- El Lenguaje Perl -
Practical Extraction Report Language

Marcelo Espinosa Alliende

27 de noviembre de 2002

Índice general

Capítulo 1

Tipos de Datos

1.1. Datos Escalares

Es el tipo de datos más simple que *perl* manipula. Un dato escalar es un número (tal como 354 o 3.2345e20) o un string de caracteres (tal como “Hola”, o “ este es un string”). Para perl la representación de un dato numérico o de texto es exactamente lo mismo.

Números

Todos los números usan el mismo formato internamente, ya sean números enteros o números de punto flotante, Perl internamente computa los valores en formato de punto flotante. Esto significa que no hay números con formato de enteros *internamente* en Perl. Una constante entera en perl es tratada internamente con su representación de número flotante. En términos simples no nos preocupamos del tipo de dato, ni de su representación, de hecho nunca notamos la conversión que perl realiza internamente al respecto.

Representación de Literales en punto flotante

Un literal es la forma en que un valor es representado en el texto de un programa Perl (eventualmente podría ser llamado una “constante”). Perl acepta el conjunto completo de literales de punto flotante disponibles en **C**, número con y sin punto decimal (incluyendo el prefijo opcional + o -), tambien como el símbolo de exponenciación . Por ejemplo:

```
1.25 # uno y un cuarto
7.25e45 # 7.25 en potencia de 10 elevado a 45
-6.5e24 # menos 6.5 en potencia de 10 elevado a 24
-12e-24 # menos 12 en potencia de 10 elevado a menos 24
```

Representación de Literales Enteros

La representación de enteros es muy sencilla :

```
12
34
```

```
-345
23240504055
```

Nunca se debe comenzar un numero entero con 0 (cero) dado que Perl soporta la representación de literales Octales y Hexadecimales. Los números Octales comienzan con 0, y los número hexadecimales comienzan con 0x. En hexadecimal los digitos A a la F representan los digitos convencionales entre el 10 y 15. Por ejemplo:

```
0377 # representa el número Octal 377, equivalente al 255 en Decimal.
-0xff # representa un número negativo hexadecimal, equivalente a 255 en Decimal
```

1.1.1. Representación de Strings

Los strings son secuencias de caracteres (como “hola”), cada caracter es un valor de 8 bits y pertenece a un set de 256 caracteres.

El string más corto posible es el que no tiene caracteres. El mayor string posible de representar llena toda la memoria. Esto va en concordancia con el principio que perl nos ofrece “sin incorporacion de limites”. Los string tipicos son secuencias de caracteres imprimibles, pero dado que perl permite manejar el set completo de 256 caracteres en un string, es posible crear, escanear y manipular datos binarios en bruto como strings, algo que sería muy dificil de realizar en otros lenguajes o utilitarios.

Al igual que los números, los strings tienen una representación literal, y los podemos definir con comillas simple ’string’ o comillas dobles “string”. Tambien es posible definir un string con la comilla simple ‘invertida‘, pero esto tiene un sentido especial para perl dado que le dice que debe ejecutar un comando externo del sistema.

Las comillas en sus diversas formas no forman parte del string, solo delimitan el comienzo y fin del string. Par representar las comillas simples dentro del string, estas deben ir precedidas por “\”. Por ejemplo : ’ el año 98\’ fue muy existoso’, obtenemos como resultado *el año 98’ fue muy exitoso.*

Las comillas dobles funcionan similarmente a los strings en lenguaje C, sin embargo ahora es posible utilizar caracteres de control (precedidas por un backslash \), o cualquier caracter a través de su representación Octal o Hexadecimal.

```
“Hola Mundo \n” # El string Hola Mundo y un retorno de carro
“Uno \t Dos” # El string Uno, un tabulador, el string Dos.
```

El backslash puede preceder muchos caracteres de control que realicen diferentes cosas. A continuación se listan algunas de las secuencias de control:

Otra característica muy importante de las comillas dobles es que pueden iterpolar variables, significando esto que variables escalares o arreglos dentro de los strings son reemplazados con sus actuales valores cuando el string es utilizado.[1]

1.1.2. Operadores Escalares

Un operador produce un nuevo valor (el resultado) desde uno o mas valores (los operandos).

Perl provee los típicos operadores (suma + , resta - , multiplicación * , división / , módulo % , exponenciación **). Los operadores de comparación lógica para valores numéricos son <, <=, ==, >=, >, !=:

[1] La representación de variables y arreglos será analizada en la sección 1.1.3, página 5.

Cuadro 1.1: Significado en la representación de un string de doble comilla.

Secuencia	**Significado**
\n	nueva lína
\t	Tabulador
\b	Retroceso
\\	Backslash
\l	transforma a minúscula la próxima letra
\L	transforma a minúscula todas las siguientes letras hasta \E
\u	transforma a mayúscula la próxima letra
\U	transforma a mayúscula las siguientes letras hasta \E
\Q	escapar todos los caracteres no alfanumericos hasta \E
\E	Termina \Q, \U, \L

Cuadro 1.2: Operadores de comparación numérica y de strings.

