ''Hola Mundo''; chop($); # $x es ahora ''Hola Mund''
```

si le damos un valor vacio , entonces no regresa nada ni da error.

Esta funcion es util para remover el caracter de salto de linea '\n' , pero corremos el riesgo de remover por error un caracter que es util, en esos casos utilizaremos la funcion chomp(), la cual solo remueve el caracter de salto de linea.

2.4.9 Interpolacion de escalares en cadenas.

Cuando una cadena tiene comillas dobles, es sujeta a interpolacion de variables. Esto significa que la cadena es escaneada por posibles valores de variables escalares, un signo de dolar seguido por letras, digitos. Cuando una variable referenciada es encontrada, esta es reemplazada por su valor actual. ej.

```
$a= ''fred'';
$b =''texto en $a'' # el valor interpolado
```

2.5 <STDIN> como un valor escalar.

Hasta este punto, si eres un programador experimentado, probablemente te preguntaras como obtener un valor en un programa de Perl. He aqui la manera mas simple. Cada vez que se usa <STDIN> en un lugar donde un escalar debe de estar, Perl lee la siguiente entrada estandar (hasta el primer salto de linea), y usa ese valor como <STDIN>.

<STDIN> significa muchas cosas, pero a menos que se haga algo extraño regularmente significara una entrada de la terminal. Si no existe algo esperando a ser leido (en el buffer) , Perl espera a que teclees algo, hasta que presiones la tecla de entrada. El valor de <STDIN> regularmente trae un salto de linea al final del mismo, y regularmente querremos deshacernos de el , y para eso utilizaremos la funcion chomp(), como en el siguiente ejemplo:

```
$a = <STDIN>; # Obtener el texto
chomp($a);    # Nos deshacemos del caracter '\n'
```

Eso hara el trabajo por nosotros.

2.6 Salida con print()

Asi que, leemos cosas con <STDIN>. Como imprimimos?, con la funcion print. Esta funcion toma los valores dentro de sus parentesis y las pone sin embellecimiento a la salida estandar. De nuevo, a menos que hagas algo extraño esta sera tu terminal. Por ejemplo:

```
print(“Hola Mundo\n”); # Decimos hola al mundo , seguido por un caracter de salto de li
print “Hola Mundo\n” ; # Lo mismo.
```

Notemos que el segundo ejemplo imprime lo mismo sin parentesis, Perl permite la no especificacion de parentesis para sus funciones internas , pero por regla general es necesario pare efectos de evitar ambiguedades.

2.7 El valor indefinido.

¿Que pasa si usas una variable escalar antes de darle un valor?. Nada serio, y definitivamente, nada fatal. Las variables tiene un valor de undef antes de ser asignadas. Este valor es como el cero cuando se usa como un numero o una cadena vacia cuando se usa como cadena. Si utilizamos la invocacion perl -w sin embargo obtendremos un warning, lo cual es una buena manera de atrapar los errores.

Muchos operadores regresan undef cuando los argumentos estan fuera de rango o no hacen sentido.

2.8 Ejercicios

1. Escriba un programa que calcule la circunferencia de un circulo con radio de 12.5. La circunferencia es 2 veces el radio, o cerca de 2 veces 3.141592654 veces el radio.

2. Modifique el programa del ejercicio anterior para preguntar y aceptar el radio de la persona que corre el programa.

3. Escriba un programa que pregunte y lea dos numeros, e imprima el resultado de dos numeros multiplicados entre si.

4. Escriba un programa que lea una cadena y un numero, e imprima la cadena el numero de veces indicado por el numero de lineas separadas. (tip: utilice el operador "x").

3 Arreglos

3.1 ¿Que es una lista o arreglo?

Una lista son datos escalares ordenados. Un arreglo es una variable que contiene una lista. Cada elemento del arreglo es una variable escalar separada con un valor escalar independiente. Estos valores estan ordenados, es decir, tienen una secuencia particular de menor a mayor.

Los arreglos pueden tener cualquier numero de elementos. El arreglo mas pequeño no tiene elementos, mientras que el arreglo mas grande puede llenar completamente la memoria, nuevamente siguiendo la filosofia de Perl de "no limites innecesarios".

Perl tiene muchos operadores que hacen cosas con listas. Mucho del poder de perl viene de la capacidad de procesar multiples items con un solo comando, y muchos de los problemas en el mundo real son solucionados naturalmente con una lista.

Una lista es un set ordenado de escalares. Los cuales ya sabemos que son no?.

La lista puede ser una lista de numeros, cadenas, o una mezcla de ambos. Algunas listas son con nombre y las llamaremos "arreglos". Otras listas son simplemente valores que son pasados de una operacion a otra y las llamaremos "valores de arreglos" o "listas", dado que los valores en una lista estan ordenados, hace sentido hablar acerca del primero , segundo y hasta el ultimo, etc. Un ejemplo de estos serian los argumentos de la linea de comandos que son copiados en el arreglo @ARGV de tal manera que:

$ARGV[0] es el primer argumento, y: $ARGV[$#ARGV] es el ultimo.

La construccion de listas en perl es simplemente una coma, usada entre cada elemento de la lista.

Un ejemplo de una lista seria la siguiente:

```
'primero','segundo','tercero', 'cuarto'
o: 1,2,3,4
```

Uno de los lugares en donde Perl busca por una lista es dentro de los parentesis, de tal manera que podriamos utilizar una lista para inicializar un arreglo entero de la siguiente manera:

```
@colores = ($rojo,$verde,$azul); @cuenta = (1,2,3,4,5,6);
```

Una lista por parentesis que esta compuesta enteramente por nombres puede ser asignada al valor de cualquier cosa que regrese un valor de lista.

```
($rojo,$verde,$azul) = (0..2); ($nombre,$pw,$gid,$gcos,$home,$shell) = split(/:/,<PASSW
```

El operador .. regresa una lista de numeros de 1 a 2.

3.2 Operaciones con arreglos.

3.2.1 Acceso a los elementos del arreglo.

Hasta ahora hemos tratado a los arreglos como un todo, agregando y removiendo valores al hacer asignaciones de arreglos. Muchos programas utiles se escriben usando arreglos sin siquiera utilizar un elemento especifico del arreglo. Sin embargo, perl provee una funcion de subscripcion tradicional para accesar un elemento del arreglo por indice numerico.

Estos son numerados utilizando enteros secuenciales, empezando en cero e incrementando en uno para cada elemento. El primer elemento del arreglo @arreglo es accesado como $arreglo[0]. Notese que el @ en el arreglo se convierte en un $ en el elemento de referencia, esto se hace por razones obvias.

Ejemplos:

```
@fred = (7,8,9);
$b=$fred[0]; # da a $b el valor de 7
$fred[0]=5; # ahora fred es (5,8,9)
```

3.2.2 Las funciones push y pop

Un uso comun en un arreglo es como una pila de informacion, donde nuevos valores se agregan y remueven del lado derecho de la lista. Estas operaciones ocurren con suficiente frecuencia para tener sus propias funciones especiales:

```
push(@lista,$valor) # Inserta un nuevo valor en mi listas.
pop(@lista); # Remueve el ultimo elemento de mi lista
```

3.2.3 Las funciones shift y unshift

Las funciones push y pop hacen cosas con el lado derecho de una lista, similarmente, las funciones shift y unshift ejecutan las funciones correspondientes en el lado izquierdo de una lista.

He aqui algunos ejemplos:

```
unshift(@lista,$a);     # como @lista=($a,@fred) inserta un nuevo valor del lado izquie
x$ = shift(@lista)      # remueve el primer elemento de la lista y la inserta en $x
```

3.2.4 La funcion reverse.

La funcion reverse reversa el orden de los elementos de su argumento, regresando la lista resultante. Por ejemplo:

```
@a=(7,8,9);             # el valor de @a
@b=reverse(@a);         # da a @b el valor (9,8,7).
```

3.2.5 La funcion sort

La funcion sort toma los argumentos y los ordena como si fueran simples cadenas en orden ascendente ASCII. Regresa la lista ordenada sin alterar el valor original de la lista. Por ejemplo:

```
@x = sort(‘‘small’’,’’medium’’,’’large’’      # @x obtiene ‘‘large’’,’’medium’’,’’small’’
```

El ordenamiento de numeros no pasa numericamente, pero por valores de cadenas de cada numero (1,16,2,32 etc).

3.2.6 La funcion chomp.

La funcion chomp funciona con una variable de arreglo asi como sobre un valor escalar. Cada separador de registro de cada elemento es removido. por ejemplo:

```
@stuff=(‘‘Hola\n’’,’’mundo\n’’,’’dias felices\n’’);
chomp(@stuff); # @stuff es ahora (‘‘Hola’’,’’mundo’’,’’dias felices’’);
```

3.3 Ejercicios

1. Escriba un programa que lea una lista de cadenas de caracteres en lineas separadas y escriba la lista al reves. Si se esta leyendo la lista de la terminal, probablemente necesitas delimitar el final de la lista presionando el caracter de fin de archivo, probablemente CTRL-D bajo UNIX, o CTRL-Z en otros sistemas operativos.

2. Escriba un programa que lea un numero y entonces una lista de caracteres (todos en lineas separadas), y las imprima de acuerdo a una seleccion hecha por numero.

3. Escriba un programa que lea una lista de cadenas de caracteres y seleccione e imprima una cadena aleatoria de la lista. Para seleccionar un elemento aleatorio de @arreglo, utilice srand; al principio del programa (esto inicializa al generador de numeros aleatorios) , y entonces utilice userand(@arreglo), donde se necesita un valor aleatorio entre cero y uno menor que la longitud de @arreglo.

4 Estructuras de control

4.1 Bloques de sentencias.

Un bloque de sentencia es una secuencia de sentencias, encerrada entre braces. Algo parecido a esto:

```
{
primera_sentencia;
segunda_sentencia;
tercera_sentencia;
.... ultima_sentencia
}
```

Perl ejecuta cada sentencia en secuencia, desde el principio hasta el fin. El semicolon final es opcional.

4.2 La sentencia if/unless

Lo siguiente en complejidad es la sentencia if. Esta construccion toma una expresion de control (evaluada por su resultado verdadero) y un bloque. Opcionalmente puede tener un else seguido por un bloque tambien. En otras palabras, se parece a esto:

```
if (expresion)
{
sentencia_verdadera_1;
sentencia_verdadera_2;
sentencia_verdadera_3;
} else
{
sentencia_falsa_1;
sentencia_falsa_2
}
```

Durante la ejecucion , Perl evalua la expresion de control. Si la expresion es verdadera, se ejecuta el primer bloque de control, si la expresion es falsa, el segundo bloque.

La evaluacion de no se hace a nivel boleano sino a nivel de cadena, de tal manera que si la cadena resultante esta vacia o consiste del caracter 0, entonces el valor de la expresion es falso, y cualquier cosa que no sea asi resulta verdadero automaticamente. He aqui algunos ejemplo de interpretacion:

```
0          # convierte a ‘‘0’’, asi que falso
1-1        # convierte a 0 y a ‘‘0’’ asi que falso
1          # convierte a ‘‘1’’, verdadero
‘‘’’       # cadena vacia, falso
‘‘1’’      # no ‘‘’’ o ‘‘0’’ asi que verdadero
‘‘00’’     # no ‘‘’’ o ‘‘0’’ , verdadero
‘‘0.000’’  # no ‘‘’’ o ‘‘0’’ , verdadero
```

Aqui tenemos un ejemplo de una linea completa:

```
#!/usr/bin/perl
# Programa de ejemplo
# ej.pl
print ‘‘Cual es tu edad?’’;
$a = <STDIN>;
chomp a$;
if ($a < 18)
{
print ‘‘No tienes edad para votar\n’’;
}
```

4.3 La sentencia while/until

No existe lenguage que este complete con alguna forma de iteracion. Perl puede iterar utilizando la sentenci while.

