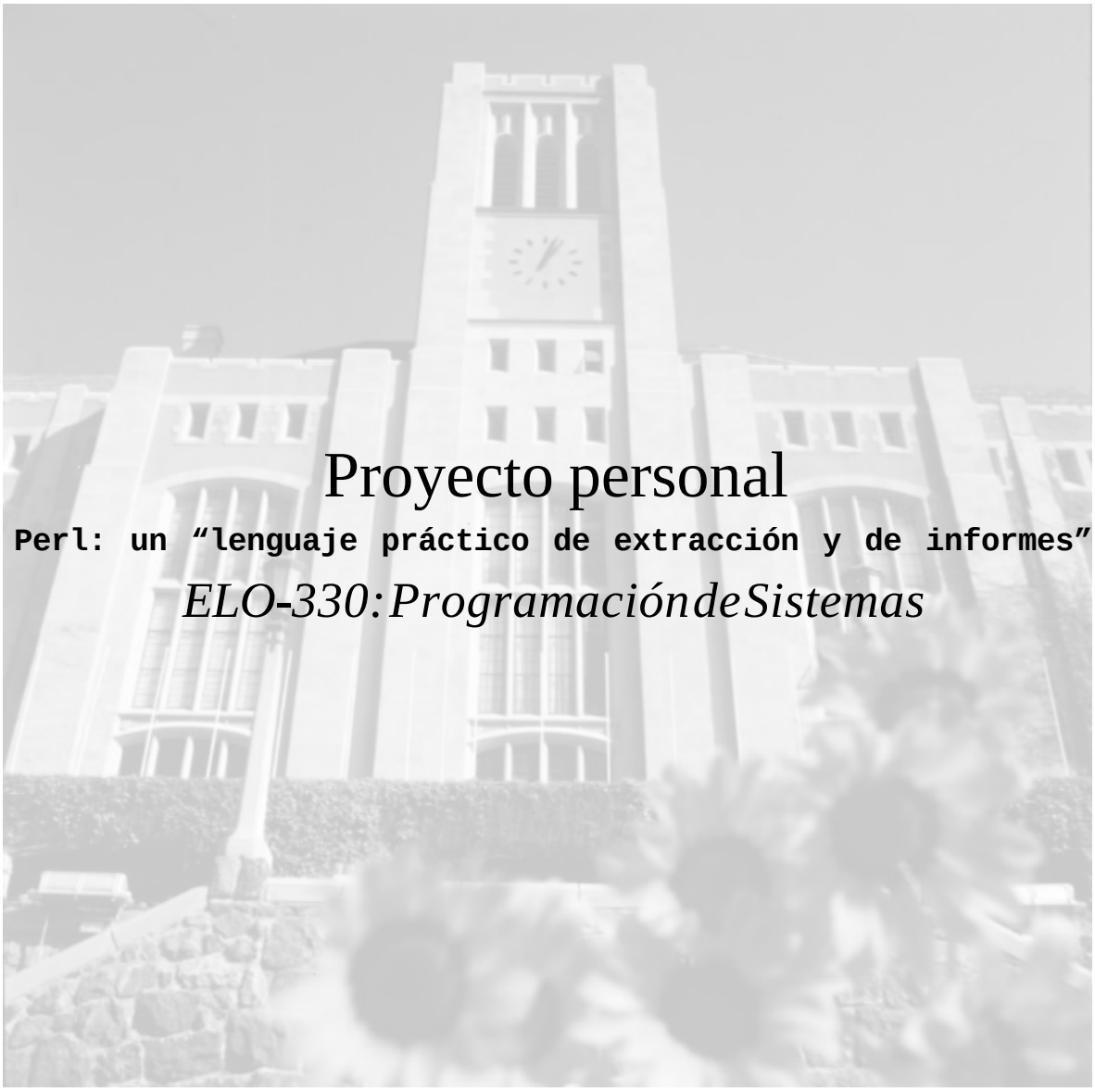
Universidad Técnica Federico Santa Maria
Departamento de Electrónica



Proyecto personal

Perl: un "lenguaje práctico de extracción y de informes"

ELO-330: Programación de Sistemas

Nombre: Claudio Ramírez S.
Rol: 9821064-4
Profesor: Agustín González
Fecha de entrega: 30/10/2002

Resumen

En este trabajo queremos aprender un nuevo lenguaje sus herramientas y la utilidad de éste, en las paginas Web's.

El lenguaje Perl no es nuevo pero sea potencializado con modificaciones para el mejor manejo de este lenguaje como es utilizar el concepto de programación orientada a objetos. Infelizmente en este trabajo no veremos esa "cara" del lenguaje.

En cambio veremos una de las mayormente utilizadas para este lenguaje que es el manejo de formularios, pero antes deberemos entender su sintaxis ya que utiliza mucho de otros lenguajes, que yo mismo no conocía.

Capitulo I: Introducción

PERL que significa "Practical Extraction and Report Language"(algo así como "lenguaje práctico de extracción y de informes") es un lenguaje de programación medianamente nuevo, el cual surgió de otras herramientas de UNIX como son: sed, grep, awk, c-shell. En realidad, puede hacer todo lo que hace cualquiera de ellos y todos ellos juntos, y la mayoría de las veces de forma más simple, comprensible y fácil de depurar.

Principalmente sirve para labores de procesamiento de texto, lo cual lo hace en una forma fácil y clara, no como es en C o Pascal; también sirve para la programación de software desistemas; y últimamente ha encontrado su aplicación en la escritura de CGI (*common gateway interface*), o scripts ejecutados desde páginas de la World Wide Web, por ejemplo un programa que ponga en pantalla las notas de un alumno dada su matricula, etc.

La mayoría de los programas que se encuentra uno para procesar formularios en la Internet llevan la extensión .**pl**, lo cual denota que están escritos en **PERL**.

PERL es un lenguaje interpretado, aunque internamente funciona como un compilador. Por eso se habla de *scripts*, y no de programas, concepto referido principalmente a programas compilados al lenguaje máquina propio del computador y sistema operativo en el que se ejecuta.