Comparación	**Números**	**Strings**
Igual	==	eq
Distinto	!=	ne
Menor que	<	lt
Mayor que	>	gt
Menor o igual a	<=	le
Mayor o igual a	>=	ge

```
3 + 5 # 8
2**3 # 8
2 > 5 # falso
2 != 3 # verdadero
```

Los string pueden ser concatenados con el operador punto. Por ejemplo:

```
“hola” . “mundo” # equivale a holamundo
“hola” . “ “ . “mundo” . “\n” # equivale a hola mundo \n
```

Otro conjunto de operadores para strings son los operadores de comparación, estos operadores utilizan el estilo Fortran. Estos operadores comparan los valores ASCII que representa los caracteres del string.

Otro operador interesante es el de repetición “x” (en minúscula). Este operador toma su operando izquierdo (un string), y realiza muchas copias concatenadas de ese string según la cantidad indicada por el operando derecho (un número). Por ejemplo:

```
“Hola” x 3 # HolaHolaHola
“abc” x (3+1) # equivalente a abc x 4 : abcabcabcabc
(3+2) x 3 # 555
```

1.1.3. Variables Escalares

Una variable en Perl es un contenedor que matiene uno o más valores. El nombre de la variable es una constante a través del programa y almacena una valor escalar simple (representando un número, un arreglo o una referencia). El formato de definición de la variable en Perl es **$nombre_variable** (comienza con una letra y puede contener más letras, digitos o underscore "_").

Las Variables son sensivas a las mayúsculas/minúsculas, luego **$Z** es distinta a **$z**.

Operador de asignación:

```
$b = 4 + ($a=3);
$b = $c = 5;
```

Operador de asignación binaria (+,-,*,/, %):

```
$a += 5; #equivale a $a = $a + 5
$a *= 4; #equivae a $a = $a * 5
```

Operador de autoincremento (+,-):

```
++$i; #incrementa el valor de $i y luego utiliza el valor para la operación deseada.
$i++; #utiliza el valor de $i, luego lo incrementa.
```

Interpolación de Variables:

Funciona cuando el string está delimitado con comillas dobles (”):

```
$a = ”mundo!!!”;
$b = ”hola $a”; # $b obtiene como resultado ”hola mundo!!!”
```

Una vez que el texto es interpolado, este no es reescaneado para buscar nuevos matchs de reemplazo, observemos el siguiente ejemplo:

```
$a = ’ Dolar : $dolar ’;
$b = ”Tipo de cambio ($a)”;
```

en este caso $b obtiene como resultado ”tipo de cambio (Dolar:$dolar)” dado que $a estaba delimitado con comillas simples no fue interpolada con el valor real de la variable $dolar, luego al interpolar la variable $a en la variable $b, este recibe el string de $a pero no es reescaneado para asignar la variable $dolar.

Delimitacion de variables dentro de strings:

Para delimitar el nombre de las variables dentro de un string y que esta no sea confundida con el resto del texto, utilizamos los delimitadores ”{” y ”}”, veamos el ejemplo siguiente:

```
$year = 80;
$a = ”estamos en la generación de los ${year}s” #obtenemos
”estamos en la generacion de los 80s”.
```

si no se hubiese delimitado la variable $year en el string, Perl la hubiese interpretado como **$years**, retornando **undef** dado que esa variable no existe.

Una alternativa para resolver un problema similar al presentado, es mediante el operado punto (”.”), el cual permite concatenar strings.

```
$a = ”estamos en la generación de los ” . $year . ”s”;
```

undef

cuando una variable es utilizada antes de que se le asigne un valor, esta retorna **undef**. Equivale a 0 cuando es usado como un número o un string de largo 0 (cero) cuando es usado en el contexto de un string, es decir ””.

1.2. Arreglos y Listas de Datos

Una lista es un conjunto de datos ordenados por algún criterio. Un arreglo es una variable que mantiene una lista. Ejemplos de listas:

```
(1, 2, 3);
(”hola”, 2, 7, 3.14);
($a, 17); # $a es reevaluado cada vez que el literal es utilizado
($a+$b, $d*$c, 2, 5, ”hola”);
(); # lista vacia
```

Constructor de listas

Se puede utilizar el operador ”**..**” para construir listas con una sintaxis más clara y sencilla, por ejmplo:

```
(1 .. 5 ); # equivale a (1, 2 , 3, 4, 5)
(1.2 .. 5.2); # equivale a (1.2, 2.2, 3.2, 4.2, 5.2)
(2 .. 5, 9, 15); # equivale a (1, 2, 3, 4, 5, 9, 15)
(”hola”, ”este”, ”es”, ”un”, ”arreglo”);
```

Este último ejemplo es dificil de digitar, pero podemos utilizar el delimitador de palabras **qw()** para simplificar la escritura de listas de strings, por ejemplo:

```
qw(este es un arreglo);
qw(
este
es
un
arreglo);
```