```
while (expresion)
{
sentencia_1;
sentencia_2;
}
```

Para ejecutar esta sentencia, Perl evalua la expresion de control, mientras su valor sea verdadero, el cuerpo del while es evaluado una vez. Esto se repite hasta que la expresion de control se vuelva falsa, hecho lo cual Perl continua con la sentencia siguiente despues del ciclo while. Por ejemplo:

```
#!/usr/bin/perl
# Programa while de ejemplo.
print ‘‘Que edad tienes?’’;
$a = <STDIN>;
chomp($a);
while($a > 0)
{
print ‘‘Una vez , tuviste $a años\n’’;
$a--;
}
```

Algunas veces es mas facil decir “Hasta que algo sea verdadero” en lugar de “mientras no sea verdadero”. Una vez mas podemos hacerlo. Al reemplazar la sentencia con until obtenemos el resultado deseado:

```
until ( expresion)
{
sentencia_1;
sentencia_2;
}
```

Algunas veces el loop nunca llegara a ser verdadero o falso y podria depender de los resultados que querramos, este podria ser algun demonio que deseemos correr para siempre.

4.3.1 La sentencia do {} while/until

La sentencia while/until es una condicion al principio de cada ciclo, antes de entrar al ciclo. Si la condicion fue falsa antes de ser verificado, el ciclo no sera ejecutado nunca

Pero algunas veces no queremos probar la condicion al principio del ciclo. En lugar de eso, queremos probarlo al final. Perl provee el ciclo do {} while para este efecto. Este seria:

```
do {
sentencia_1;
sentencia_2;
} while expresion
```

De esta manera, Perl ejecuta al menos una vez las sentencias antes de probar la vericidad de la condicion.

Como en las sentencias anteriores se puede invertir la condicion utilizando do {} until en vez de do {} while. La expresion es aun probada al final, pero su sentido se invierte. Ejemplo:

```
#!/usr/bin/perl
# Programa de do/until
$alto=0;
do {
$alto++;
print ‘‘Siguiente parada\n’’;
chomp($sitio = <STDIN>);
}
until $alto > 5 || $location eq ’home’;
```

4.4 La sentencia for

Otra iteracion es la sentencia for, la cual trabaja similar a C o Java.

```
for ( expresion_inicial; prueba_expresion; reini_expresion)
{
sentencia_1;
sentencia_2;
sentencia_3;
}
```

A continuacion un ejemplo:

```
for($i=1;$i<=10;$i++)
{
print ‘‘$i ‘‘;
}
```

Inicialmente la variable se pone a 1, despues se compara con 10 y finalmente se incrementa, sucesivamente hasta llegar a 10.

4.5 la sentencia foreach

Otra construccion de interacion es la sentencia foreach. Esta sentencia toma una lista de valores y las asigna una a la vez, a un valor escalar, ejecutando un bloque de codigo para cada iteracion. Luce asi:

```
foreach $i (@lista)
{
sentencia_1;
sentencia_2;
}
```

Un ejemplo practico seria:

```
@a = (1,2,3,4,5);
foreach $b (reverse @a)
{
print @b;
}
```

El resultado del programa es 54321.

4.6 Ejercicios

1. Escriba un programa que pregunte por la temperatura exterior, e imprima “demasiado caliente” si la temperatura esta arriba de 30 grados, y “muy frio” de otra manera.

2. Modifique el programa del ejercicion anterior de tal manera que imprima “muy caliente” si la temperatura esta arriba de 32 grados, “muy frio” si la temperatura esta abajo de 17 grados y “justo!” si esta entre 17 y 32.

3. Escriba un programa que lea una lista de numeros (en lineas separadas) hasta que se lea el numero 999, y entonces que imprima el total de numeros sumados. (sin sumar el 999!). Por ejemplo , si se teclea 1,2,3 y 999 el programa debe de regresar con la respuesta de 6 (1+2+3).

4. Escriba un programa que lea una lista de cadenas en lineas separadas e imprima la lista de cadenas en orden invertido - sin utilizar la funcion reverse(). (recuerde que <STDIN> va a leer una lista de cadenas en lineas separadas cuando se utiliza en un contexto de arreglo.

5. Escriba un programa que imprima una tabla de numeros y sus raices cuadradas de cero a 32. Busque una manera de no tener todos los numeros del 0 al 32 en una lista, y entonces trate de hacerlo tambien asi. (tip: para que se vea bien: printf "%5g %8g\n", a$,b$, imprime a$ como un numero de 5 columnas y b$ como uno de 8.)

5 Arreglos asociativos

5.1 Que es un hash? (arreglo asociativo).

El termino "hash" (o arreglo asociativo), es el de una coleccion de datos escalares, con elementos individuales seleccionados por algun valor de indice. Al contrario de una lista, los valores del indice de un hash son enteros no negativos, aunque arbitrarios. Esos escalares (llamados llaves) se usan para obtener los valores del arreglo.

Los elementos de un hash no tienen un orden en particular. Consideremoslosen lugar de un mazo de cartas. La parte media alta de cada carta es la llave, y la parte media inferior es el valor. Cada vez que se pone un valor en el hash, una nueva carta se crea. Mas tarde, cuando se quiera modificar el valor, le das la llave y Perl encuentra la carta correcta. Asi que el orden de las cartas es inmaterial. De hecho , Perl almacena las cartas (la pareja de valores clave) en un orden interno especial que hace facil encontrar una carta especifica, de manera que perl no tenga que ver a traves de todos los pares para encontrar el correcto, y no se puede controlar el orden.

5.2 Variables hash

Una variable "hash" es un signo de porcentaje (%) seguido por una letra, seguida por cero o mas letras,digitos y underscores. En otras palabras, la parte despues de porcentaje es como lo que temiamos para nombres de variables y arreglos escalares.

Cada elemento del hash es una variable escalara , accesada por un indice de cadena, llamada llave.

Como con los arreglos , se crean nuevos elementos simplemente asignandolos a un elemento hash:

```
$fred{‘‘aaa’’} = ‘‘bbb’’;   # creamos la llave ‘‘aaa’’, valor ‘‘bbb’’
$fred{234.5} = 456.7;       # crea la llave ‘‘234.5’’, valor 456.7
```

Estas lineas crean dos elementos en el hash. Accesos subsecuentes al mismo elemento (usando la misma llave) regresa el valor previamente almacenado.

```
print $fred{‘‘aaa’’} # imprime ‘‘bbb’’
$fred{234.5} +=3; # genera el valor 459.7
```

El referenciar a un elemento no existente regresa un valor indefinido.

5.3 Representacion literal de un hash .

Se puede desear accesar el hash como un total, ya sea para inicializarlo o copiarlo en otro hash. Perl no tiene realmente una representacion literal para un hash. Cada para de elementos en la lista define una llave y su correspondiente valor. En otras palabras:

```
@pedro_list = %fred;                    # @pedro_list es asignado a ("aaa","bbb","234.
%pablo = @pedro_list;                   # crea %pablo de %pedro
%pablo = %pedro;                        # Lo mismo mas rapido
```

%suave = ("aaa","bbb","234.5",456.7); # crea %suave como %fred, de valores literales.

5.4 Funciones Hash

Esta seccion lista algunas funciones hash.

5.4.1 La funcion *keys*

La funcion **keys(%nombrehash)** regresa una lista de todas las llaves actuales en el hash %nombrehash. Si no existen elementos en el hash, entonces keys() regresa una lista vacia.

Ejemplo:

```
$pedro{"aaa"} = "bbb";
$pedro{234.5} = 456.7;
@lista = keys(%pedro); # @lista obtiene ("aaa",234.5) o # (234.5,"aaa")
```

Como en otras funciones, el parentesis es opcional: keys %pedro es como keys(%pedro).

```
foreach $llave (keys (%pedro))
{                                               # uno por cada llave de %pedro
print "en $llave tenemos $pedro{$llave}\n";     # mostrar llave y valores
}
```

Este ejemplo muestra que elementos individuales pueden ser interpolados en cadenas de dobles comillas.

5.4.2 La funcion *Values.*

La funcion **values(%nombrehash)** regresa una lista de todos los valores actuales de %nombrehash, en el mismo orden como se regresa por la funcion keys(%nombrehash).

Por ejemplo:

```
%apellido=();                    # forzamos %apellido a estar vacio
%apellido{‘‘pedro’’} = ‘‘picapiedra’’;
%apellido{‘‘pablo’’} = ‘‘marmol’’;
@apellidos = values(%apellido); # obtenemos los valores
```

En este punto **@apellidos** contiene ya sea (‘‘picapiedra,’’marmol’’) o (‘‘marmol’’,’’picapiedra’’).

5.4.3 La funcion *each*

Para iterar sobre un hash entero, utilice keys, buscando por cada llave que regrese para obtener el valor correcto. A pesar de que este metodo es usado frecuentemente, un metodo mas eficiente es utilizar **each(%nombrehash)**, el cual regresa un valor de llave como una lista de dos elementos. En cada evaluacion de esta funcion para el mismo hash, el siguiente valor de par sucesivo es regresado hasta que todos los elementos hayan sido accesados, cuando no existen mas pares *each* regresa una lista vacia.

Ejemplo: para pasar a traves del hash %apellido del ejemplo anterior , hacemos algo como :

```
while (($nombre,$ultimo) = each(%apellido))
{
    print ‘‘El apellido de $nombre es $ultimo\n’’;
}
```

5.4.4 La funcion *delete*

Hasta ahora lo que sabemos es que podemos agregar elementos a un hash , pero no podemos removerlos (de otra manera que asignando un valor nuevo al hash entero) . Perl provee la funcion *delete* para remover los elementos del hash. El operando delete una referencia a hash, como si estuvieramos buscando por un valor en particular. Perl remueve el par llave-valor del hash. Ejemplo:

```
%pedro = ("aaa","bbb",234.5,34.56);  # damos a %pedro dos elementos.
delete $pedro(‘‘aaa’’);              # %pedro tiene ahora un par llave-valor
```

5.5 Dividiendo un hash

Como una variable tipo array , un hash puede ser dividido para accesar una coleccion de elementos en lugar de solamente un elemento al a vez. Por ejemplo:

```
$puntos{‘‘pedro’’} = 205;
$puntos{‘‘pablo’’} = 195;
$puntos{‘‘dino’’} = 30;
```

Podria ser reducido a:

```
($puntos{‘‘pedro’’,’’pablo’’,’’dino’’})=(205,195,30);
```

Lo cual es mucho mas corto. Podemos usar un hash dividido con interpolacion de variables tambien:

```
@jugadores= qw(pedro pablo dino);
print ‘‘Los puntos son: @puntos{@jugadores}\n’’;
```

5.6 Ejercicios.

1. Escriba un programa que lea una serie de palabras con una palabra por linea hasta el fin de archivo, entonces, que imprima un sumario de cuantas veces cada palabra se ha visto. (para un mayor grado de dificultad , ordenar las palabras en orden ASCII ascendente en la salida.

2. Escriba un programa que lea en una cadena, que imprima la cadena y su valor mapeado de acuerdo a los valores presentados en la siguiente tabla:

Entrada	**Salida**
rojo	manzana
verde	hojas
azul	oceano

6 E/S Basica

6.1 Entrada de *STDIN*

Leer de la entrada estandar (a traves del filehandel llamado STDIN) es facil. Lo hemos estado haciendo ya con la operacion <STDIN>. Evaluandolo en un contexto escalar hariamos lo siguiente:

```
$a=<STDIN>                  #leemos la siguiente linea
```

Hasta un caracter de salto de linea, o a lo que hayamos puesto $/.

Evaluandolo en un contexto de lista produce todas las lineas que faltan como una lista: cada elemento es una linea, incluyendo su caracter de terminacion de linea. Esto ya lo vimos , pero que sirva de repaso. Lo representariamos de la siguiente manera:

```
@a=<STDIN>;
```

Tipicamente, una cosa que queremos hacer es leer todas las lineas una a la vez y hacer algo con cada linea. Una manera comun de hacerl es:

```
while (defined($linea=<STDIN>))
{
     # procesamos $linea aqui
}
```

Mientras que se lea una linea , <STDIN> evalua a un valor definido, y el ciclo continua. Cuando <STDIN> no tiene mas lineas que evaluar regresa *undef*, terminando el ciclo.

6.2 Entrada del operador de diamante <>

Otra manera de leer es con el operador de diamante: <>. Este trabaja como <STDIN> en el sentido de que regresa una sola linea en un contexto escalar (undef si todas las lineas han sido leidas) o todas las lineas restantes si se utiliza en un contexto de lista. Sin embargo, a diferencia de <STDIN>, el operador de diamente obtiene sus datos de el archvo o archivo especificados en la linea de comandos que invoco al programa de Perl. Por ejemplo:

```
#!/usr/bin/perl
while(<>)
{
   print $_;
}
```

Este programa es invocado como: *pgm1 archivo1 archivo2 archivo3*

Cuando el operador de diamante lee cada linea de archivo1 seguido de cada linea de archivo2 y archivo3, regresa undef solamente cuando todas las lineas han sido leidas.

Tecnicamente, el operador de diamante no ve literalmente los argumentos de la linea de comandos; trabaja a traves del arreglo @ARGV. Este es un arreglo especial inicializadopor el interprete de perl con los argumentos de la linea de comandos. Cada argumendo de linea de comandoj va en un elemento separado del arreglo @ARGV.

Podemos inclusive modificar el funcionamiento del operador de diamante utilizandolo como en el siguiente ejemplo:

```
@ARGV= (‘‘aaa’’,’’bbb’’,’’ccc’’);
while(<>)
{
 # procesa archivos aaa, bbb y ccc
 print ‘‘esta linea es:$_’’;

}
```

6.3 Salida a STDOUT

Perl utiliza las funciones print y printf para escribir a salida estandar. Vamos a ver como se usa.

6.3.1 Usando *print* para salida normal.

Hemos usado ya print para mostrar texto en salida estandar. Vamos a explicar un poco eso.

La funcion print toma una lista de cadenas y envia cada cadena a la salida estandar en turno , sin ninguna intervencion o caracteres adicionados. Lo que puede no ser obvio es que print es realmente una funcion que toma una lista de argumentos, y regresa un valor como cualquier funcion. En otras palabras,

$a = print"Hola","mundo","\n");

Seria otra manera de decir hola mundo. El valor de regreso es verdadero o falso, indicando el exito o falla de la funcion.

6.3.2 Utilizando *printf* para salida formateada.

Si quisieramos algo mas de control sobre la salida que print provee, utilizariamos esta funcion de la misma manera que en el lenguage "C". La funcion *printf* toma una lista de argumentos, el primer argumento es el formato de control, definiendo como imprimir los argumentos restantes. Ejemplo:

```
printf ‘‘%15s %5d %10.2f\n’’,$s,$n,$r’’;
```

Imprime $s en un campo de 15 caracteres , un espacio, entonces $n como un entero decimal en un campo de 5 caracteres, y otro espacio, finalmente $r como un valor de punto flotante con 2 lugares decimales en un campo de 10 caracteres y finalmente un caracter de salto delinea.

6.4 Ejercicios

1. Escriba un programa que actue como el comando cat, pero invierta el orden de las lineas de todos los archivos especificados en la linea de comando o de todas las lineas de entrada estandar si no se especifican archivos.

2. Modificar el programa del ejercicio previo, de manera que cada archivo especificado en la linea de comando tiene sus lineas individualmente invertidos.

3. Escribir un programa que lea una lista de cadena en archivos separados e imprima las cadenas en una columna de 20 caracteres justificada a la derecha. Por ejemplo introduciendo hola, adios imprime hola y adios justificado a la derecha en una columna de 20 caracteres.

4. Modifica el programa del ejercicio previo para permitirle al usuario seleccionar el ancho de la coluna. Por ejemplo, introduciendo 20, hola y adios deberia hacer lo mismo que el pgm anterior, pero introduciendo 30, hola y adios deberia de justificar hola y adios en una columna de 30 caracteres.

7 Expresiones regulares.

7.1 Conceptos acerca de expresiones regulares.

Una expresion regular es un patron - un template - a concordar contra una cadena. La concordancia de una expresion regular contra una cadena puede ser exitosa o fallar. Algunas veces el exito o falla es algo que nos interesa. Otras veces, desearemos tomar un patron que concuerda y reemplazarlo con otra cadena , parte del cual depende sobre como y donde exactamente concuerda la expresion.

Las expresiones regulares son utilizadas por muchos programs, como los comandos de unix grep, sed , awk, edi, vi, emacs e incluso varios shells. Cada programa tiene un juego diferente de caracteres de template. Perl es un superset semantico de todas esas herramientas: cualquier expresion regular que pueda ser descrita en una de esas herramientas puede tambien ser escritas en Perl, pero no necesariamente usando los mismos caracteres.

7.2 Usos simples de expresiones regulares.

Si estuvieramos buscando por todas las lineas de una archivo conteniendo la cadena de caracteres abc , pudriamos usar el comando de grep:

```
grep abc archivo > resultado
```

En este caso, abc es la expresion regular que el comando de grep prueba contra cada linea de entrada. Las lineas que concuerdan son enviadas a la salida estandar.

En perl, podemos hablar de la cadena abc como una expresion regular encerrando la cadena en diagonales:

```
if (/abc/)
{
print $_;
}
```

Pero que es lo que esta siendo probado contra la expresion regular "abc" ?, es nuestro viejo amigo, la variable $_. Cuando una expresion regular se encierra entre diagonales como arriba, la variable $_ es probada contra la expresion regular. Si esta concuerda, el operador de concordancia regresa verdadero. De otra manera regresa falso.

Para este ejemplo, la variable $_ se presume contiene alguna linea de texto y esta es impresa si la linea contiene los caracters "abc" en secuencia en cualquier parte de la linea - similar al comando grep arriba mencionado - A diferencia del comando grep que opera sobre todas las lineas, esta opera sobre una linea. Para trabajar sobre todas la lineas agregamos un ciclo:

```
while (<>)
{
  if (/abc/)
```

```
    {
      print $_;
    }
}
```

Que tal si no sabemos el numero de b's entre la a y la c?, Si queremos imprimir la linea que contenga una a seguida por cero o mas b's, seguidas por una c. Con grep diriamos:

```
grep ‘‘ab*c’’ archivo > resultado.
```

En perl seria:

```
while (<>)
{
if (/ab*c/)
{
print $_;
}
}
```

Exactamente igual que grep.

Otra expresion regular simple es el *operador de substitucion,* el cual reemplaza la parte de una cadena que concuerde con una expresion regular con otra cadena. El operador de substitucion se parece al comando s en la utileria de UNIX sed, consistiendo de la letra s, de una diagonal , una expresion regular, una diagonal y una cadena de reemplazo, y una diagnal final tal como se muestra abajo:

```
s/ab*c/def/;
```

La variable, ($_ en este caso) es concordada contra la expresion regular (ab*c). Si la concordancia es exitosa, la parte del string que concuerda es reemplazado por la cadena def, si no , nada pasa.

7.3 Patrones.

Una expresion regular es un patron. Algunas partes del patron concuerdan con caracteres simples en la cadena de un tipo particular. Otras partes del patron concuerdan con caracteres multiples. Primero, visitaremos los patrones de caracteres simples y entonces los patrones de caracteres multiples.

7.3.1 Patrones de caracteres simples

El caracter mas comun y simple de concordancia de patrones en expresiones regulares es un simple caracter que concuerda con si mismo. En otras palabras, poner una letra en una expresion regular requiere de una letra correspondiente en una cadena.

El siguiente caracter mas comun de concordancia es el punto “.”. Este concuerda con cualquier caracter simple excepto el caracter de salto de linea (\n). Por ejemplo,

el patron /a./ concuerda con cualquier secuencia de dos letras que empiece con a y que no sea "a\n".

Un caracter de patron es representado por un par de corchetes y una lista de caracteres entre los mismos. Uno y solamente uno de esos caracteres debe de estar presentes en la parte correspondiente de la cadena para que el patron concuerde. Por ejemplo:

```
/[abcde]/
```

concuerda una cadena conteniendo cualquiera de las cinco letras del alfabeto en minusculas, mientras

```
/[aeiouAEIOU]/
```

concuerda con cualquiera de las cinco vocales en minusculas o mayusculas. Si quieres utilizar un caracter de corchete derecho (]) en la lista , deberas de poner una diagonal invertida enfrente de el, o ponlo como el primer caracter dentro de la lista. Los rangos de caracteres (como de la a a la z) pueden ser abreviados mostrando los caracteres separados por un guion; para poner un guion en la lista se debera de preceder al mismo con una diagonal invertida o colocarlo al final. he aqui algunos ejemplos:

```
[0123456789]    # concuerda cualquier digito
[0-9]           # lo mismo
[0-9\-]         # concuerda 0-9,o menos
[a-z0-9]        # concuerda cualquier letra minuscula o digito
[a-zA-Z0-9_]    # concuerda cualquier letra , digito o subguion
```

Existe tambien una clase de caracter negado, el cual es el mismo como clase de caracter, pero tiene una flecha hacia arriba (^) inmediatamente despues del corchete izquierdo. Este caracter concuerda cualquier caracter que no este en la lista. Por ejemplo:

```
[^0-9]           # concuerda cualquier cosa que no sea un digito.
[^aeiouAEIOU]    # concuerda cualquier cosa que no sea vocal.
[^\^]            # concuerda cualquier cosa que no sea (^).
```

Para facilidad , algunas clases de caracteres comunes estan predefinidos, como se describen en la tabla:

Construccion	Clase equivalente	Construccion negada	Clase equivalente negada
\d(un digito)	[0-9]	\D(no digitos!)	[^0-9]
\w(caracter alfabetico)	[a-zA-A0-9]	\W(no caracter alfabetico!)	[^a-zA-Z0-9]
\s(espacio)	[\r\t\n\f]	\S(no espacios)	[^ \r\t\n\f]

Esas clases abreviadas pueden ser parte de otro caracter tambien:

```
[\da-fA-F] # match one hex digit
```

7.3.2 Patrones de agrupamiento.

El verdadero poder de las expresiones regulares entra en juego cuando decimos "uno o mas de esos" o "hasta cinco de esos". Entremos en detalle.

Secuencia El primer (y probablemente menos obvio) patron de agrupamiento es la secuencia. Esto significa que abc concuerda con una a seguida por una b y una c. Parece simple, pero estamos nombrandolo de manera que podamos hablar de eso mas tarde.

Multiplicadores Ya hemos visto el asterisco (*) como un patron de agrupamiento. El asterisco indica cero o mas de los caracteres previos (o caracteres clase).

Dos patrones de agrupamiento que trabajan como este son el signo de mas (+), significando uno o mas del caracter previo inmediato y el signo de interrogacion (?), significando cero o mas del caracter previo inmediato. Por ejemplo, la expresion regular /fo+ba?r/ concuerda una f seguida de uno o mas de o's seguidos por una b, seguida por una a opcional, seguida por una r.

En todos los tres grupos de patrones estos se caracterizan por ser ambiciosos. Si los multiplicadores tuvieran oportunidad de concordar entre cinco y diez caracteres, escogeran la cadena de 10 caracteres siempre. Por ejemplo:

```
$_ = ‘‘pedro xxxxxxxxxx pablo’’;
s/x+/boom/;
```

Siempre reemplaza todas las x's consecutivas con boom (resultando en pedro boom pablo), aun cuando un set mas corto de x's tambien concordaria la misma expresion regular.

si necesitaras decir "cinco a diez" x's , se puede poner cinco x's seguidas por cinco x's seguidas inmediatamente por un caracter de interrogacion. Pero se veria feo. En lugar de eso hay una manera mas facil; El multiplicador general. Este consiste en un par de llaves con uno o mas numeros dentro, como en */x/{5,10}/*. El caracter inmediatamente preceente (en este caso la "x") debe de ser encotrada dentro del numero indicado de repeticiones (cinco a diez aqui).

Claro , */\d{3}/* no solo concuerda con numeros de tres digitos. Tambien concuerda con cualquier numero con mas de tres digitos en el. Para concordar exactamente tres, necesitamos usar anclas, descritas mas adelante.

Si dejamos el segundo numero, como en */x{5,}/*, significa *"esos tantos o mas"* (cinco o mas en este caso), y

no utilizamos la coma como en */x{5}/*, significa *"exactamente estos"* (cinco x's). Para obtener cinco o menos x's se debe de utilizar el cero como en */x{0,5}/*.

Asi, la expresion regular /a.{5}b/ concuerda la letra a separada de la letra b por cualquiera cinco caracteres diferentes de fin de linea en cualquier punto de la cadena. (recuerda que un punto concuerda con cualquier caracter diferente de fin de linea) .

Parentesis como memoria. Otro operador de grupo es un par de parentesis abiertos y cerrados alrededor de cualquier parte del patron. Esto no cambia si el patro concuerda o no , pero causa que la parte de la cadena que concuerde con el patron sea recordada para ser referenciada mas tarde. Por ejemplo, (a) aun concuerda una a, y ([a-z]) aun concuerda cualquier letra minuscula.

Para recordar una parte memorizada de una cadena, se debera incluir una diagonal invertida seguida por un entero. Esta construccion de patron representa la misma secuencia de caracteres concordadas anteriormente en el mismo par numerado de parentesis (contando desde uno) . Por ejemplo:

```
/pedro(.)pablo\1/;
```

Concuerda una cadena consistiendo de pedro, seguido de cualquier caracter sin salto de linea, seguido de pablo,seguido por ese mismo caracter simple. Asi que este concuerda *pedroxpablox*, pero no *pedroxpabloy*. Comparemos con

```
/pedro.pablo./;
```

En el cual los dos caracteres no especificados pueden ser los mismos o diferentes; no importa.

De donde vino el 1?, Significa la primera parte con parentesis de la expresion regular. Si existen mas que una, la segunda parte (contando de la izquierda a la derecha) se referencia como \2, el tercero como \3 y asi sucesivamente.

Alternancia Otra construccion de agrupacion es la alternancia, como en a|b|c. Esto significa concordar exactamente una de las alternativas (a o b o c en este caso). Esto trabaja aun si las alternativas tienen multiples caracteres, coom en /cancion|azul/, el cual puede concordar cancion o azul.

Patrones ancla Algunas notaciones especiales anclan un patron. Normalmente, cuando un patron concuerda contra la cadena, el principio de el patron es arrastrado atraves de la cadena de izquierda a derecha concordando a la primer oportunidad disponible. Las anclas te permiten asegurar que partes del patron se alinean con partes particulares de la cadena.

Este primer par de anclas require que una parte particular de la concordancia este localizado ya sea en un limite de palabra o no. El ancla \b require que un limite de palabra en el punto indicado por el patron concuerde. Un limite de palabra es el lugar entre caracteres que concuerda \w y \W , o entre caracteres concordadndo \w y el principio o final de la cadena Notese que esto tiene poco que ver con palabras inglesas y mucho con simbolos de C, pero eso es lo mas cercano. Por ejemplo.

```
/pedro\b/;       # concuerda pedro, pero no pedrito.
/\bmo/;          # concuerda moe y mole, pero no Elmo
/\bPedro\b/;     # concuerda Pedro pero no Pedrito o San Pedro.
/\b\+\b/;        # concuerda "x+y" pero no "++" o " + "
/abc\bdef/;      # Nunca concuerda (imposible un limite ahi)
```

De la misma manera \B requiere que no exista un limite de palabra en el punto indicado. Por ejemplo:

```
/\bPedro\B/;   # Concuerda ‘‘Pedro’’ pero no ‘‘Pedro Picapiedra’’
```

Dos anclas mas requiren que una parte particular del patron sea el siguiente al final de la cadena. El (^) concuerda el principio de la cadena si esta en un lugar que haga sentido concordar con el principio del a cadena.

Por ejemplo, ^a concuerda a si y solamente si , a es el primer caracter de la cadena. Sin embargo , a^ concuerda los dos caracteres y un ^ en cualquier parte de la cadena. En otras palabras el simbolo (^) pierde su significado especial.

El simbolo $, como ^ , ancla al patron pero al final de la cadena, no al principio. En otras palabras, c$ concuerda una c solo si ocurre al final de la cadena.

7.4 Mas acerca del operador de concordancia.

7.4.1 Seleccionando un destino diferente (el operador =~)

Usualmente la cadena que queremos hacer concordar no esta dentro de la variable $_, y podria ser un problema ponerla ahi. Para eso es el operador =~. Este operador toma un operador de expresion regular en el lado derecoh y cambia el destino del operador a algo diferente de la variable $_, por ejemplo:

```
$a=’’hello world’’;
a$ =~ /^he/;    # verdadero
$a =~ /(.)\1/;  # concuerda la doble l
if ($a =~ /(.)\l/)
{ # verdadero.
  # algo que hacer
}
```

El destino del operador =~ puede ser cualquier expresion que asigne a un escalar un valor de cadena. Por ejemplo <STDIN> .

7.4.2 Ignorando mayusculas.

Para cadenas muy cortas, como y o pedro es facil utilizar [fF] o [oO]. Pero que tal si la cadena que queremos concordar esta en mayusculas y minusculas?. En algunas versiones de grep , una opcion -i ignora las mayusculas. Perl tambien tiene esa opcion. Se indica que ignore mayusculas agregando una i en minusculas a la diagonal final, como en /algunpatron/i. Esto dice que las letras del patron concordaran en cualquier caso. Por ejemplo para concordar la palabra procedimiento en cualquier caso al principio de la linea, utilice /^procedimiento/i.

7.4.3 Usando un delimitador diferente.

Si buscamos por una cadena con una expresion regular que contenga una diagonal (/), debemos preceder cada diagonal con una diagonal invertida (\). Por ejemplo , podemos buscar por una cadena que empiece con /usr/etc com esto:

```
$ruta = <STDIN>; # Lee la ruta de acceso
if ($ruta =~ /^\/usr\/etc/)
{
# ok
}
```

opcionalmente podemos cambiar el caracter de limite. Simplemente precedemos un caracter no alfanumerico con una m y listamos nuestro patron seguido por otro limitador identico, como en :

```
m@^/usr/etc@ # utilizamos el delimitador @
```

7.5 Substituciones

Ya hablamos de la forma mas simple de substitucion: s/old/new/. Es tiempo de unas pocas variaciones.

Si queremos que el reemplazo opere con todas las concordancias posibles en lugar de la primera concordancia , agregue una g a la substitucion como en :

```
$_ = "foot fool buffoon"; s/foo/bar/g; # $_ es ahora "bart barl bufbarn"
```

La cadena de reemplazo esta interpolada, permitiendo especificar una cadena de reemplazo en tiempo de ejecucion:

```
$_ = "hello, world"; $new = "goodbye"; s/hello/$new/; # reemplaza hello con goodbye
```

Los caracteres de patron en la expresion regular permite a los patrones concordar, en lugar de caracteres fijos:

```
$_ = "this is a test"; s/(\w+)/<$1>/g; # $_ es ahora "<this> <is> <a> <test>"
```

Recuerde que $1 es modificado con los datos dentro de la primera concordancia de parentesis.

Una i de sufijo (antes o despues de la g si esta presente) causa que la expresion regular en el operador de substitucion ignore mayusculas, como en la misma opcion del operador descrito previamente.

7.6 Las funciones *split* y *join*

Las expresiones regulares pueden ser utilizadas para separar una cadena en campos. La funcion de split lo hace, y la funcion de join junta las piezas de nuevo.

7.6.1 La funcion *split*

La funcion *split* toma una expresion regular y una cadena, y busca por todas las ocurrencias de la expresion regular dentro de esa cadena. Las partes de la cadena que no concuerden con la expresion regular son regresadas en secuencia como en una lista de valores. Por ejemplo , aqui hay algo para separar campos separados por dos puntos, como en los archivos unix /etc/passwd

```
$line = "merlyn::118:10:Randal:/home/merlyn:/usr/bin/perl";
@fields = split(/:/,$line); # divide  $line, usando : como delimitador
                            # ahora @fields es ("merlyn","","118","10","Randal",
                            # "/home/merlyn","/usr/bin/perl")
```

7.6.2 La funcion *join*

La funcion *join* toma una lista de valores y las junta utilizando una cadena entre si. Este luce asi:

```
$cadena = join($pegamento, @lista);
```

7.7 Ejercicios

1. Construya una expresion regular que concuerde:
   (a) Al menos una a seguida por cualquier numero de b's
   (b) Cualquier numero de diagonales invertidas seguidas por cualquier numero de asteriscos (cualquier numero puede ser cero.
   (c) Tres copias consecutivas de lo que este contenido en $loquesea
   (d) Cualesquiera cinco caracteres, incluyendo salto de linea.
   (e) La misma palabra escrita dos o mas veces, donde "palabra" esta definida como una secuencia no vacia de caracteres sin espacio.

2. Escriba un programa que acepte una lista de palabras en STDIN y busque por una linea conteniendo todas las cinco vocales (a,e,i,o y u). Corra este programa en /usr/dict/words y vea que se muestra. en otras palabras introduzca: miprograma </usr/dict/words
   (a) modifique el programa para que las cinco vocales esten en orden y las letras que intervengan no importen
   (b) modifique el programa para que todas las vocales esten en orden creciente, de manera que las cinco vocales esten presentes y no ocurra una "e" antes que una "a" y no "i"ocurra antes que una "e" y asi.

3. Escriba un programa que busque a traves de /etc/passwd, imprimiendo el nombre real y el login de cada usuario.

4. Escriba un programa que busque a traves de /etc/passwd por dos usuarios con el mismo nombre, e imprima esos nombres.

5. Repita el ultimo ejercicio , pero reporte los nombres de login de todos los usuarios con el mismo nombre.

8 Funciones.

8.1 Definiendo una funcion de usuario

Una funcion de usuario, mas comunmente llamada una subrutina o solo sub, se define en tu programa de Perl, usando una construccion como esta:

```
sub misub
{
  sentencia_1;
  sentencia_2;
  sentencia_3;
}
```

donde misub es el nombre de la subrutina, el cual es cualquier nombre como los nombres que tenemos por variables escalares,arreglos y hashes. Tecnicamente el nombre de la subrutina es &misub, pero raramente nos referimos a ella bajo ese nombre.

Dentro de una subrutina se pueden accesar o dar valores a las variables que son compartidas con el resto del programa (una variable global) . De hecho , por default, cualquier variable referenciada dentro del cuerpo de una subrutina se refiere a una variable global. Existen excepciones que luego veremos.

8.2 Invocando una funcion de usuario.

Se invoca a una subrutina dentro de cualquier expresion utilizando el nombre de la subrutina entre parentesis, como en :

```
say_hello(); # una expresion simple
$a = 3 + say_hello(); # parte de una expresion mas grande
for ($x = start_value();
$x < end_value();
$x += increment())
{ ... } #
```

Los niveles de subrutinas estan indefinidos.

8.3 Valores de retorno.

Una subrutina es siempre parte de una expresion. El valor de la invocacion de la subrutina se llama el valor de retorno. El valor de retorno de una subrutina es el valor de la sentencia return o de la ultima expresion evaluada en la subrutina.

Por ejemplo, definamos esta subrutina.

```
sub suma_de_a_y_b
{
    return $a + $b;
}
```

8.4 Argumentos

Aun cuando las subrutinas que tienen una accion especifica son utiles, existe un mayor nivel de utilidad cuando puedes pasar argumentos a una subrutina. En perl, la invocacion de subrutina es seguido por una lista dentro de parentesis, causando que la lista sea automaticamente asignada a una variable especial llamada @_ que dura lo que dura la vida de la subrutina. La subrutina puede accesar esta variable para determinar el numero de argumentos y el valor de esos argumentos. Por ejemplo:

```
sub di_hola
{
   print ‘‘Hola, $_[0]!\n’’;    # El primer parametro es el destino
}
```

Aqui , vemos una referencia a $_[0], el cual es el primer elemento del arreglo @_. Esto significa que podemos invocar a la subrutina y esta utilizara el primer parametro que le pasemos, por ejemplo:

```
di_hola(‘‘mundo’’);  # resultado: Hola mundo!
```

El exceso de parametros es ignorado.

8.5 Variables privadas en funciones

Ya hablamos de la variable @_ y como una copia local se crea para cada subrutina invocada con parametros. Para crear una variable escalar, arreglo o variables hash que funcionen de la misma manera se utiliza el operador *my*, que toma una lista de nombres de variables y crea versiones locales de el (o instancias, como quieras decirle). He aqui la funcion de suma utilizando el operador my.

```
sub add
{
my ($suma);     # hace a $suma una variable local
$suma = 0;      # inicializa suma
foreach $_ (@_)
{ $suma += $_;  # suma cada elemento
}
return $suma;   # ultima expresion evaluada,
                # la suma de todos los elementos
}
```

Esto es tambien aplicable a arreglos y hashes como lo habiamos mencionado anteriormente. Una recomendacion para este tipo de manejo de variables es hacerlo *al principio de la subrutina*, de hecho tambien podemos hacer lo siguiente:

```
my($sum) = 0; # initialize local variable
```

Aunque el operador de sum es un ejecutable en realidad automaticamente asigna el valor a la variable al momento de inicializacion.

8.6 Ejercicios

1. Escriba una subrutina que tome un valor numerico de 1 a 9 como un argumento y regrese el nombre en español, como *uno, dos, o nueve*. Si el valor esta fuera de rango, regrese el numero original como el nombre. (La subrutina no debera de ejecutar ninguna E/S.
2. Tome la subrutina del ejercicio anterior, escriba un programa que tome dos numeros y los sume , mostrando el resultado como *Dos mas dos igual a cuatro*. (No olvide la mayuscula de la letra inicial).
3. Extienda la subrutina a que regrese un valor de menos nueve a menos uno y cero. Intentelo en un programa.

9 Estructuras de control miscelaneas

9.1 La sentencia *last*

En algunos de los ejercicios anteriores podriamos pensar: " Si tuvieramos una sentencia de break tipo C aqui, todo estaria bien". Pues bien, existe un equivalente para salir de un ciclo temprano: la sentencia *last*.

La sentencia last se sale de el bloque mas interno, provocando que la ejecucion continue con la sentencia inmediatamente siguiendo el bloque. Por ejemplo;

```
while (algo)
{
algo;
algo;
algo;
if (condicion)
{
   algunaotracosa;
   algunaotracosa;
   last;
}
mascosas;
mascosas;
}
```

La sentencia last cuenta solamente bloques de ciclo, no otros bloques.

9.2 La sentencia *next*

Como *last*, *next* altera la secuencia ordinaria de flujo de ejecucion. Sin embargo, *next* causa que la ejecucion se salte el ciclo sin terminar el bloque.

```
while (algo)
{
primeraparte;
primeraparte;
primeraparte;
if (condicion)
{
  otraparte;
  otraparte;
  next;
}
otraparte;
otraparte; # next llega hasta aqui
}
```

9.3 La sentencia *redo*

La tercera manera en que se puede saltar en un ciclo es con *redo*. Esta construccion causa un salto al principio del ciclo (sin reevaluar la expresion de control) .

```
while (condicion)
{ # redo llega aqui.
algo;
algo;
algo;
if (condicion)
{
algo;
algo;
redo;
}
mas;
mas;
}
```

Una vez mas , el bloque if no cuenta, solo los ciclos.

9.4 Bloques con etiquetas.

Y que tal si me quiero saltar dos bloques?. En C usariamos el goto, pero en Perl podemos etiquetar cada bloque y utilizar ese nombre en last, next y redo.

Una etiqueta es otro nombre del tipo escalar, pero no utiliza caracteres especiales de puntuacion, ya que este nombre podria causar problemas de conflicto con palabras reservadas, es conveniente utilizar mayusculas en su definicion. ejemplo:

```
ETIQUETA: while(condicion)
{
linea;
linea;
linea;
if(condicion1)
{
   last ETIQUETA;
}
}
```

He aqui otro ejemplo:

```
EXTERIOR: for ($i = 1; $i <= 10; $i++)
{
   INTERIOR: for ($j = 1; $j <= 10; $j++)
  {
   if ($i * $j == 63)
   {
     print "$i veces $j es 63!\n";
     last EXTERIOR;
   }
   if ($j >= $i)
   {
     next EXTERIOR;
   }
  }
 }
```

9.5 Modificadores de expresion

Otra manera de indicar "si esto, entonces eso" en Perl es insertando un modificador en una expresion que es una simple sentencia,como esto:

```
expresion if expresion_de_control;
```

En este caso , la expresion de control es evaluada primero por su valor verdadero, usando las mismas reglas de siempre, y si es verdadero, se evalua la expresion siguiente. Esto es equivalente mas o menos a:

```
if( expresion_de_control){
 expresion;
}
```

Excepto que no se necesitan lineas adicionales. Por ejemplo he aqui como podemos salirnos de un ciclo cuando cierta condicion exista:

```
LINEA: while (<STDIN>) {
last LINEA if /^From: /;

}
```

9.6 && y || como estructuras de control.

Aunque parecen caracteres de puntuacion o parte de la expresion, pueden ser considerados estructuras de control. Frecuentemente pensamos "*si esto , entonces eso*" y ya vimos dos de estas formas:

```
if (this) {
that; # una manera de hacerlo
}
if this; # otra manera.
```

existe una tercera manera, y aun hay otras.

```
this && that;
```

como trabaja?, he aqui la explicacion, veamos que pasa en cada situacion de verdadero o falso.

Si es verdadero, entonces el valor de la expresion entera es desconocida, porque depende de el valor de *that*. Asi que that tiene que ser evaluado.

Si es falso , no hay razon de continuar, porque el valor de la expresion entera tiene que ser falso. Como no hay punto para evaluarlo lo saltamos.

De hecho , perl solamente evalua *that* cuando es verdadero, haciendolo equivalente a las dos formas previas.

Asimismo, el or logico trabaja como la sentencia unless . Asi que podemos reemplazar:

```
unless (this) {
that;
}
```

con

```
this || that;
```

Esto es probablemente familiar para los que estan familiarizados con programacion de comandos en shells.

9.7 Ejercicios.

1. Extienda el problema del tema pasado para repetir la operacion hasta que la palabra fin se introduzca para uno de los valores: (*tip: utilice un ciclo infinito, y haga un last si el valor es fin*).

2. Reescriba el ejercicio del tema 4 , sumando los numeros hasta el 999 utilizando un ciclo que salga del medio. (*tip: utilice un bloque con un redo al final para obtener un loop infinito y un last en el medio basado en una condicion*).

10 Filehandles y archivos

10.0.1 Que es un *filehandle*?

Un filehandle en Perl es el nombre para una conexion de E/S entre el proceso de perl y el mundo exterior. Ya hemos visto filehandles implicitamente: STDIN es un filehandle, asimismo perl provee STDOUT y STDERR. Esos nombre son los mismos que los proveidos por C y C++.

Los nombres de los filehandles son iguales a los utilizados por los bloques con nombre, pero vienen de otro espacio de nombres asi que podemos tener un escalar $pedro , un arreglo @pedro, un hash %pedro, una subrutina &pedro, una etiqueta pedro y un filehandle pedro. La recomendacion sin embargo es utilizar mayusculas con el objeto de evitar conflictos con palabras reservadas.

10.1 Abriendo y cerrando un filehandle

Perl provee tres filehandles , STDIN,STDOUT y STDERR , que se abren automaticamente a archivos o dispositivos establecidos por el proceso padre del programa. (probablemente un shell). Pero se puede utilizar la funcion open() para abrir filehandles adicionales. su sintaxis es la siguiente:

```
open(FILEHANDLE, ‘‘nombre’’);
```

donde FILEHANDLE es el nuevo filehandle y nombre es el nombre del archivo externo, (como un archivo o un dispositivo) que estara asociado con el nuevo filehandle. Esta invocacion abre el filehandle para leer. Para abrir un archivo en modo de escritura, utilice la misma funcion, pero prefije el nombre del archivo con un signo de mayor que (como en el shell) :

```
open(SALIDA,’’>salida’’);
```

Cuando terminamos con el uso de un filehandle, lo cerramos con el operador close, como en la siguiente manera:

```
close(SALIDA);
```

El reabrir un filehandle tambien abre el archivo previamente abierto automaticamente, como el salirse del programa tambien lo hace.

10.2 Un cambio diferente: *die*.

Consideremos esto un pie de nota, en el medio de la pagina.

Un filehandle que no ha sido abierto exitosamente puede ser utilizado aun sin que perl te prevenga a traves del programa. Si lees el filehandle obtendras un fin de archivo inmediatamente. Si escribes entonces los datos se desechan silenciosamente.

Tipicamente, deberemos de checar el resultado del open y reportar un error si el resultado no es lo que esperamos. Claro , podriamos usar algo como ;

```
unless (open (DATOS,''>/tmp/datos'')){
print ''Lo siento, no pude crear /tmp/datos'';
}
```

Pero eso es mucho trabajo, y pasa lo suficiente para que perl ofrezca un medio de hacerlo mas rapido. La funcion *die* toma una lista con parentesis opcionales, escupe esa lista (como print) hacia la salida estandar de error y finaliza el proceso de perl, con un estatus no cero. Asi que reescribiendo la funcion previa tenemos:

```
unless (open DATOS,''>/tmp/datos''){
die ''Lo siento,no pude crear /tmp/datos'';
} # resto del programa.
```

Pero, considerando que tenemos la expresion de control || podemos hacerlo de otra manera:

```
open(DATOS,''>/tmp/datos'') || die ''Lo siento,no pude crear /tmp/datos'';
```

De esta manera die se ejecuta solamente cuando el resultado del open es falso.

Otra particularidad practica es la variable $!, que contiene el mensaje describiendo el error. Asi que podemos decir:

open(DATOS,''>/tmp/datos'') || die ''Lo siento,no pude crear /tmp/datos:$!'';

10.3 Utilizando filehandles.

Una vez que un filehandle se abre en modo de lectura, se pueden leer lineas como leer de la entrada estandar, por ejemplo:

```
open (EP,"/etc/passwd");
while (<EP>) {
chomp;
print "Vi $_ en el archivo de password!\n";
}
```

Si queremos utilizar un filehandle para escritura, debemos de colocar el filehandle inmediatamente despues del print y antes que otros argumentos. No deben de existir comas entre el filehandle y el resto de los argumentos.

ejemplo;

```
print LOGFILE ‘‘Termina proceso $n de $max\n’’;
print STDOUT ‘‘Hola, mundo!\n’’;
```

A continuacion mostramos un ejemplo de como copiar datos de un archivo especificado en $a en un archivo especificado en $b .

```
open(IN,$a) || die "no puedo abrir $a en modo de lectura: $!";
open(OUT,">$b") || die "no puedo crear $b: $!";
while (<IN>) {
# lee una linea del archivo $a en $_
print OUT $_; # imprime la linea en  $b
}
close(IN) || die "no puedo cerrar $a: $!";
close(OUT) || die "no puedo cerrar $b: $!";
```

Claro que hay funciones en Perl que ya hacen esto.

10.4 Las funciones *stat* y *lstat*.

El operando que se le pasa a *stat* es un filehandle o una expresion que evalua a un nombre de archivo. El valor de retorno es undef, indicando que el stat fallo o una lista de 13 elementos, descritos utilizando la siguiente lista de variables escalares.

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,
$size,$atime,$mtime,$ctime,$blksize,$blocks) = stat(...)
```

Esto esta documentado en el modulo File::stat.

Por ejemplo, para obtener el user ID y el group ID del archivo de password, podemos intentar:

```
($uid, $gid) = (stat("/etc/passwd"))[4,5];
```

Esto nos proporciona el dueño del archivo y el grupo al que pertenece. Si invocamos stat con el nombre de un link simbolico este regresa informacion sobre el link al cual apunta, no al link por si mismo. Para hacerlo sobre el link utilizaremos la funcion *lstat*.

Como las pruebas de archivos, el operando de stat o lstat utiliza $_.

10.5 Ejercicios.

1. Escriba un progama que lea un archivo de STDIN , abra ese archivo y despliegue los contenidos de cada linea precedidos por el archivo y dos puntos. Por ejemplo, si pedro se lee, y el archivo pedro consiste de tres lineas, aaa , bbb y ccc, veremos pedro: aaa, pedro: bbb y pedro:ccc.

2. Escriba un programa que pregunte por un archivo de entrada un archivo de salida , un patron de busqueda y una cadena de reemplazo, y reemplace todas las ocurrencias del patron de busqueda con la cadena de reemplazo mientras se copia el archivo de entrada a el archivo de salida. Intentelo en varios archivos. Se puede sobreescribir el archivo. Puedes utilizar caracteres de expresiones regulares en la cadena de busqueda?. Pudes usar $1 en la cadena de reemplazo?.

3. Escriba un programa que lea una lista de archivos y despliegue cual de los archivos es de lectura, de escritura y/o ejecutable, y cual no existe.

4. Escriba un programa que lea una lista de archivos y encuentre el mas viejo entre ellos. Imprima el nombre del archivo y la edad del archivo en dias.

11 Formatos

11.1 Que es un formato?

Perl significa "practical extraction and report language". Es tiempo que sepamos acerca de "...report language".

Perl, provee la nocion de un reporte simple escribiendo un template, llamado formato. Un formato define una parte constante (los encabezados de las columnas, etiquetas, texto fijo o cualquier cosa) y una parte variable (los datos que usamos) . Este es muy similar a la salida formateada de COBOL o algunas clausulas de impresion de BASIC.

El usar un formato consta de tres cosas:

1. Definir un formato

2. Cargando los datos a ser impresos en las porciones variables del formato (campos)

3. Invocar el formato.

Frecuentemente , el primer paso es hecho una vez y los otros hechos repetidamente.

11.2 Definicion de un formato

Un formato se define utilizando una definicion de formato. Esta definicion de formato puede estar en cualquier parte del programa, como una subrutina. Una definicion de formato luce asi:

```
format nombreformato =
fieldline
valor, valor_dos, valor_tres
fieldline
valor, valor_dos
fieldline
valor, valor_dos, valor_tres .
```

La primer linea contiene la palabra reservada format, seguido por el nombre del formato y un signo de igual (=) . El nombre del formato se selecciona de otro nombre de espacio y sigue la misma regla que los demas. Debido a que los formatos nunca se usan el el cuerpo del programa puedes utilizarlo sin riesgo de tener conflictos con palabras reservadas.

Despues de la primera linea vienen el template en si mismo, con cero o mas lineas de texto. El final del template se indica por una linea consistiendo de un punto.

> *Los templates son sensitivos a los espacios en blanco, este es uno de los pocos lugares donde el tipo y cantidad de espacios en blanco importa en el texto de un programa Perl.*

La definicion del template contiene una serie de fieldlines. Cada fieldline puede contener texto fijo - que puede ser impreso literalmente cuando se invoca el formato. He aqui un ejemplo de un fieldline con texto fijo:

Hola, mi nombre es Pedro Picapiedra.

Los campos de linea tambien contienen contenedores de campo (fileholders) para texto variable. Si una linea contiene fileholders, la siguiente linea del template (llamado el valor de la linea) dicta una serie de valores escalares - uno por fileholder - que provee los valores que pueden insertarse en los campos. He aqui un ejemplo de un fieldline con un filholder y la linea de valor que sigue.

```
Hello, my name is @<<<<<<<<<<
$name
```

El fileholder es el `@<<<<<<<<<<` que especifica un texto justificado a la izquierda con 11 caracteres.

Si el fieldline tiene multiples fileholders, necesita valores multiples, los valores se separan del valor por comas:

```
Hello, my name is @<<<<<<<<<< and I'm @<< years old.
$name, $age
```

Al ponerlo todo junto podemos crear un formato simple para una etiqueta de direcciones:

```
format ADDRESSLABEL =
==============================
| @<<<<<<<<<<<<<<<<<<<<<<<<<<< |
$name
| @<<<<<<<<<<<<<<<<<<<<<<<<<<< |
$address
| @<<<<<<<<<<<<<<<<, @< @<<<< |
$city, $state, $zip
============================== .
```

Note que las lineas del signo de igual arriba y abajo del formato no tienen campos, y no tienen lineas de valor consecuentemente.

Una definicion de formato es como una subrutina. No contiene codigo que se ejecute inmediatamente, y puede ser colocado en cualquier parte del archivo con el resto del programa. Regularmente se tiende a ponerlo al final del archivo adelante de nuestras definiciones de subrutina.

11.3 Invocando un formato

Se invoca un formato con la funcion write. Esta funcion toma el nombre de un filehandle y genera texto para ese filehandle utilizando el formato actual para ese filehandle. Por omision , el formato actual para un filehandle es un formato con el mismo nombres (asi para STDOUT, el formato STDOUT se utiliza). Pero podemos cambiarlo.

Veamos de nuevo al formato de etiqueta de direcciones y creamos un archivo de direcciones y etiquetas:

```
format ADDRESSLABEL =
==============================
| @<<<<<<<<<<<<<<<<<<<<<<<<<<< |
$name
| @<<<<<<<<<<<<<<<<<<<<<<<<<<< |
$address
| @<<<<<<<<<<<<<<<<, @< @<<<< |
$city, $state, $zip
============================== .
open(ADDRESSLABEL,">labels-to-print") || die "can't create";
open(ADDRESSES,"addresses") || die "cannot open addresses";
while (<ADDRESSES>) {
   # remueve el salto de linea.
   chomp;
   # cargamos las variables globales.
   ($name,$address,$city,$state,$zip) = split(/:/);
   write (ADDRESSLABEL); # imprimimos

}
```

11.4 Cambiando la seleccion por omison de los formatos.

11.4.1 Utilizando *select()* para cambiar el filehandle

El filehandle seleccionado por default para salida estandar es STDOUT, nosotros podemos modificarlo utilizando la funcion select(), esta toma un filehandle como parametro y lo pone como salida por omision para ese filehandle , y esto afecta todas las operaciones futuras por omision que dependen del filehandle seleccionado.

por ejemplo:

```
print ‘‘Hello world\n’’;              # como print STDOUT
select(LOGFILE);                       # selecciona un nuevo filehandle.
print ‘‘hola, mundo\n’’;              # imprime a LOGFILE
select (STDOUT);                       # esto va a la salida estandar.
```

Asi que una mejor definicion de STDOUT con respecto a print y write es que es el handle por omision seleccionado.