En general, los programas en **PERL** se ejecutan en el servidor, como todos los programas CGI, a diferencia de otros programas ejecutados por el cliente (generalmente un navegador como el Internet Explorer de Microsoft o el Navigator), como aquellos escritos en JavaScript o Java.

Sin embargo, existen actividades en las cuales **PERL**, no es la solución másadecuada, por ejemplo: sistemas de tiempo real, desarrollo de bajo nivel del sistema operativo que trabajen con los dispositivos del sistema de cómputo, aplicaciones de memoria compartida de procesos o aplicaciones extremadamente largas.

Capitulo II: Datos Escalares

Ejemplo práctico:

```
1:      #!/usr/local/bin/perl
2:      print ("Cual es tu nombre?");
3:      $nombre=<STDIN>;
4:      chop($nombre);          # esto es un comentario
5:      print "Hola, $nombre!\n";
```

- La primera línea lo que hace es llamar al compilador de **PERL.**
- En la segunda línea se imprime un mensaje, los paréntesis '( )' son opcionales.
- En la tercera línea se introducen el concepto de variable ($nombre), que se verá más adelante; <STDIN> indica que se leerá una línea desde la entrada standard.
- En la cuarta línea la función chop() lo que hace es sacarle el último carácter a $nombre, es decir le quita el Enter.
- Y finalmente en la última línea lo saludamos, en esta vemos que $nombre toma el valor que se le ingreso.

Los datos pueden ser números y string, al igual que en C. Para señalar los strings hay dos formas de hacerlo: comillas simples y dobles.

Estos se interpretan de la misma forma que en la línea de comando Shell. Para más detalles **ver el anexo**: **Uso de comillas**

Operadores

a) Operadores de comparación: Es idéntico a programación en Shell

Números	Strings
==	eq
!=	ne
<	lt
>	gt
<=	le
>=	ge

Números : Los operadores son los mismos que en C, lo nuevo seria el siguiente operador: 2**3 # 8

Strings :

a) Concatenación(.):

```
Ej:"Hola"."mundo"  #  "Holamundo"
     "juan"." "."pedro" # "juan pedro"
```

b) Operador de repetición de strings(x) :

```
Ej:"juan"x3 # "juanjuanjuan"
```

En **PERL** existen *tres tiposdevaraibles*: escalares ,arreglos y hash

1-Escalares

- Las variables de *escalares* empiezan por $

```
$a = 5;
$b ="xxx";
```

- Las instrucciones terminan en punto y coma...
- Un escalar puede tener números, strings u otras cosas más complicadas como referencias y descriptores.

Ej:$a,$nombre

Interpolación de variables escalares en strings

Si una variable es puesta en un string comillas dobles (") se interpola el valor de la variable.

Y si una variable se pone entre comilla simple (') no se interpola el valor de la variable.

En los strings existen también algunos modificadores: \U,\u,\L

```
Ej: $juan="juan"; $mayjuan="\U$juan"; # $mayjuan="JUAN"
    $capjuan="\u$juan"; # $capjuan=Juan
    $pedro="\LPEDRO"; # $pedro="pedro"
```

El valor undef (indefinido)

Si usamos una variable antes de darle un valor se asignara como cero si es número o " " (vacío) si es como string.

```
Ej: $a=3;
    $b=$a+$c; # $b=3
    $x="hola";
    $y=$x.$z; # $y="hola"
```

2- Arreglos

- las variables tipo *arreglos* empiezan por @

```
@a = (95, 7, 'fff' );
print $a[2];    # imprime el tercer elemento: fff
print  @a       # imprime: 957fff ...todo junto
```

- los elementos de un arreglo son escalares que empiezan por $
- los subíndices empiezan por 0 como en C, el primer elemento del arreglo @a es $a[0]
- el escalar $a no tiene que ver nada con $a[ ]

Obs.: En Perl hay mucha flexibilidad para escribir los argumentos

```
print ( $a, $b ); # con paréntesis
 print  $a, $b;   # sin paréntesis
```

- Los valores de los elementos pueden ser de distinto tipo.

Ej:
```
(1,2,3)
("juan",4.5)
($b+$c+17,$d+$e)
```

- El largo de un arreglo (ej : @a) queda guardado en $#a.
- Si uno accesa un elemento del arreglo más allá del fin del a rreglo el valor *undef* es retornado.
- Asignar un valor a un elemento más allá del fin del arreglo, lo que hace es agrandar el arreglo poniendo *undef* a los valores intermedios.

Ej:
```
@a=(1, "yo",3);
   @c=@a;       # copia el arreglo @a en @c
   $d=$a[$#a]; # $d=3
   $b=$a[0];    # asigna 1 a $b
   $a[1]=5;     # ahora @a es (1,5,3)
   $a[2]++;     # suma uno al tercer valor de @a siendo ahora (1,5,4)
   $b=$a[7];    # $b es 0 o "" , dependiendo del contexto
   $a[6]="chao"; # @a es ahora (1,5,4,undef,undef,"chao")
```

Para más operaciones sobre arreglos ver el anexo: **más sobre arreglos**

3- Hashes o arreglos asociativos

- Las variables de tipo *hash* empiezan por %
- Para crear un elemento de un *hash* se requiere una lista de 2 valores
- El primer elemento es la clave y el segundo es el valor

```
%a = ( 'x', 5, 'y', 3); # llena 2 elementos del hash
```

```
print $a{'x'};  # imprime: 5
print $a{'y'};  # imprime: 3
```

- Si la clave es un string sencillo se puede omitir las comillas

```
$a{x} es lo mismo que $a{'x'}
```

- Si se reasigna un elemento se pierde el valor viejo

```
%a = ('x', 5, 'y', 3 );
$a{x}=7;
print  $a{x};  #  imprime: 7
```

- Los elementos se accesan por claves y no hay claves duplicadas

Para más sobre *hash's* ver el anexo: **más sobre Hash**

Estructuras de control

Al igual que en la mayoría de los lenguajes de programación , en **PERL** existen estructuras como *if, for*, *while*. La sintaxis del *if , for y while* es la misma que en C, por lo que no se veran.

- ***Until***

La forma del *until* es :

```
until ( condicion ) {
        sentencia;
}
```

A diferencia del *while* el *until* se ejecuta al menos una vez, y se sigue mientras la condición sea falso.