Variables tipo arreglo

Un arreglo es una variable que mantiene una lista de valores (cero o más valores). Estas variables comienzan con **@** y utilizan las mismas convenciones que las variables escalares para su definición. Ejemplo:

```
@alumnos;
```

Operadores y funciones de arreglos:

Asignación

```
@alumnos = (”juan”, ”pedro”, ”luis”, ”angelica”);
@nuevos_alumnos = @alumnos;
@alumnos = qw(juan jose ingrid);
@mas_alumnos = (4, 5, @alumnos, ”string”, 5.8);
($a, $b, $c) = ( 1, 2, 3);
($a, $b) = ($b, $a);
```

Si una variable tipo arreglo es asignado a una variable escalar, esta retorna el largo del arreglo.

```
$arreglo = (1, 2, 3, 4, ”hola”); $largo = @arreglo; # $largo=5
@z = ( @x = (2, 3, 4) ); #@z e @x obtienen el mismo valor, es
decir, 3
@z = @x = (2, 3, 4);
```

Acceso a elementos de los arreglos

Para acceder a los elementos del arreglo se utiliza el símbolo **$** dado que accedenos a un escalar individual dentro de este. El arreglo es referenciado por medio de un índice como todo lenguaje y este comienza con el valor 0 (cero) como primer elemento del índice, por ejemplo, se tiene el arreglo @alumnos, para acceder a los miembros de este arreglo utilizamos $alumnos[0] para acceder al primer miembro, $alumnos[1] para el segundo miembro y así sucesivamente.

```
@x = (5, 6, 7);
$b = $x[0]; # $b obtiene el valor de 5
$b = $x[2]; # $b obtiene el valor de 7
Acceso a trozos del arreglo
@x[0,1] # ====> $x[0], $x[1]
@x[2,3] = (5, 6); # asigna el valor 5 al tercer elemento y el valor
6 al cuarto elemento del arreglo
@x = (1, 2, 3); $x[4] = ”hola”; #@x = (1, 2, 3, undef, ”hola”);
```

Un vector se puede recorrer inversamente (desde el final hacia el principio) por medio de la utilización de un suscriptor o índice negativo, por ejemplo :

```
$alumnos[-1];
```

función push y pop

Operan desde el lado derecho de la lista y agregan (push) o extraen (pop) elementos de la lista.

```
push(@vector, elementos); # agrega los elementos al arreglo
llamado @vector
$old = pop ( @arreglo ); # en este caso pop extrae el último
elemento de la lista y devuelve dicho valor.
```

función shift y unshift

Operan desde el lado izquierdo de la lista y agregan (unshift) o extraen (shift) elementos de la lista.

```
unshift (@vector, $a) # agrega $a como primer elemento al arreglo
vector, desplazando el resto hacia la derecha.
$old = shift(@vector); extrae el primer elemento de la lista y se
lo asigna a la variable old.
```

funcion reverse

```
@b = reverse(@a) # asigna a la variable @b los elementos de @a pero en orden inverso.
```

funcion sort

Toma sus argumentos como si fuera un string y los devuelve ordenados.
función chomp
Extrae el caracter de nueva linea (o retorno de carro) desde el string. En el caso de operar sobre un arreglo, este extrae el caracter de nueva linea desde cada elemento del arreglo.

```
$z = ”hola mundo \n”;
chomp($z); # $z ahora es ”hola mundo”
@z = (”hola\n”, ”mundo\n”, ”!!!\n”);
chomp(@z) # @z ahora es (”hola”,”mundo”,”!!!”);
```

1.3. Hash

Es una colección de datos escalares con elementos individuales seleccionados por algún índice. Una variable hash comienza con un **%** y sigue la misma convención de nombres de una variable escalar. Luego tenemos 3 formas de almacenamiento de información; un escalar **$**, un arreglo **@**, un hash **%**, así **$alumnos** es distinto a **@alumnos** y **%alumnos.**
se tiene una variable hash denominada %x;

```
$x{’clave’} = valor;
$x{123.5} = ”5.4” # la clave es el string ”123.5”
$x{123.5} += 2 # 7.4
```

Representación literal de un hash

```
@lista =%x;
@lista ====> (clave, valor, clave, valor, clave, valor,...);
%y = @lista;
%x =%y;
```

función keys();

retorna las claves de un hash en una lista

```
@lista_de_claves = keys( %hash);
foreach $clave (keys( %hash)) {
  print $hash{$clave};
}
if (keys( %hash)) {
 estamentos;
}
while (keys( %hash) < 10 ) {
 estamentos;
}
if ( %hash) { # verdadero si hay algo en el hash
}
```

funcion values();

retorna sólo los valores de un hash

```
@lista_de_valores = values( %hash);
```

función each();

obtiene el par clave, valor

```
while (($clave, $valor) = each ( %hash) ) {
estamentos;
}
```

funcion delete

elimina elementos de un hash, esto es, su clave y valor.

```
delete $hash{"clave}
```

