Índice

1. PERL, "Practical Extraction and Report Language"

1.1. Introducción

Perl es una herramienta, en general, común entre desarrolladores de sitios Web, y administradores de sistemas.

Posee una sintaxis muy clara, en cuanto a estructuras de control se refiere. No obstante, por el hecho de ser un lenguaje muy rico, hay una gran cantidad de cosas que aprender acerca de las facilidades que ofrece. Esto no implica que hay que ser un erudito en sintaxis de Perl para utilizarlo, se puede empezar a programar bajo este lenguaje de scripting con conocimientos básicos, y algunas referencias. En general, a medida que se avance sobre la dificultad del problema, se irán necesitando construcciones más complejas, que paradojicamente, con Perl se simplifican bastante respecto de otros lenguajes.

Originalmente diseñado para procesar archivos de texto, se volvio un lenguaje de proposito general, con distintos entornos de desarrollo, debuguers, bibliotecas, etc. al punto que uno puede desarrollar aplicaciones que van de las simples (o muy complicadas) tareas de administración de servidores, herramientas de Networking, aplicaciones via Web mediante el uso del módulo CGI.pm, hasta programas con interfaces gráficas, gracias al módulo Perl/Tk.

2. El interprete de Perl

Para que un script que hayamos hecho, pueda ejecutarse con exito, debe pasar por el interprete del lenguaje, de otra manera, no sera posible su ejecución.

En general, todo script de Perl lleva, antes que cualquier otra instrucción la línea **#!/usr/bin/perl**. Esta, le indica al kernel que el script debe ser interpretado por el *Interprete de Perl*. La ruta al interprete de Perl, variara según el Sistema Operativo, y tipo de distribución para los sistemas GNU/Linux. Es posible además, especificar opciones que modificaran el comportamiento del interprete, haciendo explicitos ciertos warnings, habilitar el debuguer, etc. Dentro de las más utilizadas (personalmente), podemos encontrar:

- -c Solamente verifica la sintaxis, no ejecuta ninguna instrucción
- -d Habilita el Debuguer de Perl
- -e command Es utilizado para ingresar una o más instrucciones de codigo en la línea de comandos
- -v Imprime la versión de Perl que se esta utilizando
- -V Configuración de Perl, y el array @INC
- -w Imprime advertencias

3. Sintaxis

3.1. Estructura del Programa

Escencialmente, toda instrucción debe terminar con un punto y coma (;), exeptuando los comentarios y las estructuras de control de flujo.

En cuanto a las declaraciones, solamente las funciones (más conocidas en el ambiente como *subrutinas*) deben ser declaradas (en realidad tambien los formatos de reporte, cosa que escapa al curso), el resto de las variables son inicializadas a **NULL** o **0**, según el contexto, a menos que sean definidas en forma explicita por algun tipo de asignación.

Los comentarios se inician con el signo numeral (#), y abarcan desde su posición hasta el final de línea.

3.2. Tipos de Variables

En general, en los lenguaje de scripting, no es necesario definir el tipo de una variable, desde el punto de vista de si son de tipo *char*, *int*, *float*, etc, sino que estan fijados por el contexto de la misma.

En Perl, existen básicamente tres tipos de dato, que son los **escalares**, **arrays** y **hashes**.

3.2.1. Tipo Escalar

Están precedidos por el signo $. Son escencialmente, variables, y pueden tener un valor numérico, de cadena o referencia (un escalar que apunta a otra porción de información, de cualquier tipo - si, punteros -).

Si un string es asignado a una variable de tipo escalar, cuando por el contexto se esperaba un número, Perl se encarga de realizar la conversión siguiendo una serie de reglas intuitivas. Para asignar un valor a una variable de tipo escalar, simplemente podemos hacer:

```
$variable = 1;
```

3.2.2. Tipo Array

Precedidos por el signo @, son listas de escalares ordenadas, a las cuales se puede acceder por un indice entero, el cual comienza en el 0. Definimos una lista de la siguiente forma:

```
@array = ("offset 1", 2, "1e3", 2.34);
```

Para hacer referencia a alguno de los elementos dentro del array, simplemente se antepone el signo $ (ya que un array es una lista ordenada de *escalares*) seguido por el nombre del array, y finalmente entre corchetes (`[]`) el offset, o posición.

```
print("$array[2]"); # Imprime "1e3"
```