- ***Foreach***

El *foreach* recibe una lista de valores y asigna cada uno de los valores de la lista a una variable de lectura . La estructura del *foreach* es :

```
foreach $a ( @alguna_lista ) {
        sentencia;
}
```

"Todas las estructuras de control necesitaban de llaves, aunque solo fuera una instrucción"

Capitulo III: Archivos

En este capítulo veremos archivos y también algo de Entrada Standar y del paso de parámetros a un programa.

Entrada estándar

Como vimos en el primer ejemplo, para leer una línea desde la entrada estándar se debe asignar una variable escalar a <STDIN>

```
Ej:  $a=<STDIN>; # guarda en $a la línea leída
      chop($a);  # le sacamos el enter que tienen todas las líneas por su
                 ingreso
```

Lectura de parámetros pasados a un programa

Dentro de un programa en **PERL** los argumentos que se le pasan quedan guardados en el arreglo @ARGV.

```
Ej.:    #!/usr/local/bin/perl
            print "Tu ingresaste las palabras:\n";
            for($x=0;$x<$#ARGV;$X++)
               {
               print "$ARGV[$x]\n";
               }
 %>prog1.pl hola mundo chao  #esto es la ejecución
        %> Tu ingresaste las palabras:
        %> hola
        %>mundo
        %>chao
```

Procesamiento de archivos

Como en todo lenguaje en **PERL** se puede trabajar con archivos.

- Abrir: Para abrir un archivo podemos usar el siguiente ejemplo:

```
Ej: open(FILE,"file1");
   while ( $a=<FILE> ){
     ...sentencia..;
    }
   close(FILE);
```

Lo que hicimos en la 1ª línea es abrir el archivo de nombre file1 y a ese archivo le asignamos el "file handler" : FILE (debe ser cualquier nombre pero en MAYUSCULA). Luego con el *while* leemos línea a línea (también podríamos haberlo hecho con <STDIN> de asignar una variable de tipo arreglo a <FILE> y habríamos leído el archivo de una pasada). Finalmente algo muy importante, es cerrar el archivo.

- Escritura: Para escribir ver el siguiente ejemplo.

Ej:

```
$file='hola.c';
open (A,">$file");   # Abrimos para solo escritura el archivo
hola.c
print A "hola\n";
close(A);
```

- Append: Esto corresponde a escribir pero añadiendo al archivo:

Ej:

```
@l=('a',"\n",'b',"\n");
open(G,">>/user/jperez/d.txt");  # Abrimos para escritura pero
                  #sin borrar el contenido solo añadiendo al archivo
print G @l;
close (G);
```

Una forma más elegante seria usando la "funcion" *die*:

Ej: `open (A, "file") || die "No se puede abrir\n";`

En este ejemplo, en caso de no poder abrir el archivo 'file' se ejecutara la sentencia ***die***, que una vez que muestre el mensaje termina el programa como una excepción(se termina inmediatamente).

Ej.: Un programa que simula 'cp'

```
$#ARGV != 2 || die "use copiar.pl <origen> <destino>";
open (A,"$ARGV[0]")||die("No se puede leer el origen");
open (B,">$ARGV[1]")||die("No se puede escribir en el destino");
while(  )
   {
   print B || die "No se pudo escribir";
   }
```

Inclusion de archivos

Al igual que en C, uno puede incluir otro archivo con código **PERL**.

Con **require** incluimos un archivo, el cual no necesariamente tiene que tener un 'main', es decir pueden ser solo subrutinas.

Ej.:

```
#!/usr/local/bin/perl
require "otroarchiv.pl"; # sirve para agregar el código que
                         #esta en él archivo otroarchiv.pl
```

Capítulo IV: Expresiones regulares

Es una forma general de describir un patrón de caracteres que queremos buscar en un string. En **PERL** se usa la misma notación para expresiones regulares de *sed* y *grep* :

cuantificadores

Se usan para indicar que algunas letras se repiten

.(un punto) **: Cualquiera excepto el carácter de línea nueva.**

\+ **: Uno o más del carácter anterior(que lo precede).**

? **: Ninguno o uno del carácter anterior.**

* **: Ninguno o más del carácter anterior.**

^ **: Busca la coincidencia sólo al comienzo de la línea.**

$ **: Busca la coincidencia sólo al final de la línea.**

Para dar una expresión regular se pasan en '/'. Y para aplicarla a una variable se usa el operador '=~'.

Si anteponemos \ (back slash) a cualquier comodín nos referimos al carácter mismo.

Ejemplos :
```
$a="hola mundo";
$e="chao";
$b= ($a =~ /^he/); # $b vale true = 1
$c= ( "chao" =~ /ah/ ) ; # $c=0
$d= ( "$e" =~ /ah/ ); # $d=0
if ( $a=~ /o$/ )
    { print "Correcto\n"; } # imprime Correcto
```

El operador '=~' tiene una opción que lo que hace es como el *sed*, reemplazar un trozo de texto por otro, esa opción es la 's' :

El formato es $x =~ s/expant/expnueva/ ;

Ej:
```
$cual="este es un test";
$cual =~ s/test/prueba/; #$cual="este es un prueba"
```

Más detalles ver el anexo: ExpReg

Split() y Join()

- *Split*() lo que hace es dado un separador, separa un string en un arreglo :

  Ej: `$linea="mvargas::117:10:Juan";`

  `@a=split(/:/,$line); # @a=("mvargas","","117","10","Juan")`

- *Join*() hace lo inverso de Split(), dado un arreglo lo junta en un string, separado por un delimitadorespecificado.

  Ej:`@campos=(1,2,"hola");`

  `$salida=join(".",@campos); # $salida="1.2.hola"`

Capítulo V: Interfaz con Unix

Al igual que en c-shell o C dentro de un programa en C se puede ejecutar un comando UNIX, o un programa propio en cualquier lenguaje, esto se hace con system(), y de otra forma con '`'(comilla invertida).

```
Ej: #!/usr/local/bin/perl
        print "En el archivo listado dejaremos la lista de archivos\n";
    system("ls -l > listado");
Ej: #!/usr/local/bin/perl
    $num=`wc -c listado`; # En la var. $num deja el número de caracteres
                          #que tiene el archivo listado
```

Obs.: Es importante decir que dentro de '`' y system son expandidos los valores de las variables.

```
Ej:     $a="direct1";
        @a=`ls -la $a`;
```

Aquí introducimos un nuevo concepto, resulta que después de la ejecución de estas líneas en el arreglo @a quedan guardadas las líneas de salida del comando ls -ladirect1.