Hash slices (trozos)

```
$x{"m"} = 1;
$x{"e"} = 2;
$x{"r"} = 3;
```

otra forma de escribir lo mismo es utilizando slices;

```
($x{"m"}, $x{"e"}, $x{"r"}) = (1, 2, 3);
```

mejor de esta forma:

```
@x{"m","e", "r"} = ( 1, 2, 3);
```

o mejor aún:

```
@x{ qw(m e r) } = (1, 2, 3);
```

Capítulo 2

Estructuras de Control

```
{ # representación de un bloque
... estamentos
}
if (expresion) {
estamentos_verdaderos;
} [ else {
estamentos_falsos;
} ]
while (expresion) {
estamentos;
}
unless (expresion) {
estamentos;
}
do {
estamentos;
} while (expresion);
do {
estamentos;
} until (expresion);
for (init_exp; test_exp; re-init_exp) {
estamentos;
}
foreach $i (@arreglo) {
estamentos;
}
if (expresion) {
estamentos;
} elsif (expresion) {
estamentos;
}
```

Capítulo 3

Expresiones Regulares

3.1. Patrones

Una expresión regular es un patrón, algunas partes del patron coinciden con carecteres simples de un string, otras partes del patrón coinciden con multiples caracteres de un string.

3.1.1. Patrones de caracter simple

El más sencillo y común de los patrones de igualación de caracteres en una expresión regular es un caracter simple, es decir, al poner una letra **”a”** en una expresión regular requiere de una **”a”** en el string.

Generalmente el patrón es delimitado por el caracter **”/”** (slash), pero puede ser cambiado por cualquier caracter apropiado para la diferenciación. Ejemplo

```
if ( /a/ ) {
estamentos;
}
```

En general, el patrón de caracteres más utilizado es el **”.”**, indicado la coincidencia de cualquier caracter, excepto ”nueva linea” (o retorno de carro), por ejmplo **/a./** iguala cualquier secuencia de 2 letras que comienza con una **a** y no termina en retorno de carro, es decir, **am**, **au**, **a1**, **ax**, etc, pero no ”**a\n**”.

Cuando se desea que uno y solo uno de los caracteres debe estar presente en la parte correspondiente del string para la igualación, se utilizan la *definición de clases* por medio de la utilización de los caracteres **”[”** y **”]”**, por ejmplo, **/[aeiou]/** busca dentro del string una de las 5 vocales en *minúsculas*. /**aeiouAEIOU**/ opera de manera similar pero incluye ahora las vocales en Mayúsculas.

También es posible definir rangos para simplificar la creación de los patrones mediante el **”-”**. Esto ayuda por ejemplo a definir de manera eficiente un rango de letras o digitos amplio, por ejmplo desde la ”a” a la ”z”.

```
/[a-z]/
/[a-zA-Z]/
/[0-9]/
/[0-9\-]/ #caraceteres validos son del 0 al 9 y el signo menos,
nótese que fue escapado con ″\″
/[a-z0-9]/
```

Cuadro 3.1: Abreviaciones de clases predefinidas

Constructor	Clase equivalente	Constructor Negado	Clase negada equivalente
\d (un dígito)	[0-9]	\D (no digitos!)	[^0-9]
\w (una palabra)	[a-zA-Z0-9_]	\W (no palabras!)	[^a-zA-Z0-9_]
\s (caracter de espacio)	[\r\t\n\f]	\S (no espacio!)	[^ \r\t\n\f]

También existe la clase negada que busca los caracteres que no están en la lista, para eso se utiliza en la definición de la clase el simbolo **"^"**.

```
/[^0-9]/ # match a cualquier caracter que no sea un digito.
/[^a-z056\-_]/ # da como resultado cualquier caracter no
mencionado en la lista.
```

Para mayor conveniencia Perl predefine algunas clases de caracteres, las que observamos en la siguiente tabla.

3.1.2. Patrones de Agrupación

El verdadero poder de las expresiones regulares resulta cuando decimos "uno o más de esto" o "hasta 5 de aquello".

Secuencia

Es probablemente el menos obvio, esto significa por ejemplo que **abc** equivale a una **a b** seguido por una **c**.

Multiplicadores

Simbolizado por el asterisco (*****), indica cero o más del caracter inmediatamente previo (o clase de caracter). Otros dos patrones de agrupación que operan de forma similar son el signo más (**+**), indicando uno o más del caracter inmediatamente previo, y el signo de interrogación (**?**), significando cero o uno del caracter inmediatamente previo. Por ejemplo, la expresión regular **/ho+la ?mu+ndo/** equivale a la **"h"** seguido por una o más "**o**" seguido por un **"la"** seguido por cero o un espacio, seguido por una **"m"** y una o más **"u"** seguido por **"ndo**", es decir, la expresión hoooola muuuundo es valida para el patrón de búsqueda.