11.4.2 Cambiando el nombre del formato.

El nombre del formato por omision de un filehandle en particular es el mismo que el filehandle. Sin embargo, podemos cambiarlo para este filehandle en particular seleccionando el nuevo nombre de formato en una variable llamada $~. Por ejemplo, para utilizar el formato ADDRESSLABEL en STDOUT, es tan facil como decir:

```
$~ = "ADDRESSLABEL";
```

Pero que tal si quieres especificar el formato para el filehandle REPORT a SUMMARY?. Solo unos pasos adicionales para hacer esto:

```
$oldhandle = select REPORT;
$~=’’SUMMARY’’;
select ($oldhandle);
```

La proxima vez que digamos :

```
write (REPORT);
```

obtendremos la salida en el filehandle REPORT pero usando el formato SUMMARY.

Nota: el modulo object-oriented Filehandle, parte de la distribucion estandar de Perl provee una manera mas simple de hacer lo mismo.

11.4.3 Cambiando el nombre del formato titulo de la pagina.

Tal como hacemos para cambiar el formato de un filehandle en particular cambiando la variable $~, podemos cambiar el formato de titulo de pagina al modificar la variable $^. Esta variable contiene el nombre del formato del titulo de la pagina para el filehandle seleccionado y es lectura/escritura, significando que poodemos examinar su valor para ver el nombre de formato actual y puedes cambiarlo al asignarlo a el.

11.4.4 Cambiando la longitud de pagina.

Si el formato de titulo de pagina se define, la longitud de pagina se vuelve importante. Por omision es 60 lineas; esto es cuando un write con cabe al final de la linea 60, el formato de titulo de pagina se invoca automaticamente antes de imprimir el texto.

Este valor se puede modificar a traves de la variable $=.

11.4.5 Cambiando la posicion de la pagina.

Si imprimimos texto a un filehandle, este modifica la cuenta de lineas de la posicion de pagina porque Perl no cuenta lineas para otra cosa que no sea un write. Si queremos hacerle saber a Perl que hemos impreso unas cuantas lineas extras debemos ajustar el contador interno , el cual es la variable $-.

$- es puesto a cero al principio del programa para cada filehandle. Esto asegura que el contador de titulo de pagina sera invocado cara cada filehandle ala primera escritura.

11.5 Ejercicios

1. Escriba un programa que abra el archivo /etc/passwd por nombre e imprima el usuario, userID (numero) y nombre real en columnas formateadas. Utilice format y write.

2. Agrege un formato de titulo de pagina al programa previo.

3. Agregue un numero de pagina secuencial al titulo de pagina, para ver pagina 1 , pagina 2 y asi en la salida.

12 Acceso de directorios

12.1 Moviendose alrededor del arbol de directorios.

Probablemente estas familiarizado con la nocion de directorio actual y usando el comando cd del shell. En programacion se invoca la llamada chdir para cambiar el directorio actual de un proceso, y eses es el nombre usado por perl tambien.

La funcion chdir en perl toma un argumento - evaluando a un nombre de directorio al cual el directorio actual sera puesto. Este regresa verdadero si lo hizo y falso no fallo. Ejemplo:

```
chdir("/etc") || die "cannot cd to /etc ($!)";
```

La funcion chdir sin parametros te lleva por omision a tu directorio home, como el comando cd en el shell.

12.2 Globbing

Es muy comun que utilicemos un asterisco en el manejo de nuestros comandos en un shell, por ejemplo si decimos /etc/host* obtendremos una lista de comandos que empiecen con host en el directorio etc, a la expansion de argumentos como * o /etc/host* en la lista de archivos que concuerdan se le llama *globbing*.

Perl soporta el globbing a traves de mecanismos simples: solamente pon el patron de globbing entre <> o utilice la funcion *glob* ejemplo:

```
@a = </etc/host*>;
@a = glob("/etc/host*");
```

En un contexto de lista, glob regresa una lista de todos los nombres que concuerdan con el patron. O una lista vacia si no concuerdan. Por ejemplo:

```
while (defined($nextname = </etc/host*>)) {
print "one of the files is $nextname\n";
}
```

12.3 Handles de directorios.

Si el sistema operativo en el cual se corre Perl provee la biblioteca readdir o su equivalente entoces podemos accesar a los handles de directorios. un handle de directorio es un nombre como los otros nombres de escalares , este representa una conexion a un directorio en particular. En lugar de leer datos , se usa el handle para leer listas de archivos dentro del directorio. Los handles de directorios son siempre de solo lectura.

12.3.1 Abriendo y cerrando un handle de directorio.

La funcion opendir trabaja como en C y C++. Le das a esta el nombre de un nuevo handle de directorio y un valor de cadena denotando el nombre del directorio a ser abierto. El valor de retorno de opendir es cierto si el directorio puede ser abierto y falso de otra manera.

Ejemplo:

```
opendir(ETC,"/etc") || die "Cannot opendir /etc: $!";
closedir(ETC);
```

12.3.2 Leyendo un handle de directorio

Una vez abierto , podemos leer la lista de nombres con readdir, que toma un solo parametro, el handle del directorio. Cada invocacion de readdir en un contexto escalar regresa con el siguiente nombre de archivo en orden aleatorio. Si no hay mas readdir regresa undef. Este es un ejemplo:

```
opendir(ETC,"/etc") || die "no etc?: $!";
while ($name = readdir(ETC)) {
# scalar context, one per loop
print "$name\n"; # prints ., .., passwd,
group, and so on
}
closedir(ETC);
```

12.4 Ejercicios

1. Escriba un programa que cambie el directorio especificado como entrada, entonces liste los nombres de los archivos en orden alfabetico despues de cambiarse ahi. (No mostrarla si el cambio no fue exitoso, en cambio si advertir al usuario).

2. Modificar al programa para que incluya todos los archivos, no solamente los que no empiezan con . Trate de hacerlo con glob y un handle de directorio.

13 Manipulacion de archivos y directorios

Aqui mostramos como manipular los archivos , no los datos que contienen. Asumiremos que estamos trabajando en un sistema POSIX para accesar archivos y directorios.

13.1 Removiendo archivos.

Ya vimos como crear archivos dentro de perl , ahora aprenderemos como removerlos.

La funcion unlink (llamado asi por la funcion POSIX) borra un nombre por archivo. Cuando el ultimo nombre para un archivo es removido, el archivo en si mismo es removido. Ej:

```
unlink ("pedro"); # adios pedro
print "que archivo quieres borrar? ";
chomp($name = <STDIN>);
unlink ($name);
```

La funcion unlink puede tomar una lista de nombres tambien:

```
unlink(‘‘perro’’, ‘‘datos’’);
unlink <*.o>;
```

El valor de regreso de unlink es el numero de archivos borrados.

He aqui un programa de ejemplo:

```
foreach $file (<*.o>) { # step through a list of .o files
unlink($file) || warn "having trouble deleting $file: $!";
}
```

13.2 Renombrando un archivo.

Para renombrar un archivo en Perl se utiliza la funcion rename(), este toma como argumento dos parametros, el viejo nombre y el nuevo nombre.

Ejemplo:

```
rename("pedro","pablo") || die "No puedo renombrar pedro a pablo: $!";
```

13.3 Funciones adicionales con archivos.

Perl soporta las siguientes funciones que operan sobre archivos:
Linkear directorios.

```
link("fred","bigdumbguy") || die "cannot link fred to bigdumbguy";
```

Crear directorios:

```
mkdir("gravelpit",0777) || die "cannot mkdir gravelpit: $!";
```

remover directorios:

```
rmdir("gravelpit") || die "cannot rmdir gravelpit: $!";
```

Modificar permisos:

```
chmod(0666,"fred","barney");
```

Modificar accesos a archivos

```
chown(1234, 35, "slate", "granite");
```

Modificar timestamps.

```
utime($atime,$mtime,"fred","barney");
```

13.4 Ejercicios

1. Escriba un programa que trabaje como rm, borrando los archivos dados como linea de comandos cuando es invoque el programa. Recuerde que los argumentos de la linea de comandos se encuentran en el arreglo @ARGV cuando el programa inicia.

2. Escribe un programa que trabaje como mv , renombrando el primer argumento de la linea de comandos a el segundo argumento de la linea de comandos.

3. Escriba un programa que trabaje como ln creando una liga del primer argumento de la linea de comandos al segundo argumento de la linea de comandos.

4. Si se tienen links simbolicos, modifique el programa del ejercicio previo para manejar un switch opcional.

5. Si se tienen links simbolicos escriba un programa que busque por todos los archivos con links simbolicos en el directorio e imprima el nombre y el valor simbolico del link como lo hace ls -l. Cree links simbolicos en el directorio actual y pruebelos.

14 Manejo de procesos

14.1 Usando *system* y *exec*

Cuando le damos a un shell una linea a ejecutar , el shell usualmente crea un nuevo proceso para ejecutar el comando. Este nueo proceso se convierte en un hijo del shell, ejecutado independientemente.

Similarmente Perl puede lanzar procesos de varias maneras. La manera mas simple de lanzar procesos es a traves de la funcion system. En su forma mas simple, esta funcion maneja una cadena simple que lanza un nuevo shell y el comando a ejecutar. Cuando el comando se termina La funcion del sistema regresa el valor de retorno del comando.

Ejemplo:

system(“date”);

Este ejecuta el comando date. Para la funcion system los tres archivos estandar (entrada estandar, salida estandar y error estandar) se heredan del proceso Perl.

14.2 Usando apostrofes

Otra manera de lanzar procesos es poner un comando /bin/sh entre apostrofes. Como el shell , esete envia un comando y espera por el proceso a que termine, capturando la salida estandar.

```
$ahora = ‘‘Son las ‘‘.‘date‘; #Obtiene texto y salida de fecha.
```

He aqui como manejarlo en un contexto de lista:

```
foreach $_ (‘who‘) {
# uno por cada linea de who
($who,$where,$when) = /(\S+)\s+(\S+)\s+(.*)/;
print "$quien $where en $when\n";
}
```

14.3 Usando procesos como filehandles.

Otra manera de lanzar procesos es crear un proceso que parezca un filehandle . Podemos crear un filehandle de proceso que capture la salida o provea entrada al proceso. He aqui un ejemplo de crear un filehandle de who.

```
open(WHOPROC, "who|"); # abre who como lectura
```

Note la barra vertical del lado dereco de who. Eso le dice a perl que no es un archivo sino un comando. Debido a que la barra esta del lado derecho el filehandle esta abierto de lectura, significando que la salida estandar sera capturada. Para el resto del programa WHOPROC es meramente un filehandle que esta abierto en modo de lectura. He aqui un ejemplo de leer los datos del comando en un arreglo:

```
@salidawho = <WHOPROC>;
```

Similarmente, para abrir un comando que espera entrada, abrimos un filehandle de proceso de escritura poniendo la barra vertical en la izquierda del comando, como :

```
open(LPR,''|lpr -Pimpresora'');
print LPR @reporte;
close(LPR);
```

No solamente uno sino varios comandos pueden ser ejecutados a la vez, por ejemplo, el la siguiente linea inicia un proceso de ls , cuya salida va a tail, que finalmente envia su salida a el filehandle WHOPR.

```
open(WHOPR, "ls | tail -r |");
```

14.4 Usando fork

Otra manera de crear procesos adicionales es clonar el proceso de perl utilizando la llamada de UNIX fork. La funcion fork simplemente hace lo que la funcion fork del sistema hace: crea un clon del proceso. Este clon, llamado hijo, comparte el mismo codigo ejecutable, variables e incluso archivos abiertos. Para distinguirlos de dos procesos , el valor de retorno de fork es cero para el hijo y no cero para el padro (o undef si el system call falla). El valor de no cero recibido por el padre pasa a ser el process id del hijo. Podemos checar el codigo de retorno y actuar acorde.

```
if (!defined($child_pid = fork())) { die "cannot fork: $!"; }
elsif ($child_pid) {
# I'm the parent
} else {
# I'm the child
}
```

Para aprovechar mejor este clon, necesitamos aprender un poco mas cosas acerca de las funciones wait, exit,y exec.