Existe otra forma de ejecutar comandos Unix, es con el open, el cual también tiene las funcionalidadesdel *popen* de C, es decir podemos abrir para escribir o leer, un comando:

```
Ej:     open(FD,"ls |");
        @archivos=;
        close(FD);
        # lo que hizo fue guardar todo lo que retorno el ls
        # en el arreglo @archivos, un nombre de archivo en
        # cada casilla del arreglo
```

Capítulo VI: Subrutinas

Al igual que la mayoría de los lenguajes de Programación, **PERL** soporta subrutinas, también conocidas como procedimientos o funciones.

Con ejemplos, veremos como se construyen.

Ejemplo 1

```
sub suma {
 local($x,$y)=@_; # En @_ se reciben los parámetros (es por valor)
 return($x+$y);
}
$f=8;
$c=&suma(4,$f);   # Otra opción sería: $c=&suma(4,$f,'hola'); no se ve
                  # afectado
# $c=12
```

Ejemplo 2

```
sub duplica() {
  $x*=2;
}
$x=8;
&duplicar();    # $x=16, ya que no lo declaramos como
                # local en la subrutina => duplica a x
```

Como vimos el paso de parámetros es por referencia, pero al igual que en C, los parámetros se pueden modificar, vía punteros:

Ejemplo3

```
$st='uno:1,dos:2,tres:3';
&crea($st,*a); # cuando la función termina: $a{'uno'}=1, $a{'dos'}=2,
$a{'tres'}=3
sub crea {
local($pal,*g)=@_;
local($x,$a,$b);
local(@aux);
@aux=split(/,/,$pal);
for($x=0;$x<=$#aux;$x++)
   {
   ($a,$b)=split(/:/,$aux[$x]);
   $g{$a}=$b;
   }
}
```

Capítulo VII: CGI y Perl

CGI (CGI, Interfaz de pasarela común) es como la puerta de acceso que hay entre una página Web y el servidor de Internet donde la página reside. En adelante vamos a ver por qué Perl es importante en la programación CGI.

Sabemos que un explorador, navegador o browser realiza un importante trabajo en la presentación de una página Web. Pero hay que tener en cuenta también el trabajo que se realiza en la parte del servidor, porque éste da respuesta en cada momento a las peticiones que realizan los propios browsers, por ejemplo, cuando piden una página nueva, hay que buscarla, prepararla y empaquetarla para su envío y finalmente enviarla a su destino.

Por otro lado, cuando un explorador desea algo más que otra página Web o un gráfico desde el servidor, la solicitud va al CGI para poder ser procesada. Por ejemplo, las solicitudes de búsqueda de texto, el procesamiento de datos, los informes de datos, y otros procesamientos de datos interactivos necesitan un manejo especial.

Los programas CGI realizan el procesamiento, la construcción de los archivos, y el acceso a las bases de datos para las solicitudes especiales.

HTML y Perl cruzan sus caminos a través del CGI. Las solicitudes de procesamiento fluyen desde los documentos HTML a través del CGI, donde los programas Perl reciben las solicitudes y manejan la información. La respuesta usual proporcionada por un programa CGI de Perl toma la forma de otra página HTML, que se construye frecuentemente sobre la marcha para cumplir las necesidades específicas del solicitante.

Los documentos HTML y los programas CGI para un sitio de la Web están localizados usualmente en el mismo servidor. Un ordenador puede tener un número de cuentas de sitios de la Web, cada una con su propio conjunto de directorios independientes en el servidor. Puesto que los programas CGI deben acceder algunas veces a los archivos del servidor y a otros recursos, es de la mayor prioridad para los administradores del sistema de la Web aplicar medidas de seguridad con reglas y procedimientos para la transferencia de los archivos y para la utilización de programas CGI.

Capitulo VII: PERL y los formularios

Funcionamiento de un formulario

La idea básica detrás de un formulario es sencilla: pida información al visitante, obtenga la información, después haga algo con la información. La parte delicada de tratar con los formularios se maneja entre bastidores por el código de Perl.

¿Cómo van los datos de un formulario al servidor?.

Después de que el visitante rellene un formulario, el explorador envía los datos para ser procesados o almacenados. Típicamente, la corriente de los datos del formulario es alimentada en un programa que cambia todos los datos por un formato más legible. El programa de procesamiento puede ser un URL (Universal Resource Locator o Localizador Universal de Recursos) situado en cualquier parte de Internet.

El programa de procesamiento de los datos es usualmente un script de Perl, un script de Shell de UNIX, o un programa compilado residente en el servidor que maneja sus páginas Web.

Los datos procedentes de un formulario fluyen en una corriente que utiliza un formato especialllamado **URL encoding** (codificación URL). El término procede del hecho de que los datos tienen que ser transformados en un formato que pueda viajar por Internet, lo que originariamente quería decir texto ASCII puro y sencillo que parecía como la dirección típica de la página Web; letras y números.

Desde luego, codificar los datos es solamente la mitad de la batalla, traducir el URL, los datos codificados en texto legible en el otro lado de la transmisión es la otra mitad, y una buena tarea de Perl.

Cada campo tiene un atributo NAME (nombre, de forma que puede saberse de dónde proceden los datos) y un atributo VALUE (valor, contenido del campo). La cadena enviada por el explorador empareja cada NAME del formulario con su VALUE y los conecta con un signo igual: NAME=VALUE. La pareja NAME/VALUE es la construcción básica de datos de los formularios y de los programas de procesamiento de formularios. Los informes CGI más sencillos están realizados con listas de parejas NAME/VALUE.

Métodos de envío: GET y POST.

El visitante envía al servidor los datos del formulario utilizando uno de los dos métodos: GET o POST.