Existe una forma de especificar un número determinado de equivalencias requeridas con el **multiplicador general**, este consiste de un par de llaves con uno o 2 números adentro, por ejemplo /x{5,10}/ indicando que el caracter **x** debe ser encontrado dentro del rango especificado (de 5 a 10). Un formato forma generica para definir este multiplicador general es {min, max}, donde **min** especifica el número mínimo de veces que debe aparecer el caracter buscado y **max** el número máximo de veces que debe aparecer.

{0,} es equivalente a *****
{1,} es quivalente a **+**
{0,1} es equivalente a **?**
Pero cláramete es más sencillo utilizar los caracteres de puntuación (*,+,?).

Capítulo 4

CGI’s en Perl

4.1. Formularios

Abriendo y cerrando un formulario

```
print $query->startform([$method],[$action],[$encoding]);
Elementos del formulario...
print $query->endform;
```

- **$method** puede ser ”POST”, ”GET” aunque generalmente se utiliza POST
- **$action** indica el URL que procesara el formulario

Campo tipo texto

```
print $query->textfield(-name=>’nombre_del_campo’,
 -default=>’Valor de inicio’, -size=>50, -maxlength=>80);
```

Cuadro de texto grande

```
print $query->textarea(-name=>’nombre_del_campo’,
      -default=>’Valor de inicio’,
      -rows=>10,
      -columns=>50);
```

Campo de Password

```
print $query->password_field(-name=>’nombre_del_campo’,
      -value=>’Valor de inicio’,
      -size=>50,
      -maxlength=>80);
```

Menú colgante (popup)

```
1.- Método completo
print $query->popup_menu(-name=>'nombre_del_campo',
      -values=>[qw/eenie meenie minie/],
      -labels=>{'eenie'=>'one',
                'meenie'=>'two',
                'minie'=>'three'},
      -default=>'meenie');
2.- Método simple
print $query->popup_menu(-name=>'nombre_del_campo',
      -values=>['eenie','meenie','minie'],
      -default=>'meenie');
```

Lista desplazable (scrolling)

```
print $query->scrolling_list(-name=>'nombre_del_campo',
      -values=>['eenie','meenie','minie','moe'],
      -default=>['eenie','moe'],
      -size=>5, -multiple=>'true',
      -labels=>\%labels);
```

- **%labels** es un arreglo asociativo (hash) que define las etiquetas de los elementos, es decir, la parte visible de la lista.

Un Check Box

```
print $query->checkbox(-name=>'nombre_del_campo',
      -checked=>'checked',
      -value=>'ON',
      -label=>'Activa este Check Box!');
```

Un grupo de Check box'es relacionados

```
print $query->checkbox_group(-name=>'nombre_de_grupo',
      -values=>['eenie','meenie','minie','moe'],
      -default=>['eenie','moe'], -linebreak=>'true',
      -labels=>\%labels);
```

- **%labels** es un arreglo asociativo (hash) que define las etiquetas de los elementos, es decir, la parte visible de la lista.

Un grupo de botones Radio

```
print $query->radio_group(-name=>'nombre_del_grupo',
      -values=>['eenie','meenie','minie'],
      -default=>'meenie',
      -linebreak=>'true',
      -labels=>\%labels);
```

Un botón de Enviar

```
print $query->submit(-name=>'nombre_boton',
      -value=>'Enviar!');
```

Un botón de Reset (Limpiar el formulario)

```
print $query->reset(-value=>"Limpiar");
```

Un campo escondido

```
print $query->hidden(-name=>'nombre_campo_escondido',
      -default=>['valor1','valor2'...]);
```