3.2.3. Tipo Hash

Un hash es un array asociativo, más exactamente, una lista no ordenada de pares de *llaves* (keys), y *valores*(values). En Perl, estan precedidos por el signo % cuando son definidos en forma general. Para definir un hash, la sintaxis es la siguiente:

```
%hash = ("Nombre",         "Luis",
         "Apellido",       "Rojas",
         "Color Preferid", "Azul");
```

De esta forma, se asignan de a pares los valores *key/value*, con lo que tendriamos un hash don tres posiciones. Aunque esta notación es valida, suele ser mucho más intuitiva la siguiente:

```
%hash = ("Nombre"         => "Luis",
         "Apellido"       => "Rojas",
         "Color Preferid" => "Azul");
```

De esta forma, se deja explicitamente marcado cual el valor del identificador (y viceversa). Para referirse a un elemento, anteponemos el signo $ (porque otra vez, es una lista asociativa, y recuerden que los arrays, son listas de escalares), seguido por el nombre de hash, y finalmente, entre llaves ({}) y comillas, el identificador.

```
print($hash{'Nombre'}); # Imprime "Apellido"
```

3.3. Tipos y Contextos

Toda operación invocada en un script de Perl, es evaluada en un contexto especifico. Existen dos tipos de contexto, el escalar y el de lista. Los operadores pueden identificar el contexto en el cual se realiza una operación, y actuar en función de ello. Por ejemplo, la función **localtime()** devuelve un array de nueve elementos en un contexto de lista:

```
# Devuelve los valores
# ($sec, $min, $hour, $mday, $mon, $year, $wday, $yday, $isdst)
@actual_time = localtime();
```

Pero en un contexto escalar, devuelve la fecha actual, en formato de string

```
# Devuelve "Tue Jul 15 02:28:16 2003"
$actual = localtime();
```

Otro ejemplo de contextos, podria ser

```
# $num posee la cantidad de elementos de @array
$num = @array;

# $num posee el indice del último elemento de @array
# (uno menos que la cantidad total)
$num = $#array;
```

Esta sintaxis, es util para conocer el número de elemntos disponibles en el array *@array*.

3.4. Referencias

Una referencia en Perl, es fundamentalmente un tipo de dato que apunta a una porción de información o codigo (simplemente, un puntero). Como es de suponer, contiene la dirección de la información, así como el tipo de la última.

Es básicamente un escalar, y puede utilizarse en cualquier lugar en donde puede utilizarse este, como elemento de un array o hash (no como índice).

La creación de referencias es muy simple, solamente se antepone el caracter al tipo al cual se quiere referenciar. Ejemplos de esto, son:

```
$a = "esto va a ser una referencia";
@array = ("elemento1", 5, ""último elemento);
%hash  = ("Nombre"   => "Emiliano",
          "Apellido" => "Castagnari");

sub foo
  { print("Se paso el valor $_[0]\n"); }

$ref_a = \$a;
$ref_array = \@array;
$ref_hash  = \%hash;
$ref_sub   = \&foo;
```

Para desreferenciar(esto es, obtener la información) una variable, existen unos cuantos metodos:

```
# Asigna el string a $a;
$$ref_a = "nuevo contenido";

# Los siguientes métodos son válidos para arrays:
$$ref_array[1]   = "posición 1";
$ref_array->[1]  = "posición 1";
${$ref_array[1]} = "posición 1";

# Los mismos métodos son válidos para hashes:
$$ref_hash{'Nombre'}   = "Maximiliano";
$ref_hash->{'Nombre'}  = "Margarita";
${$ref_hash}{'Nombre'} = "Martin";

# Desreferencia funciones:
&$ref_foo("Parámetro");
$ref_foo->("Parámetro");
&{$ref_foo}("Parámetro");
```

3.5. Funciones

Las funciones, más conocidas como Subrutinas en el ambito de Perl, son uno de las dos instrucciones que necesitan declararse. A diferencia de los lenguajes de bajo nivel, al declarar una subrutina en Perl, no es necesario especificar ni el tipo de valor y cantidad de parámetros que esta recibira, ni los que devuelve, o bien si no se devuelve nada. La forma de hacerlo, es la siguiente:

```
sub foo
  { codigo }

sub foo(prototipo)
  { codigo }
```