- *El método GET*, el explorador empaqueta los datos del formulario y los agrega al final de una solicitud de aspecto normal para un URL. Por ejemplo, supongamos que en WWW.arrakis.es en el directorio /cgi-bin tenemos el programa Perl de búsqueda (busca.pl) y que nuestro formulario tiene un campo llamado Buscar con el valor "Blas Infante", al hacer clic sobre el botón Entregar (Submit) se enviará lo siguiente:

www.arrakis.es/cgi-bin/busca.pl?buscar=Blas+Infante

- *El método POST* codificará los datos de la misma forma, pero los envía directamente al programa CGI a través de STDIN. El método POST usa la variable de entorno CONTENT_LENGHT (longitud del contenido) para decirle al servidor cuántos bytes debe leer desde STDIN. La corriente de datos puede ser tan larga como se necesite, cosa que no ocurre con el método GET que dependiendo del servidor se permitirán cadenas más o menos cortas. Está limitación h ace que POST sea el método más utilizado.

Más detalles sobre formulario: Más sobre formularios HTML

Ejemplo de formulario: Formulario

Anexos

Uso de comillas

Comillas simples :

- **Ej:**`'hola' # h,o,l,a`
- `'don\'t' # d,o,n,',t`
- `'hola\n' # h,o,l,a,\,n`

Comillas dobles :

- **Ej:**`"hola mundo\n" # hola mundo[enter]`
- `"cola\tsprite" # cola[tab]sprite`

En los strings con comillas dobles, se reconocen caracteres especiales como `\n`, `\t`, etc. mientras que con comillas simples no, pero tiene la ventaja de poder escribirse literalmente lo que se pone entre ellos.

Más sobre arreglos

los subíndices positivos recorren los elementos al derecho

```
@a = ('a'..'e');    # $a[0] es 'a'  $a[4] es 'e'
```

los subíndices negativos recorren los elementos al revez

```
@a = ('a'..'e');    # $a[-1] es 'e'   $a[-5] es 'a'
```

un arreglo en **contexto** escalar retorna el número de elementos

```
@a = ('a'..'e');
$n = @a;  #  aquí $n es 5 equivalente a $n = @#a;
```

Lo del contexto es una característica de **PERL**.

PERL evalúa una expresión según el uso que se piensa dar a la expresión: contexto escalar o contexto de lista subarreglos:

```
@a = ('a'..'z');
@b = @a[3, 0];  #  @b = ('d', 'a');
@c = @a[2..5];   #  @c = ('c', 'd', 'e', 'f' );
```

Es posible colocar una lista de variables escalares a la izquierda del igual

```
($a, $b ) = @x  #  $a queda con el valor de $x[0]
                # $b queda con el valor de $x[1]
```

Es posible colocar un subarreglo a la izquierda del igual

```
@a[3, 0]  = @a[0, 3]; # intercambia los elementos 0 y 3
```

Operaciones sobre arreglos

- ***Push y Pop***

Una común utilización de los arreglos es como stacks, donde los nuevos valores son agregados y borrados por el lado derecho del arreglo. *Push()* es utilizado para agregar elementos y *Pop*() para sacar.

```
Ej: @lista=(1,2,3);
    push(@lista,5); # @lista=(1,2,3,5)
    $a=pop(@lista); # $a=5, @a=(1,2,3)
```

- ***Shift* y *Unshift*()**

Al igual que *Pop* y *Push* estos sacan y meten elementos en un arreglo, pero lo hacen por el lado izquierdo.

```
Ej: @a=(1,2,3);
    unshift(@a,0);    # @a=(0,1,2,3);
    $x=shift(@a);     # $x=0 , @a=(1,2,3)
    unshift(@a,5,6,7) # @a=(5,6,7,1,2,3)
```

- **El operador *reverse***

Reverse() invierte el orden de los elementos de un arreglo, retornando la lista invertida.

```
Ej: @a=(1,2,3);
    @b=reverse(@a); # @b=(3,2,1)
```

- ***Chop***

Chop() trabaja igual que en variables escalares, le saca el último carácter a cada elemento del arreglo. Se usa para eliminar el retorno de carro de la entrada estándar.

```
Ej: @a=("hola","mundo\n","felices dias");
    chop(@a);                # @a=("hol","mundo","felices dia")
```

- ***Splice***

Splice permite extraer un subarreglo y modificar a la vez el arreglo original...

```
@a = ( 'a'..'e');
@b = splice ( @a, 1, 2);
        # @b queda con 2 elementos de @a: $a[1] y  $a[2];
        #       ( 'b', 'c')
        #  @a queda sin esos 2 elementos:  ( 'a', 'd', 'e' );
```

Con un argumento más, sirve para modificar el arreglo original

```
@a = ( 'a'..'e');
@b = ( 1..3);
splice ( @a, 2, 1, @b);
# @a queda con ('a', 'b', 1, 2, 3, 'd', 'e');
# se cambio 'c' por (1, 2, 3)
```

También se puede modificar sin eliminar ningún elemento.

```
@a = ( 'a'..'e');
@b = ( 1..3);
splice ( @a, 2, 0, @b);
# @a queda con ('a', 'b', 1,2,3, c', 'd', 'e');
```

Más sobre hashes

Para que sea más claro la asignación a un *hash* se pueden remplazar algunas comas por"=>"...

```
%x = ( 'ope' => 'ver',  'nit' => 890900285 );
```

Así se ven mejor las parejas clave-valor

En cualquier momento se puede agregar un elemento a un *hash*

```
$a{fac}=3456;
```

La función **delete** sirve para borrar un elemento

```
delete $a{ope};
```

La función **keys** crea un arreglo con las claves de un *hash*

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

La función **values** devuelve un arreglo con los valores del *hash*

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

La función **exists** prueba si existe la clave en el *hash*

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

ExpReg.:

ExpReg es una abreviatura de expresión regular.

Las ExpReg se utilizan en 2 clases de expresiones

1. match del patrón en un string...

Aquí la ExpReg es una expresión lógica.

Devuelve verdad si el string contiene un patrón

```
$a = "abcdef";
$a =~ /bc/; #  es verdadero
$a =~ /ba/; #  es falso
```

"=~" se llama el operador de **bind**...