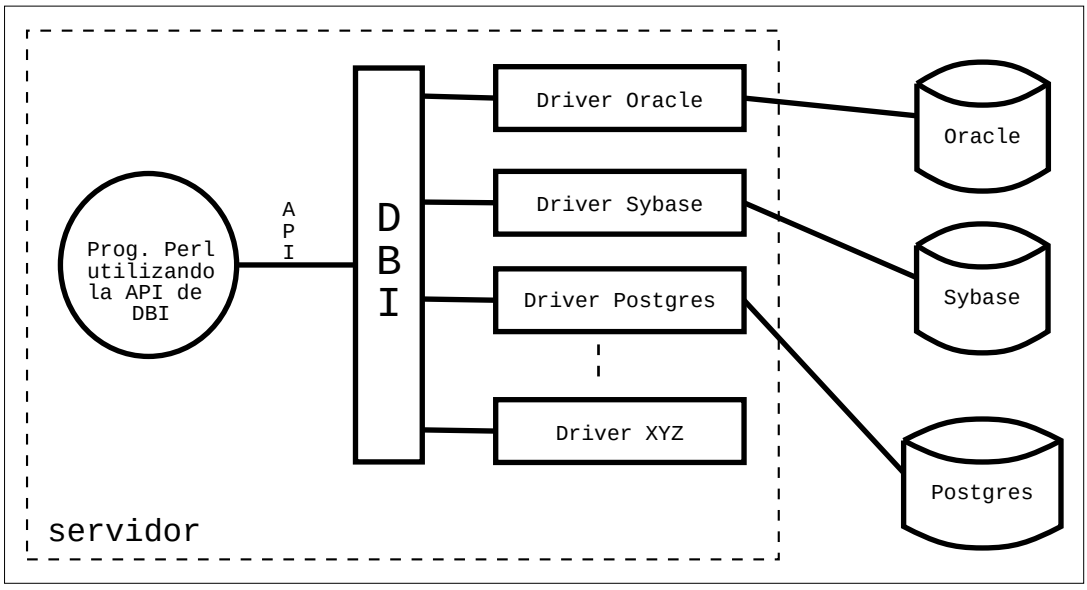
Capítulo 5

Módulo DBI -Acceso a Bases de Datos

5.1. Interactuando con una Base de datos

Para establecer una conexión a una base de datos en perl se debe utilizar el módulo DBI (database independent interface). DBI define un conjunto de métodos, variables, y convenciones que proveen una interfaz consistente, independiente de la base de datos que se utilice.

Es importante recordar que DBI es sólo una interfaz, esto es, DBI es una capa *pegamento* entre una aplicación y uno o más drivers de base de datos, finalmente es el módulo del driver de la base de datos el que realmente realiza la mayor parte del trabajo de nteracción con la BD.



Preparando la conexión a la BD

```
use DBI;
$driver=”mysql”; #puede ser cualquier otro debidamente
                  instalado en el servidor en donde reside el
                  programa CGI (oracle, sybase, access, otros)
$database=”proyectos”;
$hostname=”servidor.dominio.cl”;
$dsn = ”DBI:$driver:database=$database;host=$hostname;[port=$port]”; #port es opcional
$username = ”foo”;
$password = ””;
# establece la conexión con la BD
$dbh = DBI->connect($dsn, $user, $password, {RaiseError => 1, AutoCommit=0})
```

DBI->connect realiza la conexión real a la base de datos devolviendo un descriptor o handle de la conexión para trabajar con ella. La declaración de la conexión consiste básicamente en definir un DSN (data source name, o una fuente de datos), definido por el driver que maneja la conexión, una base de datos en donde residen los datos requeridos por la aplicación, un servidor de base de datos, y el puerto TCP (parámetro opcional) que define donde está corriendo el motor de la BD. Por otro lado está la definición del usuario y el password de este en la base de dato (se definen sólo si existen y son relevantes para el tipo de BD, por ejemplo al utilizar CSV -archivos delimitados por coma, tabulador u otro- no es necesario especificar user/password dado que no tiene sentido por tratarse de ”archivos planos” de texto puro, por lo tanto en este caso se especifica *”undef”*). Por último existe la opción de capturar los errores del motor de BD para el control de transacciones habilitando la opción *RaiseError* y el control de transacciones, que nos dice si los cambios se aplican de inmediato en la BD (autocommit=>1), o serán controladas por la aplicacion mediante commit’s o rollbacks (autocommit=>0).

> Dado que la conexión a una base de datos es un proceso caro, generalmente la conexión se abre al comienzo del programa y se cierra al final del mismo.

Ejecutando sentencias tipo SELECT

```
$sth = $dbh->prepare($statement);
```

ejemplo:

```
$sth = $dbh->prepare (”select * from costos”);
....
$sth->execute();
while ( @row = $sth->fetchrow_array ) { #recupera los datos
      print "@row\n"; #imprime los resultados
}
$sth = $dbh->prepare(”select centro_costo, cta_egreso from egresos where rut = ?”);
....
$rut = ”1111111-1”;
....
$sth->execute($rut);
while ( @row = $sth->fetchrow_array ) { #recupera los datos
```

```
      print "@row\n"; #imprime los resultados
}
```

El primer ejemplo prepara una sentencia SQL para ser ejecutada posteriormente en algún lugar del código CGI, esta no recibe parametros de ningun tipo. El método que ejecuta el código preparado es ”execute”.

El segundo ejemplo utiliza una consulta que recibe como parametro el RUT y este es enviado una vez que es llamado el método de execución de la consulta.

Ejecutando sentencias distintas a SELECT

```
$rv = $dbh->do($statement);
$rv = $dbh->do($statement, \%attr);
$rv = $dbh->do($statement, \%attr, @bind_values);
```

ejemplo:

```
$rc = $dbh->begin_work; #prepara el inicio de la transacción
$filas_afectadas = $dbh->do("UPDATE tabla SET foo = foo + 1");
$rc = $dbh->commit; # commit si no hubieron errores, $dbh->rollback para deshacer los cambios.
```

En este caso la sentencia $dbh->do(”...”) lleva implicita una preparación y una ejecución.