Para llamar a la subrutina que hemos creado, simplemente hacemos:

```
foo argumentos;
foo(argumentos);
```

Las diferencias entre la llamada a una subrutina con o sin paréntesis, reside en el lugar en donde fue definida la misma. Si hacemos referencia a la última, antes de que esta halla sido declarada, debemos hacerlo con los paréntesis, de lo contrario se producira un error. Analogamente, si la subrutina fue definida líneas arriba de la primera llamada a ella, podra hacerse sin los paréntesis.

La forma de recibir y asignar los parámetros pasados a la subrutina, es mediante la utilización del array especial **@_**, array reservado exclusivamente para pasaje de argumentos, con lo cual, tenemos la siguiente situación:

```
foo($var1,$var2);

sub foo
{
  ($a, $b) = @_;
   ...
}
```

Existe otra forma de realizar una llamada a una subrutina, y es mediante la utilización del caracter **&**. Con esto, logramos pasar de forma implicita el array genérico de argumentos, **@_**. La llamada a la subrutina se realiza de la siguiente manera:

```
&foo;   # como referirse a foo(@_);
&foo(); # como referirse a foo();
```

Debido a que el array especial **@_** es declarado por el interprete como local al bloque en el cual uno se encuentra, se puede asumir que todos los parámetros en Perl son pasados por referencia a las subrutinas.

Es posible fijar un prototipo para una subrutina, o sea, fijar el tipo y cantidad de variables que esta puede recibir, de forma tal que uno sea consistente al generar una biblioteca y evitar el mal uso de la misma, o simplemente, el hecho de querer llevar un control más exacto sobre los argumentos que son pasados.

Para fijar el prototipo de una subrutina, que en general se especifica al principio del codigo y debe ir antes que la implementación, se utiliza la siguiente forma:

```
sub foo(proto);

sub foo(proto)
  { codigo }
```

Donde `proto` es cualquiera de los tipos de variables disponibles en Perl, más especificamente, **$**(escalar), @(array), %(hash), &, o *(typeglob). La sintaxis es la siguiente:

```
sub foo($$);   # Espera recibir dos escalares
sub foo($\@);  # Espera recibir un escalar y un array
sub foo($;\@); # Espera recibir un escalar y, opcionalmente, un array
```

La utilización del caracter
fuerza a la variable a ser del tipo especificado, y el valor que se pasa al array @_ es la dirección de memoria de la misma. Veamos un ejemplo:

```
# foo espera recibir una escalar y la dirección de
# memoria de una array y un escalar
sub foo($\@\$);

sub foo($\@\$)
{
  ($scalar, $ref_array, $ref_scalar) = @_

  print("scalar = $scalar\n");
  print("ref_scalar = $$ref_scalar\n");

  for($i = 0; $i < @$#ref_array; $i++)
    { print("$ref_array->[$i] \n"); }
}

foo(13,@array,$variable);
```

La utilización de prototipos, afecta exclusivamente a la *nueva* forma de realizar llamadas a subrutinas. Esta, implica que al hacer referencia a una, no se utilice el caracter &, sino que la llame mediante su nombre. Esto, obviamente, deja de lado todas las llamadas que implican el signo, como ser los punteros a subrutinas (referencias).

3.6. Alcance de las Declaraciones

Como se dijo anteriormente, las variables en Perl no necesitan ser declaradas o inicializadas. No obstante esto, es conveniente realizarlo, desde el punto de la optimización del espacio. Es común, utilizar una variable, y no declararla como local al bloque de codigo dentro del cual estamos trabajando, con lo cual la variable estara disponible dentro y fuera del bloque actual.

Cuando nos referimos a scripts pequeños, que no estan obligados a realizar un uso racional del alcance de las variables (y por ende, memoria), esto no tiene mucho significado. Sin embargo, es una buena practica realizarlo, debido a que el día en el cual nos encontremos con una situación que requiera un script complejo y largo, la utilización de los recursos comienza a ser importante.

No obstante esto, es posible utilizar un módulo que permite obligarnos a declarar toda variable que aparezca en el codigo, este módulo se llama **strict**.