"!~" es la negación de la expresión

```
$a = "abcdef";
$a !~ /bc/; #  es falso
$a !~ /ba/; #  es verdadero
```

2. substitución s///

Ya fue visto anteriormente su explicación.

***Cuando el escalar es $_ se omite $_ y =~...

```
$_ = "abcdef";
/bc/; #  es verdadero
s/cd//;
print;  # imprime: abef
```

cuantificadores

{3,5} : mínimo 3 y máximo 5 del carácter anterior

{3,} : mínimo 3 del carácter anterior

{,5} : máximo 5 del carácter anterior

clases de caracteres comunes

\s : un espacio en blanco , tabulador o salto de línea

\S : un carácter no blanco, no tabulador y no salto de línea

\d : un digito

\D : un no digito

\w : un carácter de palabra: digito letra o _

\W : un carácter que no es de palabra

```
$_ = "d15";
/\d+$/ ; # es verdadero
```

clases de caracteres a la medida

[abcef] : uno de esas 5 letras

[a-f] : lo mismo que el anterior

[0-9] : es lo mismo que \d

[\t \n] : es lo mismo que \s

[a-zA-Z_] : es lo mismo que \w

```
@a =  ( 1..10);
foreach ( @a )
{
      /^[1-3]/
      and
      print "$_:";            # imprime: 1:2:3:10:
}
```

Más sobre ExpReg

Memoria de matchs.

Los paréntesis se usan para almacenar los matchs en las variables $1, $2, $3, hasta $9.

```
$_ = "1995 Renault azul";
s/^(\w+)\s+(\w+)/$2 $1/; # Intercambia 1995 y Renault
print $_;                # Imprime: renault 1995 azul
print $1;                # Imprime: 1995
```

También es posible sacar los matchs a un arreglo

```
$_ = "1995 renault azul";
@a = /^(\w+)\s+(\w+)/;
print "@a"; # imprime: 1995 renault
```

Las ExpReg. tienen también opciones:

/g : indica que haga varios "match's"

```
$_ = "f1=abc test=on";
s/=/ / ;    #  $_ queda con "f1 abc test=on"

$_ = "f1=abc test=on";
s/=/ /g ;   #  $_ queda con "f1 abc test on"

$_ = "1995 Renault azul";
@a = /^(\w+)/g; # @a queda con 3 elementos
```

/i : ignore mayúsculas y minúsculas

```
$_ = "Francisco francisco";
s/francisco/pacho/ig;   #  $_ queda con "pacho pacho"
```

s/ / /e : ejecuta la segunda expresión y su valor lo utiliza

Para remplazar el patrón.

```
$_ = "largo= 15";
s/(\d+)/$1 * 4/e;
print;      #  Imprime: largo= 60
```

el operador tr se usa para traducir caracteres.

Tiene un parecido con la substitución en ExpReg

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

%x = split / /, $a; # $x{f1}   queda con "abc"
             # $x{test} queda con "on";
```

Paquetes

Un paquete es un espacio de nombres.

Los espacios de nombres permiten que nosotros utilicemos otros código sin que las variables se confundan con las variables utilizadas en ambos códigos.

```
package C110;     # estamos en el espacio de nombres C110
$a = 5;            # variable del paquete C110
fun1               # función del paquete C110
{
      print  "$a\n";
}

package D110;   # ahora estamos en el espacio de nombres D110
                # ...salimos del paquete C110
$a = 7;         # esta $a es del paquete D110
print  $a;      # imprime 7


print  $C110::a;          # imprime 5
      # fijarse como accesamos el espacio de nombres C110...
      # mirar la sintaxis del $ y los ::
```

Existen tres formas distintas de llamar a la función de los paquetes.

```
Ej: C110::fun1;       # llama a fun1 de C110...imprime: 5
    fun1 C110;        # llama a fun1 de C110...imprime: 5
    C110->fun1;       # llama a fun1 de C110...imprime: 5
```

OBS.: Cuando no usamos "package" estamos trabajando en el espacio de nombres "main".

Como un paquete generalmente se hace para ser reutilizado muchas veces se guarda en un archivo librería de extensión .pl como por ejemplo, cgilib.pl y los programas que lo quieren usar lo invocan con requiere: **requiere "cgilib.pl";**

La función "requiere" lee el archivo "cgilib.pl" si este no ha sido leído antes. El archivo no tiene que tener "package" pero si debe devolver verdadero, o sea, que lo mejor es que termine con: **return 1**

Las librerías ya no se usan tanto, porque se usan los objetos que en Perl se implementan con módulos.

OBS.: Parámetro adicional que reciben las funciones de un paquete.

```
package  C110;
sub fun2
{
      print  "fun2 recibio @_\n";
```

```
}

package D110;
C110::fun2("xyz");  # llama a fun2...imprime: fun2 recibio xyz
```

Esta forma de llamar funciones de paquete no se utiliza usualmente porque, como veremos enseguida, las funciones de paquete se escriben para recibir un parámetro adicional...

```
C110->fun2("xyz");  # llama a fun2...imprime: fun2 recibio C110 xyz
fun2 C110("xyz");   # equivalente al anterior...  C110->fun2 ("xyz")
```

Observar que cuando se llama con C110->fun2, fun2 recibe un parámetro adicional, el nombre del paquete, "C110".

Más adelante veremos otra forma mas de llamar fun2.

$r->fun2()... donde $r es una referencia a un objeto C110. En este caso el parámetro adicional que recibe fun2 es la referencia $r.

Un módulo es un paquete en un archivo de su mismo nombre y extension .pm. Los nombres de los módulos empiezan por mayúscula.

Ej.: el módulo CARRO debe estar en el archivo CARRO.pm

Para utilizar un módulo en un programa se utiliza la función "use" es como un "requiere", pero que además ejecuta una función del módulo llamada import... que vamos a ignorar por ahora.

El módulo CGI

este módulo se usa para leer los campos de una forma enviada desde el Netscape a nuestro programa Perl... a través de un servidor http como el Apache

```
# programa vt6100.pl
use CGI;
$q =  new CGI;
# $q es una referencia tipo CGI...
# o mas simplemente un objeto CGI
```