Lo mas simple es la funcion exec. Es como la funcion system, excepto que en lugar de lanzar un nuevo proceso para ejecutar un comando de shell, Perl reemplaza el proceso con el shell. Despues de un exec exitoso, el programa de Perl se ha ido, siendo reemplazado por el programa. Por ejemplo:

exec "date";

reemplaza al programa Perl con el comando date, causando la salida de date que vaya a la salida estandar del programa Perl. Cuando este termina no hay nada mas que hacer porque el programa de perl se ha ido.

Otra manera de ver esto es que la funcion system es como un fork seguido por un exec, como sigue:

```
# METHOD 1... using system: system("date");
# METHOD 2... using fork/exec:
unless (fork) {
# fork returned zero, so I'm the child, and I exec:
exec("date"); # child process becomes the date command
}
```

Esto no es correcto del todo , porque el comando date y el proceso padre estan trabajando al mismo tiempo, mezclando su salida probablemente. Lo que necesitamos hacer es decirle a padre que espera hasta que el proceso hijo termine. Eso es exactamente lo que la funcion wait hace; espera hasta que el hijo ha completado. La funcion waitpid es mas discriminativa; espera por un hijo especifico a que termine ejemplo:

```
if (!defined($kidpid = fork())) {
  # fork returned undef, so failed
  die "cannot fork: $!";
}
elsif ($kidpid == 0) {
  # fork returned 0, so this branch is the child
  exec("date"); # if the exec fails, fall through to the next statement
  die "can't exec date: $!";
}
else {
  # fork returned neither 0 nor undef,
  # so this branch is the parent
  waitpid($kidpid, 0);
}
```

14.5 Sumario de operaciones de proceso

Operacion	Entrada estandar	Salida estandar	Error estandar	Esperado?
system()	Heredado del programa	Heredado delprograma	Heredado del programa	Si
apostrofe	Heredado delprograma	Capturado como valor de cadena	Heredado delprograma	Si
n()como filehandle de salida	Conectado al filehandle	Heredado delprograma	Heredado del programa	Solo altiempo declose()
()como filehandlede entrada	Heredado del programa	Conectado al filehandle	Heredado delprograma	Solo altiempo declose()
exec, waitwaitpd	Seleccionado por el usuario	Seleccionadopor el usuario	Seleccionadopor el usuario	Seleccionadopor el usuario

14.6 Envio y recepcion de señales

Un metodo de comunicacion interprocesos es enviar y recibir señales. Una señal es un mensaje de un bit (significando "esta señal ha pasado") enviada a un proceso de otro proceso o del kernel. Las señales estan numeradas, usualmente de 1 a algun otro numero como 15 o 31. Algunas señales tienen un significado predefinido y son enviadas automaticamente a un proceso en ciertas condiciones (como fallas de memoria o excepciones de punto flotante). Otros son estrictamente generados de otros procesos. Esos procesos deben de tener permiso para enviar dicha señal. solo si eres un superusuario o si el que envia el proceso y el receptor tienen el mismo proceso se permite enviar la señal.

La respuesta a la señal es la accion de la señal. Las señales predefinidas tienen ciertas acciones por omision, com abortar el proceso o suspenderlo. Otras son completamente ignoradas por omision. Casi todas las señales pueden ser manipuladas, para ser ignoradas o capturadas.

Hasta aqui las generalidades. Cuando un proceso de Perl captura una señal, una subrutina escogida se invoca asincronamente y automaticamente, momentaneamente interrumpiendo lo que este ejecutando. Cuando la subrutina termina, lo que haya estado ejecutando se reinicia como si nada hubiera pasado.

Tipicamente la subrutina de captura de señal hace una de dos cosas: abortar el programa despues de ejecutar algun codigo de limpieza o prender alguna bandera (como una variable global) que el programa checa rutinariamente.

De hecho, hacer algo mas complicado que esto podria complicar las cosas; a muchos programas no le gustan ser ejecutados desde el programa principal y la subrutina al mismo tiempo.

Necesitas conocer los nombres de las señales para registrar un manejador de señales en Perl. Al registrar un manejardor de señales perl llamara a la subrutina seleccionada cuando la señal se reciba.

Los nombres de señales estan definidos en la pagina del manual de signal y usualmente en el include file de C /usr/include/sys/signal.h. Los nombres generalmente inician con SIG, como SIGINT, SIGQUIT y SIGKILL. Para declarar la subrutina mi_sigint() como el manejador de señal que maneja el SIGINT, ponemos el valor en el hash magico %SIG. En este hash, ponemos el valor de la llave INT (eso es SIGINT sin el SIG) al nombre de la subrutina que atrapara la señal de SIGINT, como:

```
$SIG{'INT'} = 'mi_sigint';
```

Pero claro que necesitamos una definicion para esa subrutina. He aqui una simple:

```
sub mi_sigint {
   $saw_sigint = 1; # enciende una bandera
}
```

He aqui otra mas larga:

```
$saw_sigint = 0; # limpia la bandera
```

```
$SIG{'INT'} = 'my_sigint_catcher'; # registra el catcher
foreach (@huge_array) {
# hace algo
# hace algo mas
# hace aun mas
if ($saw_sigint) {
# se encendio la interrupcion?
# algo de limpieza aqui;
}
}
$SIG{'INT'} = 'DEFAULT'; # restaura la accion de default.
```

El truco aqui es que el valor de la bandera es ejecutado en puntos utiles durante la evaluacion y es usado para salir del ciclo prematuramente, aqui tambien manejamos algunas acciones de limpieza. Note que la ultima sentencia en el codigo precedente: establecer la accion de default para el manejador de señal. El ponerlo a DEFAULT restaura la accion por omision para esta señal en particular. Otro valor util es IGNORE, significando que ignore la señal. Se puede hacer esto si no hay acciones de limpieza requeridas , y no quieres terminar las operaciones temprano.

Una de las maneras en que la señal SIGINT se genera es al hacer que el usuario presione el caracter de interrupcion seleccionado (como CTRL-C) en la terminal. Pero un proceso puede generar la señal SIGINT directamente utilizando la funcion kill. Esta funcion toma un numero de señal o nombre y envia la señal a la lista de procesos (identificados por el process ID) siguiendo a la señal. Asi, enviando una señal de un programa requiere el determinar el ID de los procesos recipientes. Supongamos que quieres enviar la señal 2 (tambien conocida como SIGINT) a los procesos 234 y 237. Es tan simple como esto:

```
kill(2,234,237);        #  envia  SIGINT a 234 y 237
kill ('INT', 234, 237); # lo mismo
```

14.7 Ejercicios.

1. Escriba un programa que maneje la salida del comando date para obtener el dia actual de la semana. Si el dia de la semana es un dia habil, imprimir *a trabajar*, de otra manera imprimir *ve a jugar.*

2. Escriba un programa que obtenga todos los nombres reales de los usuarios de el archivo /etc/passwd, y que transforme la salida de el comando who, reemplazando el nombre del login (primera columna) con el nombre real. (Tip: crear un hash donde la llave es el nombre del login). Trate esto con el comando who utilizando apostrofes y abierto como pipe. Cual es mas facil?.

3. Modifique el programa previo para que la salida automaticamente vaya a la impresora. (si no puedes accesar una impresora tal vez puedes enviarte un correo).

4. Supongamos que la funcion mkdir no funciona. Escriba una subrutina que no utilice mkdir, pero invoca /bin/mkdir con la funcion system.

5. Extienda la subrutina del ejercicio previo que emplee chmod para cambiar los permisos.

15 Transformacion de datos miscelaneas

15.1 Encontrando una subcadena.

Encontrar una subcadena depende en donde la perdiste. Si pasa que la perdiste dentro de una cadena grande estas de suerte, porque index puede ayudar. Por ejemplo:

```
$x = index($string,$substring);
```

Perl localiza la primera ocurrencia de la subcadena dentro de una cadena, regresando un entero a la ubicacion del primer caracter. El valor de indice regresado se basa en cero: si la subcadena es encontrada al principio de la cadena se obtiene un 0. Si es un caracter posterior se obtiene 1 y asi. Si no se encuentra se obtiene un -1.

Veamos este:

```
$donde = index("hello","e");            # $donde es 1.
$persona = "pablo";
$donde = index("pedro pablo",$persona); # $donde es 5.
@rockers = ("pedro pablo'');
$donde = index(join(" ",@rockers),$person); # lo mismo.
```

Notice that both the string being searched and the string being searched for can be a literal string, a scalar variable containing a string, or even an expression that has a string value. Here are some more examples:

```
$donde = index("una cadena muy larga","larga"); # $donde obtiene 13
$donde= index("una cadena muy larga","pedro");  # $donde obtiene -1
```

If the string contains the substring at more than one location, the index function returns the leftmost location. To find later locations, you can give index a third parameter. This parameter is the minimum value that will be returned by index, allowing you to look for the next occurrence of the substring that follows a selected position. It looks like this:

Si la cadena contiene la subcadena en mas un una ubicacion, la funcion index regresa la ubicacion mas a la izquierda. Para encontrar ubicaciones posteriores le podemos dar a index un tercer parametro. Este parametro es el valor minimo que sera regresado por index, permitiendo buscar por la siguiente ocurrencia de la subcadena que sigue a la posicion seleccionada. Como esto:

```
$x = index($bigstring,$littlestring,$skip);
```

He aqui unos ejemplos de como trabaja.

```
$where = index("hello world","l");   # regresa 2 (primero l)
$where = index("hello world","l",0); # lo mismo
$where = index("hello world","l",1); # aun lo mismo
$where = index("hello world","l",3); # ahora regresa 3
                                     # (3 es el primer lugar mas grande
                                     # o igual a 3)
$where = index("hello world","o",5); # regresa 7 (segundo o)
$where = index("hello world","o",8); # regresa -1 (nada despues de 8)
```

15.2 Ejercicios

1. Escriba un programa que lea una lista de archivos, separandolos en sus componentes principio y fin. (todo hasta la ultima diagonal es principio y todo despues dela diagonal es el fin. Si no hay diagonal todo es fin.) Trate esto con cosas como /pedro, pablo y pedro/pablo.

16 Revision de caracteristicas avanzadas.