Existen dos formas de lograr que una variable posea un alcance limitado, que es mediante la utilización de las declaraciones **local** y **my**.

3.6.1. local

Se utiliza para crear el llamado *alcance dinámico*. La declaración de variables de esta forma, genera un alcance global, pero solo dentro del bloque en el cual se esta trabajando. Para clarificar esto:

```
{ # Bloque de codigo
  local($var_local) = "Variable local";

  # Imprime: "Valor de la variable local: Variable local"
  print("Valor de la variable local: $var_local \n");

  # Imprime: "Valor dentro de función: Variable local"
  test_scope();

} # Fin de alcance de $var_local

# Imprime: "Valor de la variable local: "
print("Valor de la variable local: $var_local \n");

sub test_scope
  { print("Valor dentro de función: $var_local \n"); }
```

3.6.2. my

Al declarar una variable de esta manera, se especifica que la misma sera privada, exclusiva del bloque de codigo dentro de la cual se declara. Puntualmente:

```
{ # Bloque de codigo
  my($var_my) = "Variable privada";

  # Imprime: "Valor de 'my' variable: Variable privada"
  print("Valor de 'my' variable : $var_my \n");

  # Imprime: "Valor de 'my' variable: "
  test_scope();
} # Fin de alcance de $var_my

# Imprime: "Valor de la 'my' variable: "
print("Valor de 'my' variable: $var_my \n");

sub test_scope
```

```
{ print("Valor dentro de función: $var_my \n"); }
```

3.7. Expresiones Condicionales

Las expresiones condicionales, ejecutan un bloque de codigo, según la veracidad de una condición. Para ello, disponemos de las construcciones **if** y **unless**.

La construcción **if**, verificara si la condición enunciada entre paréntesis es verdadera, de serlo, ejecutará la porción de codigo dentro del *bloque1*. En caso que la condición no se cumpla, se ejecutarán las sentencias que se encuentren dentro del bloque del **else**.

```
if(condición)
  { bloque1 }
else
  { bloque2 }
```

En cambio, la construcción **unless** hará exactamente lo contrario al **if**. Ejecutará el *bloque1* si la condición no se cumple (una suerte de negación).

```
unless(condición)
  { bloque1 }
else
  { bolque2 }
```

3.8. Ciclos Iterativos

Los ciclos iterativos, en su forma más simple, ejecutan una serie de instrucciónes mientras cierta condición sea verdadera, posibilitando repetir una estructura de codigo.

La instrucción **while**, permite iterar dentro de un bloque, mientras una (o varias) condiciones sean evaluados a valores verdaderos. La sintaxis de la misma es:

```
while(condición)
  { bloque1 }
```

Por otra parte, la construcción **for**, permite la misma acción, con la salvedad que su sintaxis es un poco más extensa, y de más esta decir que, no es muy similar a la de C, es exactamente igual.

```
for(bloque1; condición/es; bloque2)
  { bloque de codigo }
```

A modo informativo, es analogo a las siguiente expresión:

```
bloque1;
while(condición/es)
{
   bloque de codigo;
   bloque2;
}
```

Finalmente, existe una construcción del lenguaje, típica de Perl, llamada **foreach**, que básicamente, itera sobre los elementos de una lista, asignando cada uno de los valores del array, a una variable de control, por ejemplo:

```
foreach variable (lista)
  { ... }
```

Con esto, se logra iterar a lo largo de una lista o hash, sin la necesidad de conocer los límites del mismo, ni tampoco, en el caso de arrays asociativos, los *keys* de las posiciones.

3.9. Expresiones alternativas

En caso de encontrarnos con simples instrucciones que deban ser ejecutadas, ya sea dentro de ciclos iterativos, o en base a expresiones condicionales, podemos utilizar la siguiente sintaxis:

```
instrucción if(condición);
instrucción unless(condición);
instrucción while(condición);
```

Por ejemplo:

```
# Llama a la subrutina si i vale 0
&call_sub() if($i == 0);

# Incrementa $i en uno mientras sea menor que 10
$i++ unless($i > 10);

# Imprime las líneas del archivo mientras no encuentre EOF
print($_ . "\n"); while(<FILE>);
```