```
$nom = $q->param('nom');
$art = $q->param('art');
$can = $q->param('can');
# param es una función de CGI que nos da el valor de un
# campo de la forma...  'nom' 'art'  'can'  son nombres
# de campos de la forma en una página html que muy
# posiblemente salio de nuestro servidor http

# el pgma continúa revisando el pedido, aceptándolo si esta ok
# y finalmente,  dándole al cliente (con print por supuesto)
# una respuesta adecuada...
```

la historia completa es esta:

un cliente pide nuestra forma de pedidos... digamos vt6100.html... vt6100.html es algo como esto:

```
<h1>pedido</h1>
<form method=post action=http://epq.com.co/cgi-bin/vt6100.pl>
<p>nombre <input name=nom size=30>
<p>codigo del articulo <input name=art size=8>
<p>cantidad<input name=can size=10>
<p><input type=submit value=enviar>
</form>
```

una vez que el cliente llena la forma y da click en "enviar" el Netscape del ciente envia los campos de la forma al servidor...

el servidor ejecuta el programa vt6100.pl (el programa Perl que se habló arriba) y le pasa los campos de la forma...

el programa vt6000.pl lee los campos de la forma usando el módulo CGI como se explicó arriba

El módulo DBI

Este módulo se usa para accesar una base de datos como ORACLE...

ejemplo de una operación de consulta

```
use DBI;
use CGI;

$qry = new CGI;
$dbh = DBI->connect('dbi:Oracle:', 'useru', 'clave' );   # $dbh es un objeto de una clase
$sth = $dbh->prepare ("select codemp, nomemp, vinemp from emp where ciaemp=?");
        # esto hace que Oracle compile el "select"...
        # $sth es un objeto de otra clase

        $cia = $qry->param ( "cia");        # lea $cia de la forma del cliente

        $sth->execute ( $cia ) ;            # esto crea el cursor

        while  ( ($cod, $nom, $vin ) = $sth->fetchrow_array )  # aqui se lee el cursor
        {
                printf  "%5s %30s %3s", $cod, $nom, $vin ;
        }
        $sth->finish;     # esto cierra el cursor...
                          # parece que no es muy necesario... yo ya no los uso
```

ejemplo de una operación de actualización

```
        use DBI;
        $dbh = DBI->connect('dbi:Oracle:', 'usuario', 'clave' );
        $sth = $dbh->prepare ("delete  from emp  where ciaemp=?");
        $sth->execute ( $cia ) ;
        $dbh->commmit;
```

El *prepare* y *execute* se pueden hacer simultáneamente con do...
pero "prepare" permite utilizar "placeholders" (las interrogaciones)
que se llenan en el "execute"...

Información sobre el resultado de un prepare o execute

```
        $DBI::err                  # número del error... análogo al sqlcode...
```

```
                                # es falso (no definido) si no hay error

        $DBI::errstr            # texto del error
                                # es falso (no definido) si no hay error
```

Información que se puede obtener después del execute

```
        $DBI::rows              # nro de filas afectadas... puede ser 0
                                # no sirve en SELECT
```

Información que se puede obtener después del execute de un select

```
        $sth->{NAME}            # referencia al arreglo de los nombres de la columnas
                                # se puede usar aunque no se seleccione ninguna fila

        $sth->{TYPE}            # referencia al arreglo de los tipos de los campos

        $sth->{SCALE}           # referencia al arreglo de longitudes de los campos
```

El módulo LWP

este módulo se usa para accesar servicios de internet como poner
correo o leer una página, sin usar el Netscape

el módulo LWP maneja varios objetos:

LWP::UserAgent : el que se conecta al servidor
HTTP::Request : lo que se pide al servidor
HTTP::Response : lo que se recibe del servidor

ejemplo para enviar correo:

```
use LWP;

# 1. crear un  agente ( user-agent )
$wuag = new LWP::UserAgent;

# 2. crear una peticion ( request )
$wreq = new HTTP::Request (
        POST =>  'mailto:cjara@epq.com.co' );
```

```
# 3. llenar el encabezado (header) de la peticion
$wreq->header (
        Subject => 'prueba de LWP' ,
        From    => 'alguien' );

# 4. llenar el contenido de la peticion
$wreq->content ( "me gusta este tutorial");

# 5. enviar la peticion con el agente
#    y obtener una respuesta  "
$wres = $wuag->request ( $wreq );

# 6. examinar la respuesta
$wres->is_success ? print  "exito \n": print  "error \n";
```

Variables de entorno (CGI)

Las funciones y los procesos de un servidor pasan datos entre sí a través de variables de entorno. Estas variables actúan de forma muy parecida a la de una fila de buzones de correo. Los procesos y las funciones pasan información a esas variables y después vuelven a otra parte a realizar su tarea.

Las variables CGI de mayor importancia generalmente para comprender el proceso de comunicación explorador/servidor se describen en la tabla siguiente:

QUERY_STRING - Datos de entrada que se agregan a URL para un método GET.
REQUEST_METHOD - Expresa el método usado: GET o POST.
CONTENT_LENGHT - Número de bytes de la corriente de datos a leer para una solicitud con el método POST.

Más sobre formularios HTML

<FORM>...</FORM>.Define el comienzo y final de un formulario.
ACTION="URL". Es el nombre del archivo del script o del programa que manejará los datos desde un formulario.
METHOD="GET" o "POST". Así se indica cómo se mueven los datos desde el formulario hasta el script que los maneja.
Ejemplo: <FORM ACTION="cgi-bin/formecho.pl/"METHOD="POST">

Elementos de un formulario

<INPUT> Define campos de un formulario. Tiene un atributo NAME para asignar un nombre al campo y un atributo TYPE para definir el tipo de entrada del campo:
TYPE="TEXT" Texto.
TYPE="PASSWORD" Contraseña.
TYPE="CHECKBOX" Casilla de verificación.
TYPE="RADIO" Botón de opción.
TYPE="SUBMIT" Botón de envío de datos.
TYPE="RESET" Botón de borrado de formulario.
TYPE="HIDDEN" Campo oculto.

Otros atributos de INPUT son:

VALUE="Valor" Con las casillas de verificación y los botones de opción especifica la opción seleccionada. En el botón SUBMIT personaliza la etiqueta del botón.

SIZE="Tamaño" Longitud en caracteres del cuadro de entrada para los tipos TEXT y PASSWORD.

MAXLEGHT="Tamaño" Longitud en caracteres máxima del campo para los tipos TEXT y PASSWORD.

<SELECT></SELECT> Define una lista de elementos para selección.

SIZE="1,2" 1=Cuadro emergente 2=Cuadro desplazable >2=Cuadro desplazable con entrada en blanco.

MULTIPLE Permite seleccionar más de una opción al mismo tiempo.

<OPTION> Etiqueta para identificar el texto de un elemento de una lista de opciones.

<OPTION SELECTED> Opción por defecto.

<TEXTAREA></TEXTAREA> Define un cuadro de texto de varias líneas.

NAME="Nombre" Nombre del campo.

ROWS="Filas" Número de filas.

COLS="Columnas" Número de columnas.

Ejemplo Formulario