Recuperando datos de las consultas SELECT

”fetchrow_array” (una de las muchas formas de recuperar datos -la más sencilla)

```
@ary = $sth->fetchrow_array;
```

Recupera la próxima fila de datos y los retorna como una lista conteniendo los valores de los campos. Los campos nulos son retornados como valores ”undef” dentro de la lista. Si es que no hay más filas (o un error ocurre en el proceso), entonces fetchrow_array retorna una lista vacia. Se debe chequear **$sth->”err”** posteriormente para descubrir si la lista vacia fue producto de un error.

ejemplo:

```
while ( @row = $sth->fetchrow_array ) { #recupera los datos
      print "@row\n"; #imprime los resultados
}
#para recuperar el primer y segundo campo de la fila...
while ( ($rut, $nombre) = $sth->fetchrow_array[0,1]) { #recupera los datos
      print "RUT: $rut\t NOMBRE: $nombre\n"; #imprime los resultados
}
```

Control de Transacciones

Esto aplica a las bases de datos que soportan transacciones y donde AutoCommit esta en off (->0). La manera recomendada de implementar transacciones robustas en perl es mediante el uso de ”RaiseError” y eval {....}. Ejemplo:

```
$dbh->{AutoCommit} = 0; # habilita transacciones si es posible
$dbh->{RaiseError} = 1; # habilita el control de errores
eval {
   foo(...) # codigo de la aplicacion
   bar(...) # incluye, insert's, update's, delete's
   baz(...) #
   $dbh->commit; # commit!!! si hemos llegado hasta este punto
                   no han ocurrido errores por lo tanto aplicamos
                   las transacciones en la BD en forma permantente!
}
if ($@) {
   warn "Transacción abortada debido a $@";
   $dbh->rollback; # deshacemos la transaccion completa
   # más codigo de la aplicación
}
```

Dado que la secuencia de instrucciones perl está dentro del contexto de evaluación (producto de eval {...} y RaiseError -detener ante un error!), si ocurre un error, este bloque definido por eval es terminado, de inmediato se chequea la existencia del error (if ($@)...) para deshacer los cambios producidos en la base de datos. Por el contrario, si no hay errores en las transacciones definidas dentro del bloque eval {...} entonces los cambios son aplicados (commit) y se termina el bloque de evaluación, de inmediato se chequea la existencia de errores, en este caso no existe el error, por lo tanto no se hace rollback.

"commit"

```
$rc = $dbh->commit or die $dbh->errstr;
```

Commit (hacer permanente) la serie de cambios más reciente si es que la base de datos soporta transacciones y AutoCommit esta en off.

"rollback"

```
$rc = $dbh->rollback or die $dbh->errstr;
```

Rollback (deshacer) la serie de cambios más recientes a la base de datos (commit aún no aplicado), si es que la base de datos soporta transacciones y AutoCommit está en off.

"begin_work"

```
$rc = $dbh->begin_work or die $dbh->errstr;
```

Habilita Transacciones (seteando "AutoCommit" en off) hasta la próxima llamada a commit o rollback. Después del próximo commit o rollback, AutoCommit será cambiado a ON nuevamente. Esto asume que no se explicita formalmente $dbh->{AutoCommit} = 0;

Obteniendo información de las sentencias

"rows"

```
$rv = $sth->rows;
```

Retorna el número de filas afectadas por el último comando ”que afectó filas”, o devuelve -1 si el número de filas no es conocido o no está disponible.

"err"

```
$rv = $h->err;
```

Retorna el código de error nativo del motor de base de datos desde la última llamada al método del driver involucrado. El código es típicamente un entero, pero nunca se debe dar por asumido.

"errstr"

```
$str = $h->errstr;
```

Retorna el ”mensaje” de error nativo del motor de base de datos (string) desde la última llamada al método del driver involucrado.

"state"

```
$str = $h->state;
```

Retorna el código de error en el formato estándar SQLSTATE de 5 caracteres. Notar que el código específico de éxito ”00000” es traducido a ” (falso). Si el driver no soporta SQLSTATE (muchos no lo soportan), entonces el valor retornado será ”S1000” (error general) para todos los errores.

Desconectando la Base de Datos

"disconnect"

```
$rc = $dbh->disconnect or warn $dbh->errstr;
```

Desconecta la base de datos desde el descriptor de BD. ”disconnect” es utilizado típicamente cuando se sale del programa, el descriptor es de poco uso después de la desconexión.

Capítulo 6

Como obtener ayuda de las funciones y módulos ?