```
<HTML>
<HEAD>
<TITLE>Incidencias de los usuarios de PCs.</TITLE>
<BODY BGCOLOR="#35afa1" TEXT="#000000">
<H1>Incidencias de los usuarios de PCs.</H1>
<H3>Si tiene problemas o n ecesita de los servicios del departamento</H3><P>
<H3>de micro rellene y envíe este formulario.</H3>
<! siguiente es el lugar donde buscar el formulario>
<FORM ACTION="./cgi-bin/avisos.pl" METHOD="GET">
<PRE>
<INPUT TYPE="HIDDEN" NAME="AAFormID" VALUE="Avisos">
Nombre: <INPUT TYPE="TEXT" NAME="Nombre" SIZE="20" MAXLEGHT="20"><P>
Centro Directivo: <SELECT NAME="Centro" SIZE="0">
<OPTION VALUE="Gabinete Consejera"> Gabinete Consejera
<OPTION VALUE="Viceconsejeria"> Viceconsejeria
<OPTION VALUE="S.G.T."> S.G.T.
<OPTION VALUE="S.G.E."> S.G.E.
<OPTIONVALUE="Presupuestos">Presupuestos
<OPTION VALUE="Patrimonio"> Patrimonio
<OPTION VALUE="Intervención General"> Intervención General
<OPTION VALUE="Tesorería"> Tesorería
<OPTION VALUE="Fondos Europeos"> Fondos Europeos
```

```
<OPTION VALUE="Tributos"> Tributos
<OPTION VALUE="Relacionesfinancieras">RelacionesFinancieras
<OPTION VALUE="Informática"> Informática
<OPTION VALUE="Otro departamento"> Otro departamento
</SELECT><P>
Telefono: <INPUT TYPE="TEXT" NAME="Teléfono" SIZE="5" MAXLEGHT="5"><P>
Incidencia: <INPUT TYPE="RADIO" NAME="Tipo" VALUE="Avería" CHECKED> Avería
<INPUT TYPE="RADIO" NAME="Tipo" VALUE="Programa"> Programa
<INPUT TYPE="RADIO" NAME="Tipo" VALUE="Configuración"> Configuración
<INPUT TYPE="RADIO" NAME="Tipo" VALUE="Información" > Información
<INPUT TYPE="RADIO" NAME="Tipo" VALUE="Otro" > Otros<P>
Descripción: <INPUT TYPE="TEXT" NAME="Incidencia" SIZE="40" MAXLEGHT="40"><P>
<INPUT TYPE="SUBMIT" VALUE="Enviar al Servicio de Informática">
<INPUT TYPE="RESET" VALUE="Borrar">
</PRE>
</FORM>
</BODY>
</HTML>
```

Ejemplo de analizador de cadenas codificadas URL

Suponiendo que en el formulario anterior hubiéramos tecleado los siguientes datos:

Nombre = "Manuel"
Centro = "Informatica"
Telefono ="64022"
Tipo = "Averia"
Incidencia = "No arranca PC"

La cadena URL que se enviará al servidor después de pulsar sobre el botón "Enviar al servicio de informática" será:

?AAFormID=Avisos&Nombre=Manuel&Centro=Informatica&Telefono=64022
&Tipo=Averia&Incidencia=No%20arranca%20PC

Con este envío y ayudados de un programa en Perl podemos analizar la cadena URL y crear un formulario que nos presente los datos de una forma más inteligible, por ejemplo en HTML para ser visualizado por un navegador (formula1.pl):

```
# Analizador de cadenas codificadas URL
# Ejecutar perl formula1.pl formula1.txt >formula2.htm
print "<HTML>\n<HEAD><TITLE>Form Data</TITLE></HEAD>\n";
print "<BODY>\n<H3>Form Data</H3>\n";
while (<>) {
if (/AAFormID=/) {
chop();
@url = split(/&/);
foreach (@url) {
tr/+/ /;
s/=/ = /;
```

```
s/%(..)/pack("C",hex($1))/ge;
print "$_<BR>\n";
}
print "<P><HR>\n";
}
}
print "</BODY>\n</HTML>\n";
close;
```

Conclusiones

Con este trabajo pude aprender un nuevo lenguaje de programación, Perl, el cual es muy útil para interactuar con una pagina Web, aunque se deba tener permisos “especiales”, como es usar CGI por estar dentro de una red de computadores.

Este nuevo lenguaje se utiliza mucho en los servidores de paginas Web donde se interactúa con el usuario.

Aunque no se pudo abarcar más sobre este interesante lenguaje, espero que haya valido el esfuerzo por aprender este lenguaje y su utilización en la interacción cliente / pagina Web.

Bibliografía

- Teach yorself Perl 5 in 21 Days **autor:** David Till
  Editoral Sams Publishing
- Creating Cool WEB pages with Perl **autor:** Jerry Muelver
  Editorial IDG Books WorldWide
- Curso de Perl **autor:** Mauricio A. Vasquez M.
  Disponible en: http://www.geocities.com/SiliconValley/Station/8266/perl/

Paginas Web's:

http://epq.com.co/~cjara/perl/tutorial.html

http://www.geocities.com/SunsetStrip/Backstage/6023/CGI_perl_01.html

http://usuarios.lycos.es/asualam/perl/

http://members.tripod.com/~MoisesRBB/perl.html

http://www.tjhsst.edu/~dhyatt/perl/

INDICE