Para conocer Perl, es necesario partir por el manual de perl (aunque existe mucha documentación en línea al respecto, ver www.perldoc.com).

```
$ man perl
```

Al ejecutar el comando previo se describe la sintaxis "*de la línea de comandos de Perl*" más una serie de referencias a manuales que pueden ser consultados directamente y que dicen relación con los diferentes aspectos del lenguaje, por ejemplo podemos mencionar algunos

- Aspectos preliminares del lenguaje
  - **perl** Perl overview (this section)
  - **perlintro** Perl introduction for beginners
  - **perltoc** Perl documentation table of contents
- Tutoriales
  - **perlreftut** Perl references short introduction
  - **perldsc** Perl data structures intro
  - **perllol** Perl data structures: arrays of arrays
  - **perlrequick** Perl regular expressions quick start
  - **perlretut** Perl regular expressions tutorial
  - **perlboot** Perl OO tutorial for beginners
  - **perltoot** Perl OO tutorial, part 1
  - **perltooc** Perl OO tutorial, part 2
  - **perlbot** Perl OO tricks and examples
  - **perlstyle** Perl style guide

- **perltrap** Perl traps for the unwary perldebtut Perl debugging tutorial

- Manual de Referencia del Lenguaje
  - **perlsyn** Perl syntax
  - **perldata** Perl data structures
  - **perlop** Perl operators and precedence
  - **perlsub** Perl subroutines
  - **perlfunc** Perl built-in functions
  - **perlopentut** Perl open() tutorial
  - **perlpacktut** Perl pack() and unpack() tutorial
  - **perlpod** Perl plain old documentation
  - **perlpodspec** Perl plain old documentation format specification
  - **perlrun** Perl execution and options
  - **perldiag** Perl diagnostic messages
  - **perllexwarn** Perl warnings and their control
  - **perldebug** Perl debugging perlvar Perl predefined variables
  - **perlre** Perl regular expressions, the rest of the story
  - **perlref** Perl references, the rest of the story perlform Perl formats
  - **perlobj** Perl objects perltie Perl objects hidden behind simple variables

- etc...

Es posible ver directamente alguno de estos manuales invocandolos directamente de la línea de comandos, ej.

```
$ man perfunc
(muestra las funciones internas del lenguaje)
$ man perlre
(muestra la documentación acerca de las expresiones
regulares en perl)
```

Es posible también conocer la documentación específica de un módulo con el comando ”perldoc”, ejm.;

```
$ perldoc CGI
$ perldoc GD::Graph
$ perldoc DBD::Pg
```

Para conocer el uso de las funciones internas de perl (incorporadas)

```
$ perldoc -f localtime
$ perdoc -f open
$ perldoc -f socket
$ perldoc -f rand
```

Capítulo 7

Cpan

CPAN es una gran colección de software y documentación Perl que puede ser bajada e incorporada a la biblioteca de módulos de la instalación local. CPAN puede ser explorado ya sea desde http://www.cpan.org/, http://www.perl.com/CPAN/ o cualquier mirror listado en http://www.cpan.org/SITES.html y http://mirror.cpan.org/.

Es necesario notar además de que CPAN es el nombre de un módulo -CPAN.pm- y es utilizado para bajar e instalar software Perl desde el repositorio CPAN, normalmente se utiliza de la siguiente forma

```
# perl -MCPAN -e shell
```

- Para obtener ayuda de como usar CPAN

  ```
  cpan> ?
  ```

- Se puede buscar los módulos con los comandos

  ```
  a,b,d,m WORD or /REGEXP/
  respectivamente se busca acerca de autores (a),
  paquetes (b), distribuciones (d), modulos (m), ej.
  cpan> m /barcode/
  cpan> a /smith/
  ```

- Instalar modulos

  ```
  cpan> install GD::Barcode
  ```

- Para bajar módulos pero sin instalarlos (quedan almacenados en el directorio .cpan)

  ```
  cpan> get Net::Time
  ```

- Para instalar directamente desde la línea de comandos sin interactuar con el shell de CPAN

  ```
  perl -MCPAN -e 'install Chocolate::Belgian'.
  ```

Capítulo 8

Bibliografía recomendada

Para iniciados

- **Beginners Learning Perl (3rd ed)** by Randal L. Schwartz, Tom Phoenix. 316 pages. O'Reilly & Associates. (July 15, 2001).
- **Beginning Perl** by Simon Cozens, Peter Wainwright. 700 pages. Wrox Press Inc. (May 25, 2000).
- **CGI Programming** with Perl (2nd ed) by Scott Guelich, Shishir Gundavaram, Gunther Birznieks, Linda Mui. 451 pages. O'Reilly & Associates. (January 15, 2000).

Escencial

- **Programming Perl, 3rd** ed by Larry Wall, Tom Christiansen, Jon Orwant. 1092 pages. O'Reilly & Associates. (July 2000).
- **The Perl Cookbook** by Tom Christiansen, Nathan Torkington, Larry Wall. 794 pages. O'Reilly & Associates. (August 1998).

Avanzado

- **Advanced Perl Programming** by Sriram Srinivasan. 434 pages. O'Reilly & Associates. (August 1997).
- **Effective Perl Programming** by Joseph N. Hall. 288 pages. Addison-Wesley Pub Co. (January 1998).
- **Programming the Perl DBI** by Alligator Descartes, Tim Bunce. 346 pages. O'Reilly & Associates. (February 2000).
- **Mastering Regular Expressions**, 2nd edition by Jeffrey E. Friedl. 496 pages. O'Reilly & Associates. (July 15, 2002).